Г.И.Шпаковский, В.И.Стецюренко, А.Е. Верхотуров,
Н.В. Серикова
Белорусский государственный университет

Применение технологии MPI в Грид

Минск
2008 , март

УДК 681.3.06 +519.68

Настоящий материал «Применение технологии MPI в Грид» представлен в виде лекций, которые в течение длительного времени читались студентам радиофизического факультета Белорусского государственного университета (г. Минск). Изучение ограничивалось мелкозернистым параллелизмом (суперскалярные МП и технология ММХ) и заканчивалось кластерами для вычислений на основе стандарта MPI. Сейчас к этим лекциям добавлены обзорные материалы по распределенным вычислениям и лекции приобрели завершенную содержание: мелкозернистый (микропроцессорный) параллелизм, кластерный параллелизм (стандарт MPI) и параллелизм в Грид. Для применения MPI в Грид используется пакет MPICH-G2, при этом достигается максимальное распараллеливание решения задачи.

В лекциях в большей степени рассматривается организация параллельных систем (аппаратура, системные средства), а прикладные параллельные алгоритмы используются только для демонстрации возожностей этих систем.

По тематике параллельных вычислений и сетей изданы фундаментальные работы, в частности, книга В.В.Воеводина и Вл.В Воеводина «Параллельные вычисления» [1], книга В.Г. Олифер и Н.А. Олифер «Комьютерные сети» [2], существует много известных сайтов [3 – 9]. Настоящие же лекции предназначены для первичного ознакомления со всеми уровнями и основными механизмами параллельной обработки.

Оглавление

Организация вычислительных систем

Мелкозернистый параллелизм

Векторные ЭВМ (SIMD – архитектуры)

Кластеры (MIMD - архитектуры)

Распределенные вычисления

Лекция 1. Принципы построения быстродействующих ЭВМ

1.1. Большие задачи

Понятие «большие задачи» определяется временем решения задачи, которое зависит от количеством вычислительных операций в задаче и быстродействием вычислительных машин. Естественно, с ростом быстродействия вычислительных машин растет и размер решаемых задач. Для сегодняшних суперЭВМ доступными являются задачи с числом операций до 10^{12} -10^{15} операций с плавающей запятой.

На примере моделирования климата покажем, как возникают большие задачи. Климатическая система включает атмосферу, океан, сушу, криосферу и биоту. В основе климатической модели лежат уравнения динамики сплошной среды и и уравнения равновесной термодинамики.В модели также описываются все энергозначимые физические процессы: перенос излучения в атмосфере, фазовые переходы воды, облака и конвекция, перенос малых газовых примесей и их трансформация, мелкомасштабная диффузия тепла и диссипация кинетической энергии и многое другое. Рассмотрим только часть климатической модели – модель атмосферы.

Предположим, что моделирование обеспечивает период 100 лет. При построении алгоритмов используется принцип дискретизации. Атмосфера разбивается сеткой с шагом 1 градус по широте и долготе на всей поверхности земного шара и 40 слоями по высоте. Это дает около 2.6×10^6 элементов. Каждый элемент описывается примерно 10 компонентами. Следовательно, в любой фиксированный момент времени состояние атмосферы характеризуется ансамблем из 2.7×10^7 чисел. Условия моделирования требуют нахождения всех ансамблей через каждые 10 минут, то есть за 100 лет нужно определить около 5.3×10^4 ансамблей. Итого, за *один* численный экперимент приходится вычислять 1.4×10^{14} значимых результатов промежуточных вычислений. Если считать, что для получения каждого промежуточного результата требуется 10^2 - 10^3 арифметических опреаций, то проведение одного эксперимента с моделью атмосферы тербуется выполнить 10^{16} – 10^{17} операций с плавающей запятой.

Напомним некоторые обозначения, используемые для больших машин.

Обозначение	Величина	Достигнутый уровень
Кило	10^3	
Мега	10^6	
Гига	10^9	ПЭВМ до 5 Гфлопс
Тера	10^{12}	СуперЭВМ - Тфлопс
Пета	10^{15}	

Эта система обозначений относится к измереню частоты – герцы, объему памяти – байты, к количеству плавающих операций в секунду – флопс (flops – float hjint operations per second).

Следовательно, расчет атмосферы на персональной ЭВМ займет многие тысячи часов. А для полной модели земли и множества вариантов моделирования время возрастет на порядки. Значит, проводить моделирование на ПЭВМ - утопия, но на суперЭВМ – можно.

Как поведет себя земля через 100 лет – это для сегодняшних людей философия, не более. Но есть и более существенные, сегодняшние задачи, например, проблема разработки ядерного оружия. После объявления в 1991 году моратория на испытания ядерного оружия остался единственный путь совершенствования этого оружия – численное моделирование экспериментов. По количесву операций такое моделирование соизмеримо с моделированием климата. Но это касается больших стран – США, Россия. А зачем суперЭВМ малым странам, например, Беларуси.

Приведем пример задачи расчета дозвукового обтекания летательного аппарата сложной конструкции. Математическая модель требует задания граничных условий на бесконечности, однако реально она берется конечной. На практике это составляет десятки длин размера аппарата. Значит, область оказывается трехмерной и большой. При построении алгоритма численного решения используется принцип дискретизации. Из-за сложной конфигурации летательного аппарата разбие-

ние выбирается очень неоднородным. Общее число элементов, на которое разбивается область, определяется сеткой с числом шагов порядка 10^2 по каждому измерению, то есть будет всего 10^6 элементов. В каждой точке надо знать 5 величин . Следовательно, на одном временном слое неизвестных будет равно 5.10^6 . 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 $10^2 – 10^3$ операций. И все это только для одного варианта компановки и режима обтекания. А всего нужно провести расчеты для десятков вариантов. Приблизительные оценки показывают, что для решения такой задачи требуется выполнить 10^{15} -10^{16} операций с плавающей запятой.

Большое количество вычислительных моделей строится на базе решения СЛАУ, при этом размеры решетки (число уравнений) может достигать многих тысяч. Например, при моделировании полупроводниковых приборов число уравнений может быть равно $N=10^4$. Известно, что решение такой системы требует порядка N^3 вычислений по 10^2 плавающих операций для каждого вычисления. Тогда общее время расчета одного варианта моделирования будет порядка 10^{14} операций.

Число больших задач в разных областях науки, техники, экономики стремительно растет, поскольку вычислительный эксперимент значительно дешевле и информативнее натурного.

1.2. Методы повышения быстродействия компьютеров

Естественно, для решения больших задач нужны все более быстрые компьютеры. Повысить быстродействие можно:

1. За счет использования конвейеров команд и арифметических конвейеров.
2. За счет повышения быстродействия элементной базы (тактовой частоты).
3. За счет увеличения числа одновременно работающих в одной задаче ЭВМ, процессоров, АЛУ, умножителей и так далее, то есть за счет параллелизма выполнения операций.

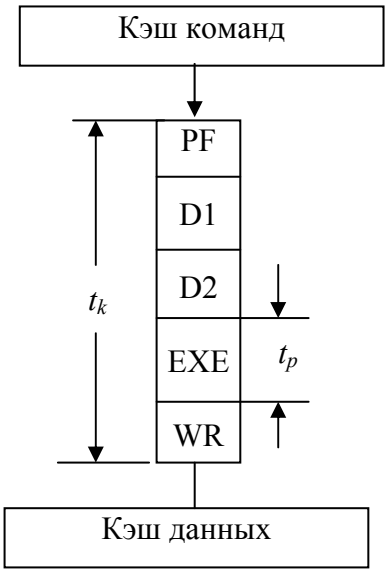
Конвейеры

Для примера рассмотрим конвейер команд известного микропоцессора Pentium. Конвейер содержит следующие ступени (рис. 1.1):

- ступень предвыборки PF (Prefetch), которая осуществляет упреждающую выборку группы команд в соответствующий буфер;
- ступень декодирования полей команды D1 (Decoder 1);
- ступень декодирования D2 (Decoder 2), на которой производится вычисление абсолютного адреса операнда, если операнд расположен в памяти;
- на ступени исполнения EXE (Execution) производится выборка операндов из РОН или памяти и выполнение операции в АЛУ;
- наконец, на ступени записи результата *WR* (Write Back) производится передача полученного результата в блок РОН.

В таком конвейере на разных ступенях выполнения находится 5 команд. После очередного такта на выходе конвейера получается новый результат (каждый такт), а на вход выбирается новая команда. В идеальном случае быстродействие микропроцессора возрастает в 5 раз.

Время такта конвейера определяется временем срабатывания самой медленной ступени, которойкак правило является ступень выполнения арифметических действий EXE. Например, умножение 2-ух чисел по 32 разряда требует 32 сложений. Чтобы уменьшить время умножения (и других сложных операций) ступень EXE в свою очередь разбивается на ступени с меньшим тактом. Такой конвейер называется арифметическим. В современных процессорах как правило наряду с конвейером команд используется также арифметический конвейер. Конечно, конвейеризация обладает ограниченными возможностями повышения быстродействия, но, вследствие малых затрат на ее реализацию, она всегда применяется на нижних уровнях схемотехники процессоров.



$t_p < t_k$

Рис.1.1 Схема конвейера команд. Здесь: t_p – время получения результата, t_k – полное время выполнения команды.

Повышения быстродействия элементной базы

Допустимая частота зависит от энергии переключения транзистора, а энергия – от размеров транзистора. Физическим порогом считается размер шага – 0.05 мкм, что соответствует десяткам и сотням Ггц, но это через десятки лет, а сегодняшнее положение определяется приведенной ниже таблицей. Из-за высокой стоимости оборудоваия переход от диапазона к диапазону совершается медленно. При этом для каждого диапазона существует предельная частота элементной базы.

Разрешение, мкм	Годы	F(max)	Число транзисторов	Стоимость завода
2	1980 – 1985	10 мгц	30 тыс	
1	1985 - 1990	40 мгц	250 тыс	$ 200 млн
0.6	1990 – 1995	0.1 ггц	1 млн	
0.35	1995 – 1998	0.4 ггц	10 млн	$ 2.4 млрд
0.18	1998 – 2000	0.9 ггц	40 млн	$ 10 млрд
0.1	2005	1.2 ггц	1 млрд	

Нанотехнология — это область прикладной науки и техники, имеющая дело с объектами размером менее 50 - 100 нанометров (1 нанометр равен 10^{-9} метра). Нанотехнология качественно отличается от традиционных инженерных дисциплин, поскольку на таких масштабах привычные, макроскопические, технологии обращения с материей часто неприменимы, а микроскопические явления, пренебрежительно слабые на привычных масштабах, становятся намного значительнее: свойства и взаимодействия отдельных атомов и молекул, квантовые эффекты. В практическом аспекте это технологии производства устройств и их компонентов, необходимых для создания, обработки и манипуляции частицами, размеры которых находятся в пределах от 1 до 100 нанометров.

Но сейчас речь не идет о квантовых ЭВМ, а только о том, что надо учитывать квантовые взаимодействия при разработке кристаллов. Большую роль играют размеры кристаллов.

Тактовая частота определяется также размерами конструктива, ниже на рисунке представлена схема совпадения, которая срабатывает при одновременном появлении двух сигналов. Один сигнал поступает непосредственно от генератора, другой – с задержкой, определяемой размером пла-

ты, на котоой размещена сжема. Учитыая скорость света, нетрудно вычислить задержку сигнала на входе схемы И. Кроме того, сигнал должен некоторое время держаться на выходе схемы. Таким образом, для большой платы период генератора составит 60 нс, а тактовая частота – 15 Мгц, соответственнно для платы ПЭВМ и кристалла СБИС частота будет 150 Мгц и 3Ггц.

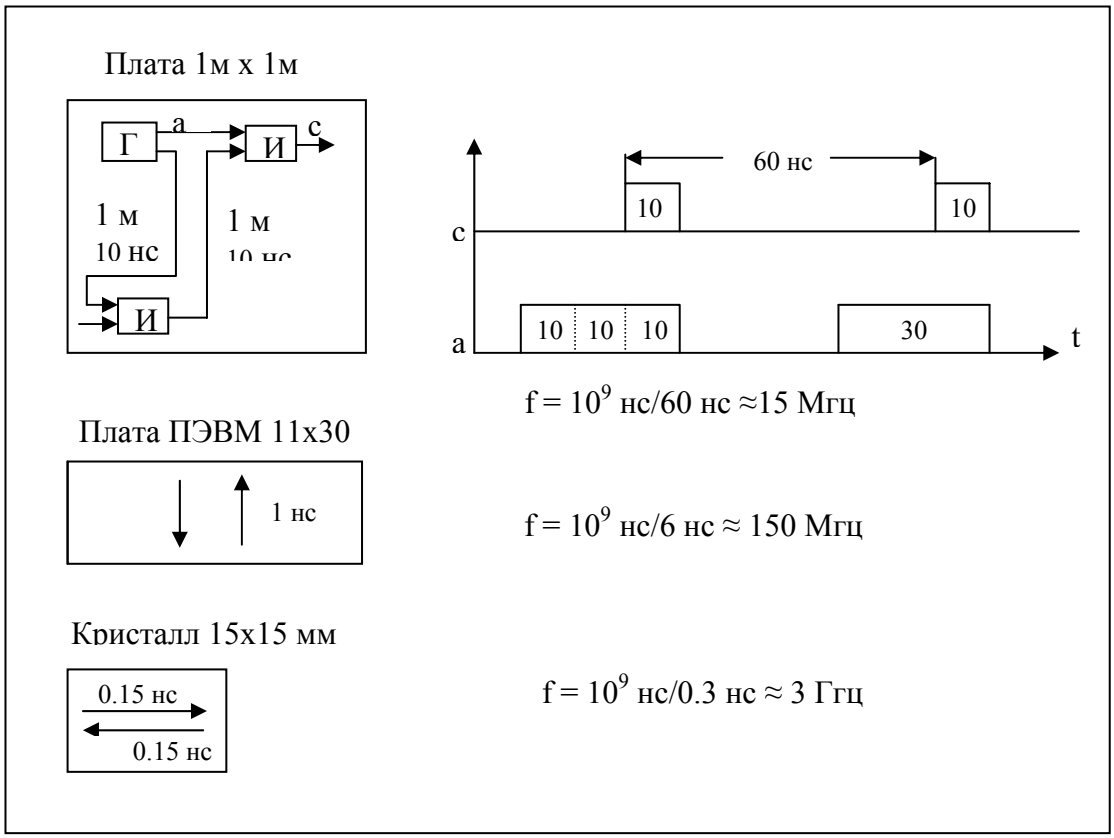


Рис.1.2. Рост частоты синхронизации с умешением раззмеров схемы

Повышение тактовой частоты является важным средством повышения быстродействия компьютеров, но ограничено фундаментальными физическими законами и стоимостью производства.

Параллельные вычисления.

В приведенной ниже таблице показано количество транзисторов, необходимое дла построения различных узлов компьютера.

N	Структура	Число транзисторов
1	Вентиль	4 -5
2	Триггер	25
3	Сумматор (32)	30000
4	16 РОН	16000
5	АЛУ	36000
6	Процессор (32)	300000

С другой стороны, существует известный закон Г. Мура (ранее глава и создатель компании Intel), который гласит: **каждые 2 года количество транзисторов на кристалле удваивается.**

Сейчас это количество составляет 100 – 200 млн транзисторов. Куда девать эти транзисторы? – Только на параллелизм.

Параллелизм — это возможность одновременного выполнения более одной арифметико-логических или служебных операций. Параллельная машина содержит множество процессоров П, объединенных сетью обмена данными (рис.1.3)

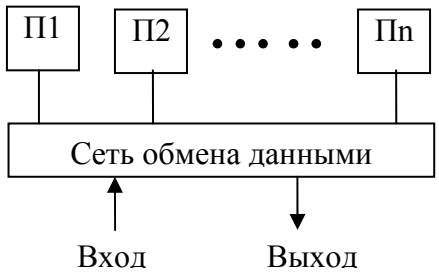


Рис.1.3. Схема параллельной системы

Таким образом, возможности микроэлектроники по увеличению числа процессоров на одном кристалле и возможности технологии параллеизма позволяют неограниченно повышать быстродействие вычислительных средств, необходимых для решения больших задач.

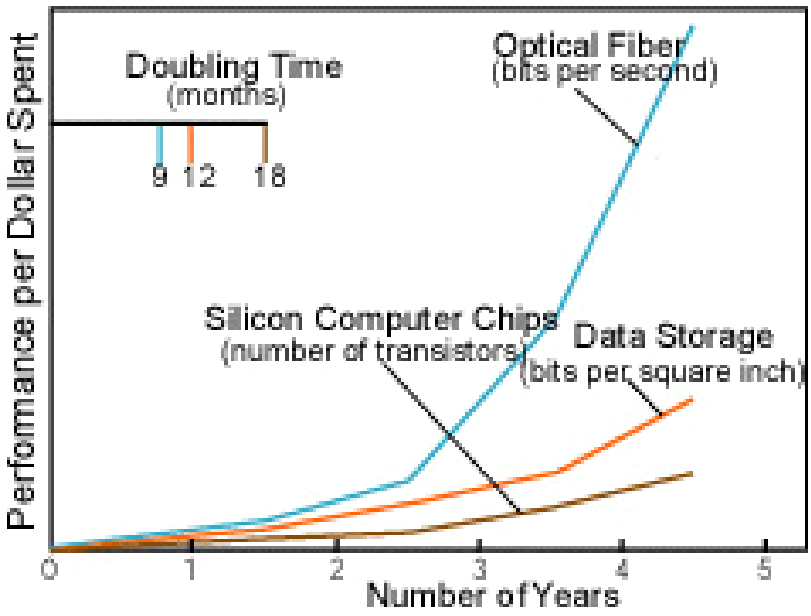


Рис.1.4.Скорость роста характеристик (синий – сеть, красный – устройства хранения инф., коричневый – компьютерные чипы).

Принципиальной проблемой в построении таких машин является скорость межпроцессорного обмена данными. Если сеть обмена работает также быстро, как внутренние шины компьютера, параллельная машина расщепляется по сети на набор специализированных устройств. Конечно, пропускная способность и скорость сетей еще недостаточны, но они очень быстро развиваются. Если число транзисторов компьютерного кристалла удваивается каждые 2 года, то скорость сети передачи данных удваивается всего за 9 месяцев.

Настоящие лекции предназначены для ознакомления с организацией «Больших ЭВМ», построенных на принципах параллелизма и предназначенных для решения «Больших задач».

Лекция 2. Формы параллелизма

2.1. Зависимость по данным

Параллелизм — это возможность одновременного выполнения более одной арифметико-логической операции. Возможность параллельного выполнения этих операций определяется правилом Рассела, которое состоит в следующем [10].

Программные объекты A и B (команды, операторы, программы) являются независимыми и могут выполняться параллельно, если выполняется следующее условие:

$$(\mathrm{InB} \wedge \mathrm{OutA}) \vee (\mathrm{InA} \wedge \mathrm{OutB}) \wedge (\mathrm{OutA} \wedge \mathrm{OutB}) = \varnothing, \qquad (2.1)$$

где In(A) — набор входных, а Out(A) — набор выходных переменных объекта A. Если условие (2.1) не выполняется, то между A и B существует зависимость и они не могут выполняться параллельно.

Если условие (2.1) нарушается в первом терме, то такая зависимость называется прямой. Приведем пример:

A: R = R1 + R2
B: Z = R + C

Здесь операторы A и B не могут выполняться одновременно, так как результат A является операндом B.

Если условие нарушено во втором терме, то такая зависимость называется обратной:

A: R = R1 + R2
B: R1 = C1 + C2

Здесь операторы A и B не могут выполняться одновременно, так как выполнение B вызывает изменение операнда в A.

Наконец, если условие не выполняется в третьем терме, то такая зависимость называется конкуренционной:

A: R = R1 + R2

B: R = C1 + C2

Здесь одновременное выполнение операторов дает неопределенный результат.

Увеличение параллелизма любой программы заключается в поиске и устранении указанных зависимостей.

Наиболее общей формой представления этих зависимостей является *информационный граф* задачи (ИГ). Пример ИГ, описывающего логику конкретной задачи, дан на рис.2.1. В своей первоначальной форме ИГ, тем не менее, не используется ни математиком, ни программистом, ни ЭВМ.

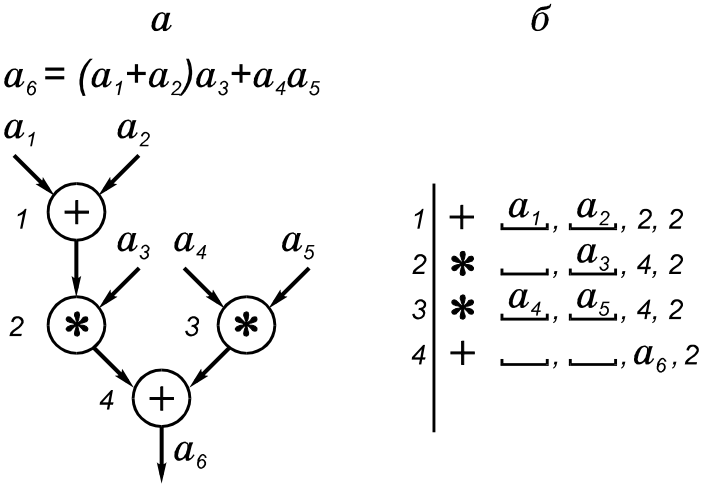


Рис.2.1. Информационный граф математического выражения (*а*) и порядок выполнения операций в выражении (*б*)

Более определенной формой представления параллелизма является *ярусно-параллельная форма* (ЯПФ): алгоритм вычислений представляется в виде ярусов, причем в нулевой ярус входят операторы (ветви), не зависящие друг от друга, в первый ярус — операторы, зависящие только от нулевого яруса, во второй — от первого яруса и т. д.

Для ЯПФ характерны параметры, в той или иной мере отражающие степень параллелизма метода вычислений: b_i — ширина *i*-го яруса; *B* — ширина графа ЯПФ (максимальная ширина яруса, т. е. максимум из b_i, $i = 1, 2, ...$); l_i — длина яруса (время операций) и *L* длина графа; ε — коэффициент заполнения ярусов; θ — коэффициент разброса указанных параметров и т. д.

Главной задачей настоящего издания является изучение связи между классами задач и классами параллельных ЭВМ. *Форма параллелизма* обычно достаточно просто характеризует некоторый класс прикладных задач и предъявляет определенные требования к структуре, необходимой для решения этого класса задач параллельной ЭВМ.

Изучение ряда алгоритмов и программ показало, что можно выделить следующие основные формы параллелизма: *естественный* или *векторный параллелизм*; *параллелизм независимых ветвей*; *параллелизм смежных операций* или *скалярный параллелизм*.

2.2. Мелкозернистый (скалярный) параллелизм

При исполнении программы регулярно встречаются ситуации, когда исходные данные для *i*-й операции вырабатываются заранее, например, при выполнении (*i* - 2)-й или (*i* - 3)-й операции. Тогда при соответствующем построении вычислительной системы можно совместить во времени выполнение *i*-й операции с выполнением (*i* - 1)-й, (*i* - 2)-й, ... операций. В таком понимании скалярный параллелизм похож на параллелизм независимых ветвей, однако они очень отличаются длиной ветвей и требуют разных вычислительных систем. Это представлено на рис.2.2.

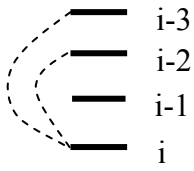


Рис.2.2. Предварительная подготовка операндов для команды i.

Рассмотрим пример. Пусть имеется программа для расчета ширины запрещенной зоны транзистора и в этой программе есть участок — определение энергии примесей по формуле

$$E = \frac{mq^4\pi^2}{8\varepsilon_0^2\varepsilon^2h^2}.$$

Тогда последовательная программа для вычисления *E* будет такой:

```
F1 = M * Q ** 4 * P ** 2
F2 = 8 * E0 ** 2 * E ** 2 * H ** 2
E = F1/F2
```

Здесь имеется параллелизм, но при записи на Фортране (показано выше) или Ассемблере у нас нет возможности явно отразить его. Явное представление параллелизма для вычисления *E* задается ЯПФ (рис. 2.3.).

Ширина параллелизма первого яруса этой ЯПФ (первый такт) сильно зависит от числа операций, включаемых в состав ЯПФ. Так, в примере для $l_1 = 4$ параллелизм первого такта равен двум, для $l_1 = 12$ параллелизм равен пяти.

Поскольку это параллелизм очень коротких ветвей и с помощью операторов FORK и JOIN описан быть не может (вся программа будет состоять в основном из этих операторов), данный вид параллелизма должен автоматически выявляться аппаратурой ЭВМ в процессе выполнения машинной программы.

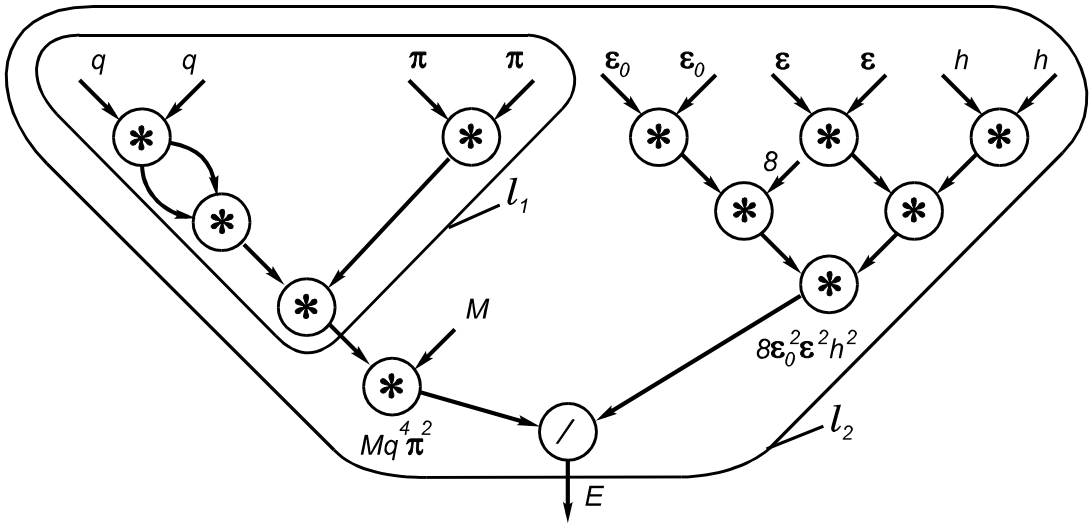


Рис.2.3. ЯПФ вычисления величины *E*

Для скалярного параллелизма часто используют термин мелкозернистый параллелизм (МЗП), в отличие от крупнозернистого параллелизма (КЗП), к которому относят векторный параллелизм и параллелизм независимых ветвей.

2.3. Крупнозернистый параллелизм

Векторный параллелизм. Наиболее распространенной в обработке структур данных является векторная операция (естественный параллелизм). *Вектор* — одномерный массив, который образуется из многомерного массива, если один из индексов не фиксирован и пробегает все значения в диапазоне его изменения. В параллельных языках этот индекс обычно обозначается знаком ∗. Пусть, например, A, B, C — двумерные массивы. Рассмотрим следующий цикл:

```
 DO 1 I = 1,N
1   C(I,J) = A(I,J) + B(I,J)
```

Нетрудно видеть, что при фиксированном J операции сложения для всех I можно выполнять параллельно, поскольку ЯПФ этого цикла имеет один ярус. По существу этот цикл соответствует сложению столбца J матриц A и B с записью результата в столбец J матрицы C. Этот цикл на параллельном яэыке записывается в виде такой векторной операции:

C (∗, j) = A (∗, j) + B (∗, j).

Возможны операции и большей размерности, чем векторные: над матрицами и многомерными массивами. Однако в параллельные ЯВУ включаются только векторные операции (сложение, умножение, сравнение и т. д.), потому что они носят универсальный характер, тогда как операции более высокого уровня специфичны.

Области применения векторных операций над массивами обширны: цифровая обработка сигналов (цифровые фильтры); механика, моделирование сплошных сред; метеорология; оптимизация; задачи движения; расчеты электрических характеристик БИС и т. д.

Рассмотрим решение линейной системы уравнений:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1,$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2,$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n.$$

Для решения таких систем уравнений при положительно определенной матрице коэффициентов используется метод простой итерации:

$$x_1^{(k+1)} = c_{11}x_1^{(k)} + c_{12}x_2^{(k)} + \ldots + c_{1n}x_n^{(k)} + d_1$$
$$x_2^{(k+1)} = c_{21}x_1^{(k)} + c_{22}x_2^{(k)} + \ldots + c_{2n}x_n^{(k)} + d_2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$x_n^{(k+1)} = c_{n1}x_1^{(k)} + c_{n2}x_2^{(k)} + \ldots + c_{nn}x_n^{(k)} + d_n.$$

Пусть *C(N,N)* — матрица коэффициентов $c_{i,j}$ системы уравнений; *D (N)* — вектор d_1, d_2, ..., d_n; *XK(N)* — вектор $x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}$ (в исходный момент хранит начальное приближение); *XK1(N)* — вектор $x_1^{(k+1)}, x_2^{(k+1)}, \ldots, x_n^{(k+1)}$; ε — заданная погрешность вычислений. Тогда программа для параллельной ЭВМ может выглядеть следующим образом:

```
DIMENSION C(N,N), D(N), XK1(N), XK(N)
XK(∗) = начальные значения
XK1(∗) = D(∗)
4 DO 1 I = 1,N
1 XK1(∗) = XK1(∗) + C(∗,I)∗XK(I)
IF (ABS(XK1(∗)-XK(∗))-ε) 2,2,3
3 XK(∗) = XK1(∗)
GO TO 4
2 Вывод XK1(∗)
STOP
```

В этой программе многократно использована параллельная обработка элементов векторов (практически во всех строках программы). Цикл по I соответствует перебору столбцов матрицы C, которые выступают в качестве векторов.

Параллелизм независимых ветвей. Суть параллелизма независимых ветвей состоит в том, что в программе решения большой задачи могут быть выделены независимые программные части — ветви. Под независимостью ветвей в первую очередь понимается независимость по данным.

В параллельных языках запуск *параллельных ветвей* осуществляется с помощью оператора FORK M1, M2 , ..., ML, где M1, M2, ..., ML — имена независимых ветвей. Каждая ветвь заканчивается оператором JOIN (R,K), выполнение которого вызывает вычитание единицы из ячейки памяти R. Так как в R предварительно записано число, равное количеству ветвей, то при последнем срабатывании оператора JOIN (все ветви выполнены) в R оказывается нуль и управление передается на оператор K. Иногда в JOIN описывается подмножество ветвей, при выполнении которого срабатывает этот оператор. В последовательных языках аналогов операторам FORK и JOIN нет и ветви выполняются друг за другом.Рассмотрим пример задачи с параллелизмом ветвей.

Пусть задана система уравнений

$$f_1(x_1, x_2, \text{K}\ , x_n) = 0,$$
$$f_2(x_1, x_2, \ldots, x_n) = 0,$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$f_n(x_1, x_2, \ldots, x_n) = 0.$$

Эту систему можно вычислять методом итераций по следующим формулам (n=3):

$$x_1^{(k+1)} = F_1(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}),$$
$$x_2^{(k+1)} = F_2(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}),$$
$$x_3^{(k+1)} = F_3(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}).$$

Функции F_1...F_3 из-за их различной программной реализации должны вычисляться отдельными программными сегментами, которые можно использовать как ветви параллельной программы. Соответствующая параллельная программа имеет вид:

```
L     FORK M1, M2, M3
M1    Z1 = F1 (X1, X2, X3)
      JOIN (R, K)
M2    Z2 = F2 (X1, X2, X3)
      JOIN (R,K)
M3    Z3 = F3 (X1, X2, X3)
      JOIN (R, K)
K     IF (ABS(Z1-X1)<ε)AND(ABS(Z2-X2)<ε)AND(ABS(Z3-X3)<ε)
          THEN вывод результатов; STOP
          ELSE X1=Z1; X2=Z2; X3=Z3; GO TO L
```

Если при выполнении оператора JOIN оказалось, что R ≠ 0, то вычисления в данной ветви останавливаются, но могут быть запущены повторно, если условие в операторе K не выполняется. Этот процесс представлен на рис.2.4.

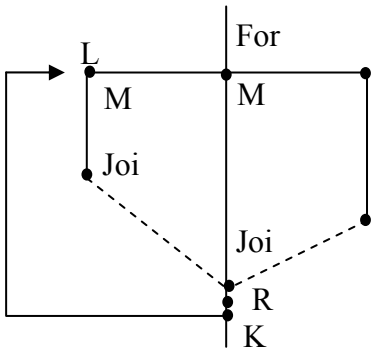


Рис.2.4. Блок-схема выполнения параллельной программы.

Для приведенного примера характерны две особенности: 1) присутствует *синхронизация процессов*, для которой используются оператор JOIN и ячейка R; состояние R = 0 свидетельствует об окончании процессов разной длительности; 2) производится *обмен данными* (обращение за X1...X3 из разных ветвей).

2.4. Параллелизм вариантов

Это широко распространенный в практие вариант решение одной и той же модели при разных входных параметрах, причем, все варианты (как правили, с длительным счетом) должны быть по-

лучены за ограниченное время. Например, варианты моделирования используются при анализе атмосферной модели климата, при расчете ядерного взрыва, обтекания летательного аппарата, расчета полупроводниковых приборов.

Параллелизм вариантов отличается от идеологии крупнозернистого параллелизма. Отличие состоит в том, что в случае крупнозернистого параллелизма вычисления проводятся внутри одной задачи и требования к скорости обмена между частями задачи достаточно высокие. В параллелизме вариантов распараллеливаются целые задачи, обмен между которыми в принципе отсутствует. Системы распределенных вычислений идеальны для решения вариантных задач.

2.5. Сетевой закон Амдала

Закон Амдала. Одной из главных характеристик параллельных систем является ускорение R параллельной системы, которое определяется выражением:

$$R = T_1 / T_n ,$$

где T_1 – время решения задачи на однопроцессорной системе, а T_n – время решения той же задачи на n – процессорной системе.

Пусть $W = W_{ск} + W_{пр}$, где W – общее число операций в задаче, $W_{пр}$ – число операций, которые можно выполнять параллельно, а $W_{ск}$ – число скалярных (нераспараллеливаемых) операций. Обозначим также через t время выполнения одной операции. Тогда получаем известный *закон Амдала* [13]:

$$R = \frac{W \cdot t}{(W_{ск} + \frac{W_{пр}}{n}) \cdot t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow[n \to \infty]{} \frac{1}{a} . \qquad (2.2)$$

Здесь $a = W_{ск} / W$ – удельный вес скалярных операций.

Закон Амдала определяет принципиально важные для параллельных вычислений положения:

- Ускорение зависит от потенциального параллелизма задачи (величина $1 - a$) и параметров аппаратуры (числа процессоров n).
- Предельное ускорение определяется свойствами задачи. Пусть, например, $a = 0{,}2$ (что является реальным значением), тогда ускорение не может превосходить 5 при любом числе процессоров, то есть максимальное ускорение определяется потенциальным параллелизмом задачи. Очевидной является чрезвычайно высокая чувствительность ускорения к изменению величины a.

Сетевой закон Амдала. Основной вариант закона Амдала не отражает потерь времени на межпроцессорный обмен сообщениями. Эти потери могут не только снизить ускорение вычислений, но и замедлить вычисления по сравнению с однопроцессорным вариантом. Поэтому необходима некоторая модернизация выражения (2.2).

Перепишем (2.2) следующим образом:

$$R_c = \frac{W \cdot t}{(W_{ск} + \frac{W_{пр}}{n}) \cdot t + W_c \cdot t_c} = \frac{1}{a + \frac{1-a}{n} + \frac{W_c \cdot t_c}{W \cdot t}} = \frac{1}{a + \frac{1-a}{n} + c} . \qquad (2.3)$$

Здесь W_c – количество передач данных, t_c – время одной передачи данных. Выражение

$$R_C = \frac{1}{a + \frac{1-a}{n} + c} \qquad (2.4)$$

и является сетевым законом Амдала. Этот закон определяет следующие две особенности многопроцессорных вычислений:

1. Коэффициент сетевой деградации вычислений *c:*

$$c = \frac{W_c \cdot t_c}{W \cdot t} = c_A \cdot c_T, \qquad (2.5)$$

определяет объем вычислений, приходящийся на одну передачу данных (по затратам времени). При этом c_A определяет алгоритмическую составляющую коэффициента деградации, обусловленную свойствами алгоритма, а c_T – техническую составляющую, которая зависит от соотношения технического быстродействия процессора и аппаратуры сети. Таким образом, для повышения скорости вычислений следует воздействовать на обе составляющие коэффициента деградации. Для многих задач и сетей коэффициенты c_A и c_T могут быть вычислены аналитически и заранее, хотя они определяются множеством факторов: алгоритмом задачи [14,15], размером данных, реализацией функций обмена библиотеки MPI, использованием разделяемой памяти и, конечно, техническими характеристиками коммуникационных сред и их протоколов.

2. Даже если задача обладает идеальным параллелизмом, сетевое ускорение определяется величиной

$$R_c = \frac{1}{\frac{1}{n} + c} = \frac{n}{1 + c \cdot n} \xrightarrow[c \to 0]{} n \qquad (2.6)$$

и увеличивается при уменьшении с. Следовательно, сетевой закон Амдала должен быть основой оптимальной разработки алгоритма и программирования задач, предназначенных для решения на многопроцессорных ЭВМ.

В некоторых случаях используется еще один параметр для измерения эффективности вычислений – коэффициент утилизации *z:*

$$z = \frac{R_c}{n} = \frac{1}{1 + c \cdot n} \xrightarrow[c \to 0]{} 1 \ . \qquad (2.7)$$

Если система имеет несколько архитектурных уровней с раными фрмами параллелизма, например, мелкозернистый параллелизм на уровне АЛУ, параллелизм среднего уровня на процессорном уровне и параллелизм задач, то *качественно* общее ускорение в системе будет:

$$R = r_1 \times r_2 \times r_3,$$

где r_i - ускорение некоторого уровня.

Лекция 3. Некоторые этапы развития параллельных технологий

Эта лекция предназначена для краткого ознакомления с основными технологиями, которые используются для организации параллельных распределенных вычислений.

Информатизация сегодня вступила в четвертый этап своего развития. Первый был связан с появлением больших компьютеров (мейнфреймов), второй — с созданием персональных компьютеров, третий — с появлением Интернета.

Четвертый этап информатизации включает ряд новых технологий. Интернет, Всемирная паутина WWW - World Wide Web (всемирная паутина или веб) и Грид – связанные между собой, но различные технологии:

- Интернет это глобальная система сетей, соединяющая множество компьютеров и локальных (сравнительно небольших) сетей и позволяющая им взаимодействовать друг с другом.
- Веб (паутина) это способ доступа к информации находящейся на удаленном, но включенном в Интернет компьютере.
- Web службы (Web Services) – это удаленные сервисные объекты, реализующие по запросу пользователя некоторую функциональность.
- Грид – способ совместного использования ресурсов, распределенных по разным, географически удаленным друг от друга, точкам планеты.

Все упомянутые выше технологии могут и используются для организации параллельных вычислений. Причем, по времени и характеру используемого оборудования эти этапы делятся на две принципиально различные части:

Цетрализованные вычисления (закон Амдала)

Распределенные вычисления (сетевой закон Амдала)

3.1. Ценрализованные (локальные) вычисления

На рис.3.1 представлены 3 класса высокопроизводительных ЭВМ. Высота треугольника определяет диапазон быстродействия, а ширина – диапазон применений.

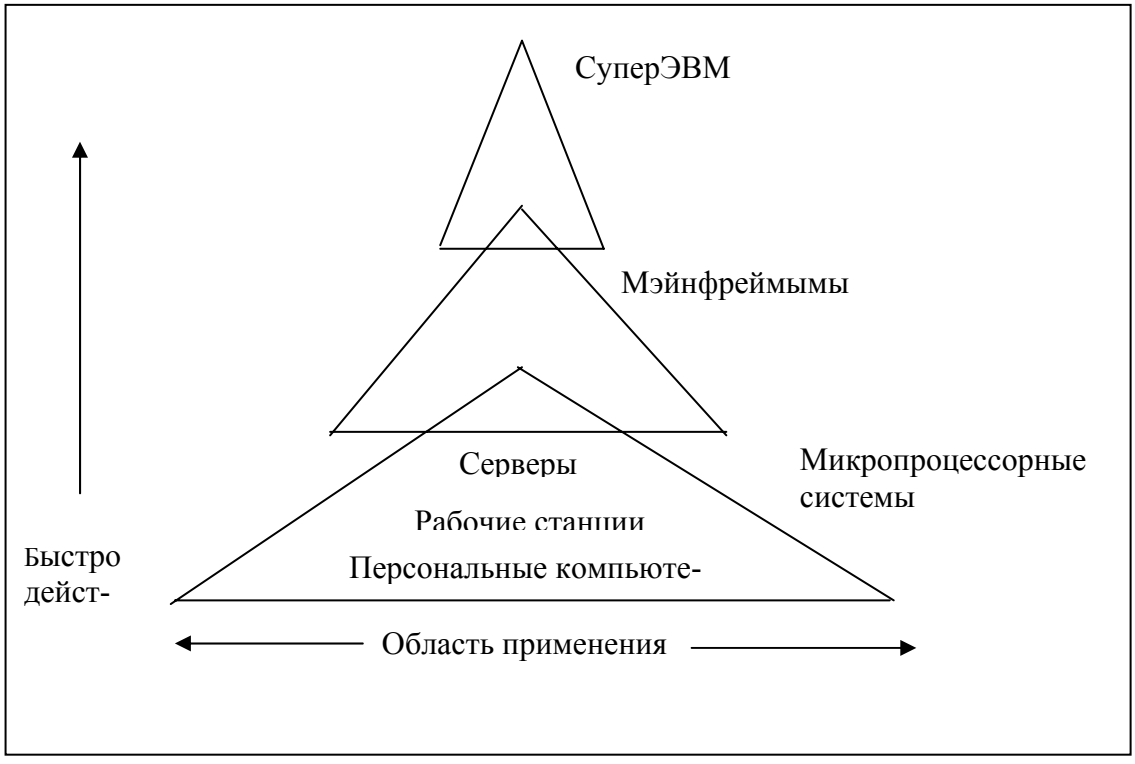


Рис.3.1 Компьютерная пирамида

Исторически первыми появились мэйнфреймы, в них отсутствовали ограничеия на объем оборудования, поэтому их называют еще большими ЭВМ. Это произошло в 1964 году, когда компания IBM выпустила семейство ЭВМ IBM – 360, которое 25 лет единолично господствовало на рынке ЭВМ. Советское семейство ЕС ЭВМ – его аналог. В этих машинах широко использовался мелкозернистый параллелизм. Мэйнфреймы выпускаются и сейчас как интегратор систем - ESA (Enterprise System Architecture). Но их потеснили персональные ЭВМ.

Персональные ЭВМ благодаря успехам микроэлектроники сейчас позиционируются как **массовые** ЭВМ систем.

СуперЭВМ – это продолжение больших ЭВМ в сторону увеличения числа АЛУ, процессоров, целых машин, объединенных в одну систему для выполнения параллельных вычислений в рамках одной задачи. Такие ЭВМ включают сотни и тысячи процессоров во множестве стоек, расположенных локально, так что эффективность вычислений определяется простым законом Амдала. В суперЭВМ наряду с мелкозернистым используется и крупнозернистый параллелизм.

Возможности централизованных вычислений ограничены, поскольку такие системы по своей природе ограничены в объеме оборудования, поэтому для больших задач больший интерес представляют распределенные системы, суммарная мощность которых по определению превосходит мощность любой централизованной системы.

Рассмотрим очень кратко средства для организации таких распределенных систем.

3.2. Интернет

Интернет есть единое информационное пространство, в котором можно строить различные сооружения - сайты, хосты, серверы и т. д.

История Интернет в некотором смысле началась с 1958 года, когда в ответ на запуск первого спутника, США создали организацию под названием ARPA (Advanced Research Projects Agency). До 68-го года внутри ARPA и в других организациях велась работа по соединению компьютеров в сеть. В 67-м году был проведен симпозиум трех независимых разработчиков компьютерных сетей: ARPA, NPL, RAND. В 68-м году была построена первая Сеть, основанная на современных принципах Интернет. Она состояла из 4-х компьютеров. В течение следующих десяти лет в Сеть ARPANET подключились множество организаций и университетов. К 1978-му году были выработаны все базовые протоколы (то есть языки общения между компьютерами), которые и сейчас используются в Интернет.

Для того чтобы компьютеры разных моделей могли общаться друг с другом в Интернет, надо было разработать единый протоколы. Кроме того, понадобилось установить способ различения одного компьютера в Сети от другого, а также - способ определения их места нахождения. Этой цели служат IP (Internet Protocol address) адреса. IP адреса это четыре числа от 0 до 255. Обычно их разделяют точками. Например: 123.23.110.1. Однако такой способ хорошо подходит только для общения самих компьютеров, но очень неудобен для человека. Для удобства же человека служат домены. Домен - это более простой и понятный способ идентифицировать компьютер(-ы) в Интернет. Например, компьютер с IP адресом 209.155.82.19 имеет доменное имя www.cdrom.com. Домены имеют несколько уровней. Например, в доменном имени www.cdrom.com, "com" - домен первого уровня, "cdrom" - второго уровня, "www" - третьего уровня. Тот, кто владеет доменом определенного уровня, может создавать сколько угодно доменов более низких уровней. Домены первого уровня не подлежат продаже. Они определяются организацией по развитию Интернет. Домены второго во многих странах продаются или же на них передается право администрирования на коммерческой основе. Это назвается делегированием домена.

Компьютер в Интернет может владеть многими протоколами. Каждый протокол обслуживается отдельной программой. У компьютера в Сети есть адрес и много портов. Когда Вы пишите в своем браузере http://www.ipform.ru фраза http:// указывает, что вы хотели бы пообщаться с компьютером www.ipform.ru, используя протокол (язык) http (HyperText Transfer Protocol).

Общепринято, что по протоколу http общаются по порту 80. И ваш браузер, и программа, которая обслуживает порт 80, умеют говорить на протоколе http. HTTP означает Hyper Text Transfer Protocol, т. е. Протокол передачи гипертекста. Гипертекст - это текст, в котором есть ссылки на другие гипертексты или места в этом тексте. При отправке почты, Ваша почтовая программа использует протокол 'SMTP', который использует порт 25, а при получении - протокол POP3 на порту 110. Есть еще огромное количество различных протоколов и портов. Их для простоты можно называть сервисами, а компьютеры, которые ожидают, что к ним придут запросы на используемых протоколах, - серверами.

3.3. WWW

WWW - есть попытка организовать всю информацию в Internet, плюс любую локальную информацию по вашему выбору как набор гипертекстовых документов. Вы перемещаетесь по сети, переходя от одного документа к другому по ссылкам. Все эти документы написаны на специально разработанном для этого языке, который называется HyperText Markup Language.

World Wide Web, Всемирная паутина, WWW, Web, Веб, - это все названия одного и того же сервиса, который был придуман в 1991 году и использует протокол HTTP для передачи гипертекстовых документов и других файлов от Веб сервера к клиентам. Главное отличие WWW от остальных инструментов для работы с Internet заключается в том, что WWW позволяет работать практически со всеми доступными сейчас на компьютере видами документов: это могут быть текстовые файлы, иллюстрации, звуковые и видео ролики, и т.д.

Принцип работы WWW следующий. Пользователь запускает у себя программу, понимающую протокол HTTP и специальный язык, на котором создается содержимое WWW. Эта программа называется «программой просмотра HTML-страницы», или по-английски - browser (браузер). При этом HTML (Hyper Text Markup Language) – это «язык разметки гипертекста», т. е. язык, с помощью которого создается гипертекст. Далее пользователь набирает адрес www сервера. Браузер обращается к серверу, с просьбой отдать документ, расположенный по этому адресу. Сервер отдает документ. Браузер получает документ, обрабатывает его и, если в нем есть картинки, также просит сервер отдать ему их, как и другие материалы документа. Этот документ принято называть страницей, а также WEB(Веб)-страницей, или HTML страницей. После этого браузер обрабатывает все пришедшие данные и показывает готовую страницу на Вашем экране. Некоторые элементы страницы (тест, картинки, кнопки) могут быть ссылками. Если Вы нажмете на них, то Ваш браузер пошлет запрос серверу, указанному в ссылке, чтобы попросить у него документ, который в ней же и обозначен. Таким образом, Вы можете передвигаться от документа к документу, от сервера к серверу, что превращает весь Интернет в одну гигантскую Сеть, связывающую документы и сервера друг с другом нитями гиперссылок

WWW - система в целом состоит из следующих компонент:

- Язык гипертекстовой разметки HTML
- Протокол передачи гипертекста HTTP
- Спецификаций на типы данных в Internet (Internet Media Types)
- Системы WWW-адресации (URL, URN, URI etc.)

Язык HTML, как уже упоминалось ранее, очень прост. С чисто с практической точки зрения HTML представляет собой разметку, сделанную обычными английскими словами внутри документа. HTML был разработан для того, чтобы выделить в документах логическую структуру.

Аббревиатура URL расшифровывается как Uniform Resource Locator, что можно вольно перевести, как "единый указатель на ресурс". Практически, это адрес документа.

3.4. Web service

Сначала определим, что такое Web служба. Web – сервис это серверный объект, реализующий некоторый элемент функциональности, с которым могут взаимодействовать удаленные программы по протоколу HTTP посредством сообщений на языке XML.

В качестве примера возможного использования Web-сервисов рассмотрим планирование путешествий. Обычно в такой ситуации требуются: заказ билетов на самолет, бронирование мест в гостинице, аренда автомобиля и, возможно, использование услуг местных компаний, организующих экскурсии.

Традиционно, используя Internet, клиениту придется посетить сервер авиакомпании, сервер гостиницы или сети гостиниц, сервер компании по аренде автомобилей и сервер компании, специализирующейся на организации экскурсий в выбранном вами месте. Все эти действия могут занять достаточно много времени, прежде чем клиент достигнете цели. Более удобно было бы запустить приложение, которое бы приняло от клиента необходимую информацию и выполнило все рутинные действия — заказ билетов, бронирование гостиницы и т.п. — автоматически, без участия клиента. Чтобы это стало возможным, следует использовать Web-сервисы. Рассмотрим, что изменится в этом случае.

Предположим, авиакомпания предоставляет Web-сервис, позволяющий приложениям получать список рейсов между двумя городами для заданной даты. В этом случае больше не требуется обращаться к Web-узлу авиакомпании и указывать различные критерии поиска — вся необходимая информация доступна в виде единого XML-документа. Теперь предположим, что авиакомпания, отель и агентство по прокату автомобилей предоставляют Web-сервисы, позволяющие программно приобретать билеты, бронировать номера и арендовать автомобили. В этом случае можно объединить вызовы всех этих сервисов в единое приложение, которое сможет выполнить всю рутинную работу без участия пользователя.

Помимо очевидного повышения качества обслуживания клиентов использование Web-сервисов имеет множество других преимуществ. Например, если агентство проката автомобилей знает, что ваш рейс задерживается, оно может более гибко распорядиться своими автомобилями.

Технология Web Services предназначена для создания распределенных приложений, функционирующих в гетерогенной среде Интернет (и всех его вариаций типа интранет и экстранет), компоненты которых взаимодействуют на базе стандартных Web - протоколов.

Архитектура сети Web Services и взаимодействие между клиентами и службами представлены на рис.3.2.

Архитектура Web-служб предполагает слабую связность между компонентами сети (loose coupling). Слабая связность означает, в частности, что компонентам системы вовсе не обязательно знать, как устроены взаимодействующие с ними подсистемы, а для взаимодействия нет необходимости в определении новых форматов данных и создании специального программного обеспечения. Принцип слабой связности далеко не нов. Считается, что именно слабая связность позволила web-технологиям в рекордно короткие сроки стать чрезвычайно популярными.

Как уже было сказано, Web Services базируется на применении открытых, утверждаемых консорциумом ИТ-сообщества стандартах и протоколах, ключевыми из которых являются следующие:

1. SOAP (Simple Object Access Protocol) — протокол доступа к простым объектам, т.е. механизм для передачи информации между уделенными объектами на базе протокола HTTP и некоторых других Интернет-протоколов;
2. WSDL (Web Services Description Language) — язык описания Web-сервисов;
3. UDDI (Universal Description, Discovery and Integration) — универсальное описание, обнаружение и интеграция — упрощенно говоря, протокол поиска ресурсов в Интернете.

Основой для реализации всех этих протоколов является язык XML (EXtensible Markup Language).

С учетом упомянутых стандартов алгоритм описания некоторого задания в системе, представленной на рис.3.2 можно описать следующим образом.

Клиент сначала обращается на UDDI в реестр ресурсов (линия 1), чтобы по заданным им признакам требуемого ресурса сведения о наличии нужного ресурса и месте его расположения (линия 2). Такой запрос мы обычно делаем в любой поисковой системе, например, Google.

Полученный из реестра ответ содержит один или несколько адресов ресурсов. Клиент по одному из полученных адресов обращается к ресурсу (линия 3), чтобы получить WSDL на интер

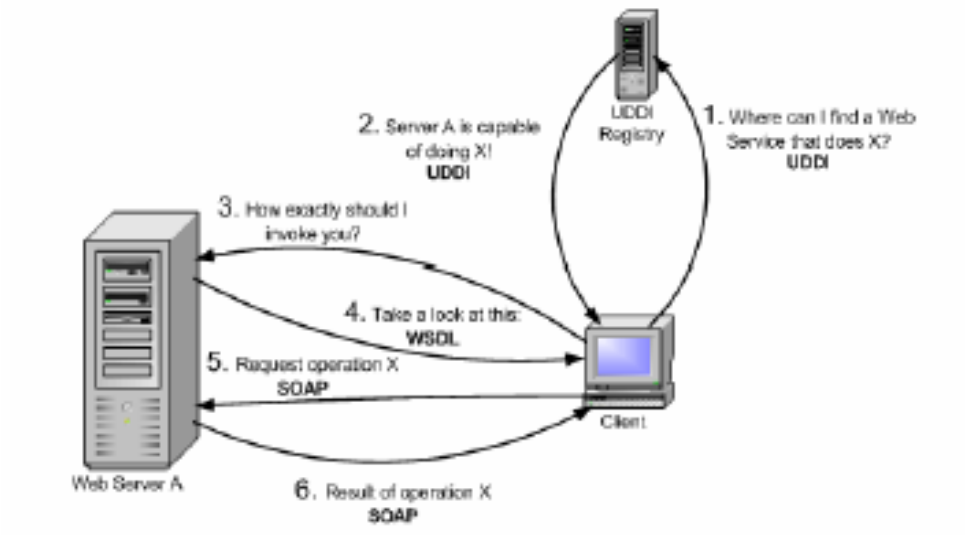


Рис.3.2.Архитектура Web Services

фейс ресурса (линия 4). Интерфейсом называется видимая пользователю часть ресурса. В интерфейсе указывается способ задания ресурсу требуемой работы. Наконец, клиент SOAP передает ресурсу задание (линия 5) и через некоторое время получает результат. В общем случае задание может выполняться не одним, а множеством ресурсов.

Язык **XML**

XML (EXtensible Markup Language) это в переводе "расширяемый язык разметки". HTML и XML создавались с различными целями:

- HTML создавался для демонстрации данных и фокусируется на том, как данные выглядят.
- XML создавался для описания данных и фокусируется на том, чем являются данные. Еще нужно написать какое-то программное обеспечение, чтобы отправить эту записку, получить ее или показать ее на экране.

В отличие от HTML, XML-теги идентифицируют данные (указывает тип данных), а не способ их отображения. Если HTML-тег указывает, например, "отобразить эти данные жирным шрифтом" (`<b>...</b>`), XML-тег действует как имя поля в вашей программе. Он ставит метку на часть данных, которые идентифицирует (например: `<message>...</message>`). Рассмотрим на пример:

```
<h1>Что XML грядущий нам готовит</h1>
<h2>Дмитрий Петров</h2>
<p>Страна: Беларусь</p>
<p>Организация: Design Studio DS</p>
<p>WWW: http://петров.virtualave.net/ds/</p>
<p>E-Mail: bcf@mail.ru</p>
<p>UIN: 35325827</p>
```

Это разметка в языке HTML. Никакой информации о структуре, только теги визуального отображения, минимум логической разметки. При использовании CSS можно несколько улучшить картину. Теги заголовков предварительно описываются, можно описать и форматирование абзацев. Но что еще лучше, различным записям можно задать уникальные стилевые идентификаторы, которыми в дальнейшем можно манипулировать. Например изменение атрибутов вывода конкретного стиля приведет соответствующим изменениям во всех документах сайта.

Посмотрите насколько дальше пошел XML – представим вышеприведенную информацию на XML.

```
<?xml version = "1.0" ?>
<editor_contacts>
  <author>
    <first_name>Дмитрий</first_name>
    <last_name>Петров</last_name>
    <article_title>Что XML грядущий нам готовит</article_title>
    <adress>
      <coutry>Беларусь</country>
      <work>Design Studio DS</work>
      <url>http://петров.virtualave.net/ds/</url>
      <email>bcf@mail.ru</email>
      <uin>35325827</uin>
    </address>
  </author>
</editor_contacts>
```

Это напоминает структуру базы данных, ине только внешним видом. XML позволяет такие манипуляции с полученными записями, как сортировка, поиск по заданным критериям. Кроме того, как вы наверняка заметили, в описаниях XML поощряется вложенность задаваемых тегов, как способ задания иерархии данных. Пользовательские теги задаются вами в подключаемой таблице стилей XSL.

Представим, что мы создали следующее приложение, которое извлекает элементы <autor>, <last name>, <adress>, <email> из XML-документа и выдает следующий результат:

```
MESSAGE
Author/Last name: Петров
Adress/email:     Петров@bsu.by
```

Это позволяет, например,автоматически создавать пофамильный список адресов элктронной почты всех авторов. Этого нельзя сделать, если информация представлена на языке HTML.

3.5. Грид

Под английским термином GRID (решетка) понимается совокупность пространственно распределенных вычислительных узлов, связанных некоторой сетью для обмена данными. В дальнейшем вместо GRID будет использоваться слово Грид.

Web Service позволяет клиенту выполнить на оборудовании владельца ресурса некоторую функцию из списка, составленного владельцем этого оборудования оборудования. Грид принципиально отличается от Web Services.

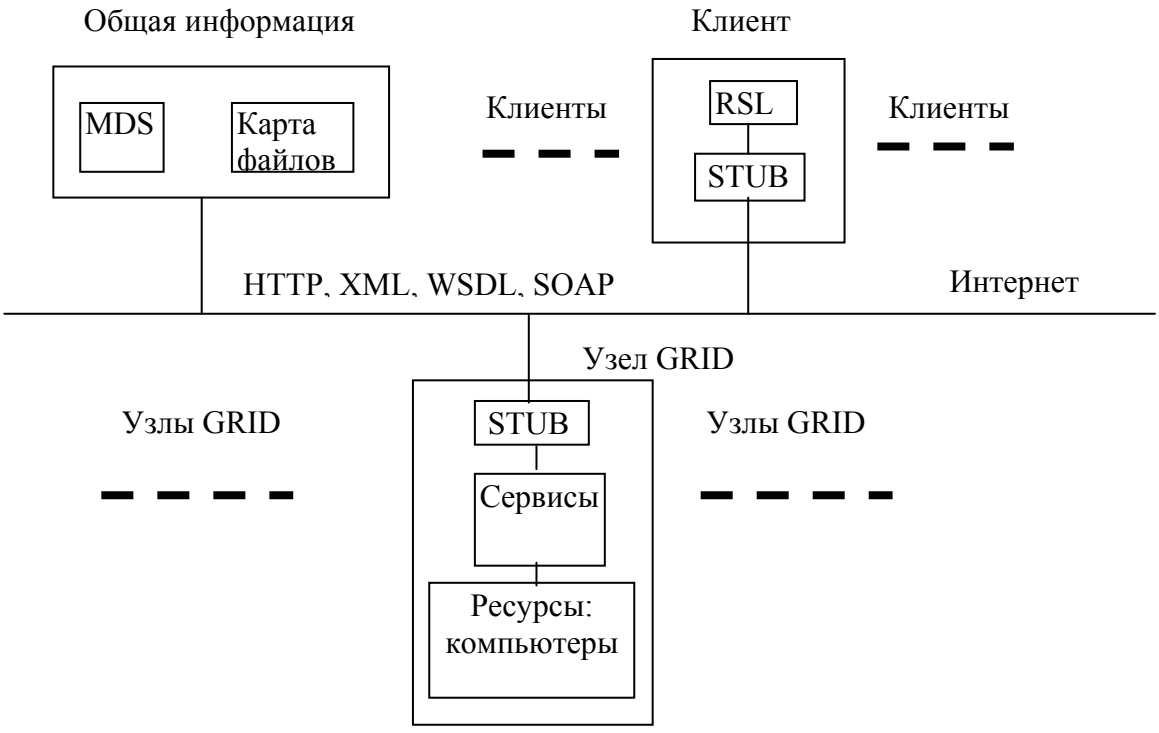
Грид - метод использования глобально процессорных мощностей и систем хранения информации (дисковые системы большой емкости) на основе повременной аренды без их физического перемещения в пространстве.

Естественно, Грид включает как ресурсы и все, наработанное в WebServices. . Более того, протоколы WebServices (WSDL, SOAP, UDDI), средства адресации в расширенном варианте являются основными протоколами GRID. Грид характеризуется следующим образом:

- География ресурсов глобальная
- Ресурсы неоднородные, отличаются по аппаратуре, операционным системам, протоколам функционирования и связи
- Состав ресурсов переменный, в общем случае непредсказуемый
- Элементы среды могут быть враждебными, то есть могут повредить или захватить закрытые данные

Существуют следующие две основные разновидности GRID:
- Коммунальный GRID. Такой GRID позволяет предоставлять по запросу клиента соответствующий ресурс.
- Метакомпьютер. В этом случае объединяются ресурсы ряда корпораций, в пределе – весь Интернет для решения общечеловеческих задач.

Структура и функционирование Грид задается рисунком ниже. Здесь клиент сначала обращается в реестр ресурсов MDS, чтобы получить по заданным им признакам требуемого ресурса сведения о наличии нужного ресурса и месте его расположения. Клиент по одному из полученных адресов обращается к ресурсу, чтобы получить интерфейс ресурса. В интерфейсе указывается способ задания ресурсу требуемой работы. Наконец, клиент передает ресурсу задание и через некоторое время получает результат.



Здесь RSL (Resource Specification Language) – язык для описания задания, которое нужно выполнить: программа, данные, где все это размещено, куда послать результат, требования к ресурсам и др.).

Лекция 4. Методы увеличения параллелизма

4.1. Метод планирования трасс

Базовые блоки невелики по размеру (5 – 20 команд) и даже при оптимальном планировании параллелизм не может быть большим. Экспериментально установлено, что ускорение зависит от размера распараллеливаемого блока следующим образом:

$$h = a + b\,w,$$

где a и b — константы ($a \approx 1$, $b \approx 0{,}15$), h - средняя ширина параллелизма (число параллельных ветвей), w – число команд в программе. Следовательно, можно сказать, что

$$t_{пар} = t_{посл} / h = \frac{t_{посл}}{a + b \cdot w} = \frac{t_{посл}}{1 + 0.15 \cdot w}$$

Здесь: $t_{пар}$ *и* $t_{посл}$ – времена параллельного и последовательного исполнения одного и того же отрезка программы.

Таким образом, основной путь увеличения скалярного параллелизма программы – это удлинение ББ, а развертка – наиболее простой способ для этого. Цикл

```
      DO 1 I=1,N
      C(I) = A(I) + B(I)
    1 CONTINUE
```

имеет небольшую длину ББ, но ее можно увеличить путем развертки приведенного цикла на две, четыре и так далее итераций, как показано ниже

```
      DO 1 I=1,N,2
      C(I) = A(I) + B(I)
      C(I+1) = A(I+1) + B(I+1)
    1 CONTINUE
      DO 1 I=1,N,4
      C(I) = A(I) + B(I)
      C(I+1) = A(I+2) + B(I+1)
      C(I+1) = A(I+2) + B(I+2)
      C(I+1) = A(I+3) + B(I+3)
    1 CONTINUE
```

К сожалению, развертка возможна только, если:

- Все итерации можно выполнять параллельно
- В теле цикла нет условных переходов

Но в большинстве случаев тело цикла содержит операторы переходов. Достаточно универсальный метод планирования трасс предложил в 80-е годы J.Fisher. Рассмотрим этот метод на примере рис.4.1, на котором представлена блок-схема тела цикла. На схеме в кружках представлены номера вершин, а рядом – вес вершины (время ее исполнения); на выходах операторов переходов проставлены вероятности этих переходов.

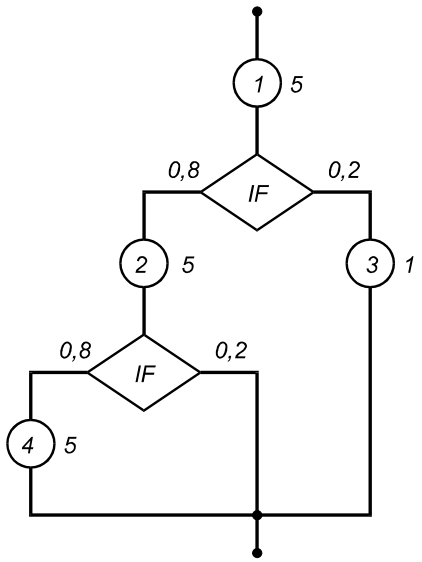


Рис.4.1. Пример выбора трасс

Возможны три варианта путей исполнения тела цикла:

- Путь 1-2-4 обладает объемом вычислений (5+5+5) и вероятностью 0.8*0.8=0.64
- Путь 1-2 имеет объем вычислений 5+5 и вероятность 0.8*0.2=0.16
- Путь 1-3 имеет объем вычислений 5+1 и вероятность 0.2

Примем путь 1-2-4 в качестве главной трассы. Остальные пути будем считать простыми трассами. Ограничимся рассмотрением метода планирования трасс только по отношению к главной трассе.

Если метод планирования трасс не применяется, то главная трасса состоит из трех независимых блоков. Суммарное время выполнения этих блоков будет в соответствии с вышеприведенными формулами равным:

$$T_1 = 0.64 \cdot \left(\frac{5}{1+0.15 \cdot 5} + \frac{5}{1+0.15 \cdot 5} + \frac{5}{1+0.15 \cdot 5} \right) = 5.5$$

В методе планирования трасс предлагается считать главную трассу единым ББ, который выполняется с вероятностью 0,64. Если переходов из данной трассы в другие трассы нет, то объединенный ББ выполняется за время

$$T_2 = 0{,}64 \frac{3 \cdot 5}{1+0{,}15 \cdot 3 \cdot 5} = 3$$

Таким образом, выигрыш во времени выполнения главной трассы составил *T1/T2* = 1.8 раз. В общем случае при объединении *k* блоков с равным временем исполнения *w* получаем:

$$\frac{T_1}{T_2} = \left(\frac{k \cdot w}{a + b \cdot w}\right) / \left(\frac{k \cdot w}{a + b \cdot k \cdot w}\right) = \frac{a + b \cdot kw}{a + b \cdot w} \xrightarrow[w \to \infty]{} k$$

При построении ЯПФ объединенного ББ и дальнейшем планировании команды могут перемещаться из одного исходного ББ в другой, оказываясь выше или ниже оператора перехода, что может привести к нарушению логики выполнения программы.

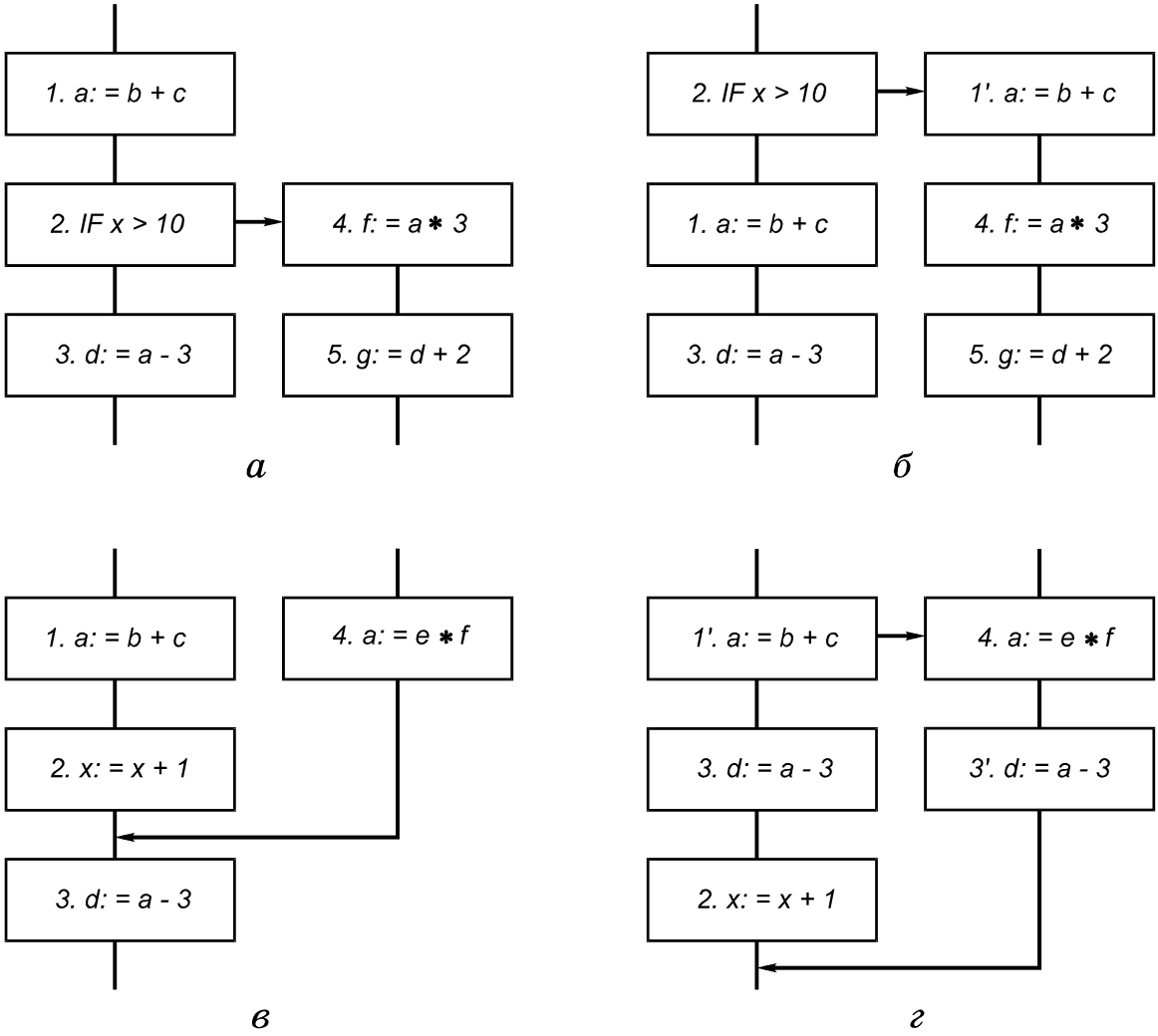


Рис.4.2. Способы введения компенсационных кодов

Чтобы исключить возможность неправильных вычислений, вводятся компенсационные коды. Рассмотрим примеры (рис.4.2). Пусть текущая трасса (рис.4.2,а) состоит из операций 1, 2, 3. Предположим, что операция 1 не является срочной и перемещается поэтому ниже условного перехода 2. Но тогда операция 4 читает неверное значение *a*. Чтобы этого не произошло, компилятор водит компенсирующую операцию 1 (рис.4.2,б).

Пусть теперь операция 3 перемещается выше IF. Тогда операция 5 считает неверное значение *d*. Если бы значение *d* не использовалось на расположенном вне трассы крае перехода, то перемещение операции 3 выше IF было бы допустимым.

Рассмотрим переходы в трассу извне. Пусть текущая трасса содержит операции 1, 2, 3 (рис.4.2,в). Предположим, что компилятор перемещает операцию 3 в положение между операциями 1 и 2. Тогда в операции 3 будет использовано неверное значение *a*. Во избежание этого, необходимо ввести компенсирующий код 3 (рис.4.2,г).

Порядок планирования трасс для получения конечного результата таков:

1. Выбор очередной трассы и ее планирование.
2. Коррекция межтрассовых связей по результатам упаковки очередной трассы. Компенсационные коды увеличивают размер машинной программы, но не увеличивают числа выполняемых в процессе вычислений операций.
3. Если все трассы исчерпаны или оставшиеся трассы имеют очень низкую вероятность исполне-

ния, то компиляция программы считается законченной, в противном случае осуществляется переход на пункт 1.

4.2. Метод гиперплоскостей.

Часто параллелизм не очевиден, но он есть. Наиболее распространенными методами выявления такого параллелизма является метод гиперплоскостей. Объектами векторизации в этих методах являются циклы типа *DO* в Фортране. Необходимое условие параллельного выполнения i-й и j-й итераций цикла записывается в виде правила Рассела для циклов:

(OUT(i) AND IN(j)) OR (IN(i) AND OUT(j)) OR (OUT(i) AND OUT(j))=0

Здесь *IN(i)* и *OUT(i)*—множества используемых и генерируемых переменных i-й итерации соответственно, где *i*—целочисленный вектор; *IN(i)* и *OUT(i)* называют также множеством входных и выходных переменных i-й итерации. Два первых дизъюнктивных члена учитывают прямую или обратную зависимость по данным между итерациями, т. е. параллельное выполнение может быть невозможным, если одна и та же переменная используется в теле цикла как входная и как выходная. Третий член учитывает конкуренционную зависимость, которая появляется, если определенная переменная применяется в теле цикла в качестве выходной больше одного раза.

Приведем примеры зависимостей между итерациями – прямая (а), обратная (б) и конкуренционная (в):

а) Итерация i a (i) = a (i –1) итерация 5 a(5) = a(4)

Итерация i+1 a (i + 1) = a (i) итерация 6 a(6) = a(5)

б) Итерация i a (i – 1) = a (i) итерация 5 a(4) = a(5)

Итерация i + 1 a (i) = a (i = 1) итерация 6 a(5) = a(6)

в) Итерация i s =

Итерация i + 1 s =

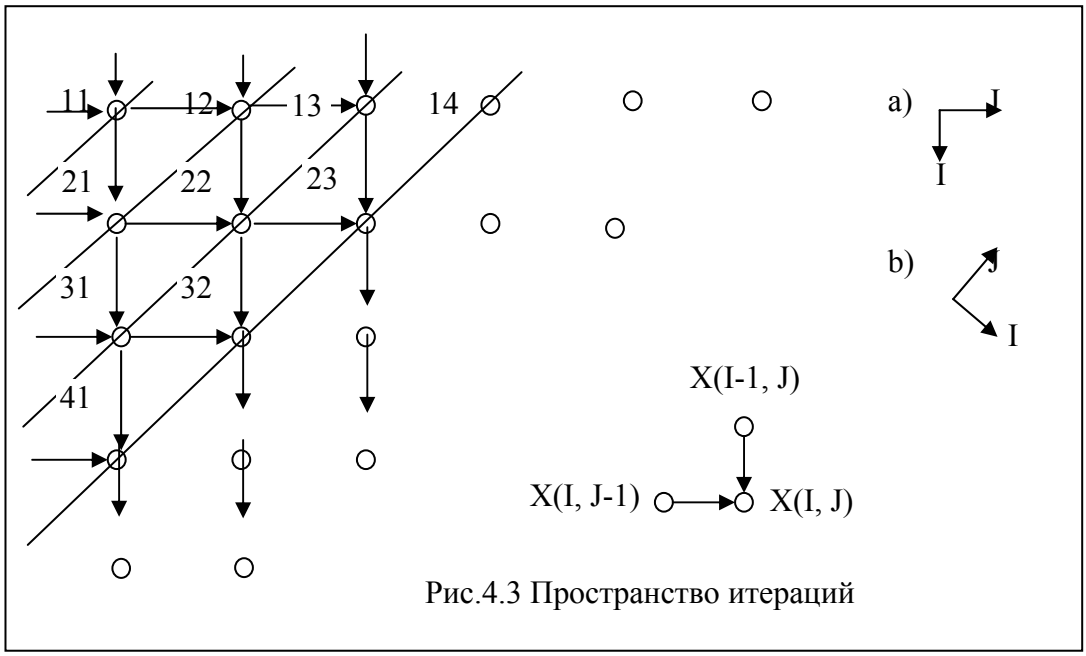


Рис.4.3 Пространство итераций

Рассмотрим метод гиперплоскостей, предложенный Lamport (1974). Метод носит название “фронта волны”. Пусть дана программа для вычисления в цикле значения $X_{i,j}$ как среднего двух смежных точек (слева и сверху):

```
      DO 1  I = 1,N
      DO 1  I =  1,N
    1 X (I, J) = X (I-1, J) + Y (I, J –1) + C
```

Рассмотрим две любые смежные по значениям индексов итерации, например:

```
X(2,2) = X(1,2) + X(2,1)
X(2,3) = X(1,3) + X(2,2)
```

Очевидно, что использовать для сложения смежные строки нельзя, так как нижняя строка зависит от верхней. Нельзя складывать и смежные столбцы, так как правый столбец зависит от левого. Тем не менее параллелизм в задаче есть, например, все операции в диагонали 41, 32, 23, 14 можно выполнять параллельно. Если повернуть оси I, J на 45 градусов и переименовать операции внутри каждой диагонали (рис.4.4), то можно использовать этот параллелизм.

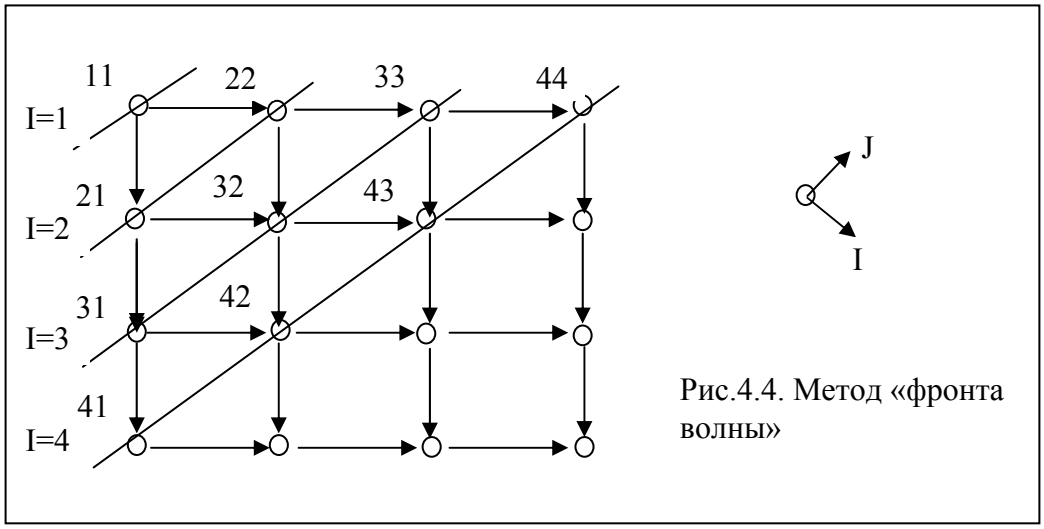


Рис.4.4. Метод «фронта волны»

Соответствующая программа для верхних диагоналей (включая главную) и нижних диагоналей приведена ниже:

```
    DO 1 I = 1,N                    Пусть  =3, тогда   x(3,1)=x(2,1)+x(3,0)
    DO PAR J = 1, I                                    x(3,2)=x(2,2)+x(2,1)
    X (I, J) = X(I-1, J) + X(I-1, J-1)                 x(3,3)=x(2,3)+x(2,2)
1   CONTINUE                    Эти итерации действительно независимы
  ---------------------------------------
    K + 2
    DO 2 I = N +1, 2N – 1
    DO PAR J + K, N
    R = K + 1
    X(I, J) = x(I-1, J) + X(I-1, J-1) + C
2   CONTINUE
```

Алгоритм метода гиперплоскостей состоит в следующем:

1. Производится анализ индексов и построение зависимостей в пространстве итераций
2. Определяется угол наклона осей и переименование переменных
3. Строится параллельная программ

Лекция 5. Классы систем с мелкозернистым параллелизмом по Фишеру.

5.1.Классы систем с мелкозернистым параллелизмом по Фишеру

Мелкозернистый параллелизм – это параллелизм смежных команд или операторов (Fine Grain). Чтобы реализовать этот вид параллелизма, компилятор и аппаратура должны выполнить следующие действия: определить зависимости между операциями; определить операции, которые не зависят ни от каких еще не завершенных операций; спланировать время и место выполнения операций.

В соответствиис этим архитектуры с МЗП классифицируются следующим образом (таб. 5.1):

1. *Последовательностные архитектуры*: архитектуры, в которых компилятор не помещает в программу в явном виде какую-либо информацию о параллелизме. Компилятор только перестраивает код для упрощения действий аппаратуры по планированию. Представителями этого класса являются суперскалярные процессоры.
2. *Архитектуры с указанием зависимостей*: в программе компилятором явно представлены зависимости, которые существуют между операциями, независимость между операциями определяет аппаратура. Этот класс представляют процессоры потока данных.
3. *Независимостные архитектуры*: архитектуры, в которых компилятор помещает в программу всю информацию о независимости операций друг от друга. Это VLIW-архитектуры.

Таб. 5.1. Классы микропроцессоров по Фишеру

Тип архитектуры	Кто определяет зависимость по данным	Кто определяет независимость по данным	Кто планирует время и место выполнения	Представители
Последовательностные	Аппаратура	Аппаратура	Аппаратура	Сперскалярный Pentium
Зависимостные	компилятор	Аппаратура	Аппаратура	Потоковый Pentium Pro
Независимостные	Компилятор	Компилятор	Компилятор	VLIW процессоры

5.2. Суперскалярный Pentium

В таких архитектурах компилятор не производит выявление параллелизма, поэтому программы для последовательностных архитектур не содержат явно выраженной информации об имеющемся параллелизме, и зависимости между командами должны быть определены аппаратурой. В некоторых случаях компилятор может для облегчения работы аппаратуры производить упорядочивание команд. Если команда не зависит от всех других команд, она может быть запущена. Возможны два варианта последовательной архитектуры: суперконвейер и суперскалярная организация. В первом случае параллелизм на уровне команд нужен для того, чтобы исключить остановки конвейера. Здесь аппаратура должна определить зависимости между командами и принять решение, когда запускать очередную команду. Если конвейер способен вырабатывать один результат в каждом такте, такой конвейер называется *суперконвейером*. Его эффективность зависит от величины параллелизма в программе.

Процессор с несколькими конвейерами процессор называется *суперскалярным*. Здесь главная проблема - определить зависимости между командами, которые следует запустить одновременно. В качестве примера приведем схему микропроцессора Pentium (рис.5.1).

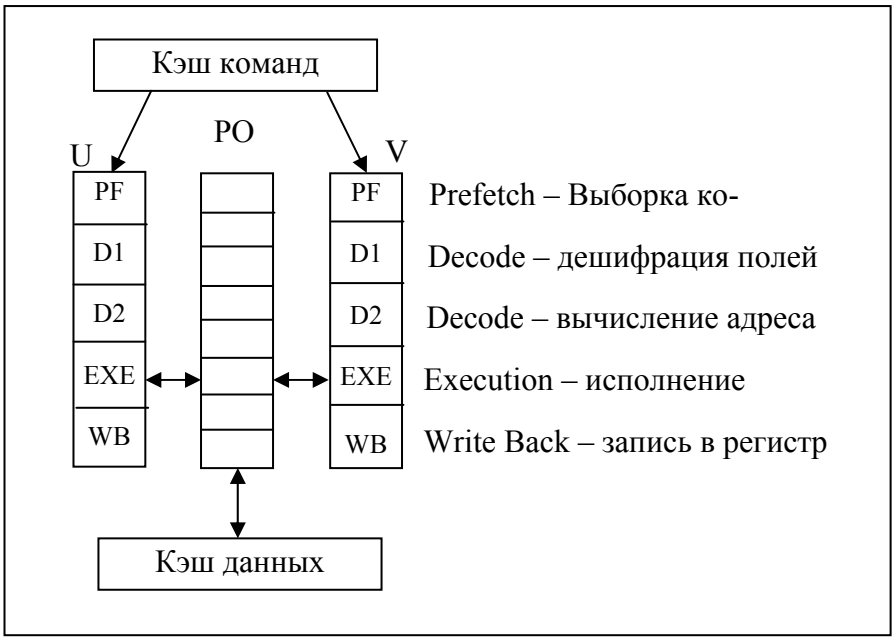


Рис. 5.1. Схема микропроцессора Pentium

Здесь имеется два целочисленных конвейера U и V, которые образуют суперскалярную архитектуру. Рассмотрим возможости суперскалярного микропроцессора. Пусть имеется отрезок программы на ЯВУ и соответствующий ему ассемблерный вариант на языке ассемблера МП *i*80x86:

```
    DO 1 I = 1, 10          M:  mov edx, [eax + a]
    A (I) = A (I) + 1           mov ecx, [eax + b]
1   B (I) = B (I) + 1           inc edx
                                inc ecx
                                mov [eax + a], edx
                                mov [eax + b], ecx
                                add eax
                                jnz M
```

В этой программе команды расположены последовательно, никаких отметок о параллелизме команд нет, но на этапе компиляции команды переставлены так, что параллельные команды в программе являются смежными.

В результате потактного распараллеливания в конвейеры U и V будет загружена и исполнена следующая программа:

```
        Конвейер U              Конвейер V
M:  mov edx, [eax + a]      mov ecx, [eax + b]
    inc edx                 inc ecx
    mov [eax + a], edx      mov [eax + b], ecx
    add eax, 4              jnz M
```

которая требует для своего исполнения 4 такта вместо 7 в исходном тексте.

Суперскалярные процессоры хороши только для 2-ух конвейеров.

5.3. VLIW архитектуры

Эта архитектура будет рассмотрена на примере процессора C6 (Texas Instruments). Особенностью процессора является наличие двух практически идентичных вычислительных блоков, которые могут работать параллельно и обмениваться данными. Каждый блок содержит четыре функциональных устройства, которые также могут работать параллельно.

На рис.5.2 представлена структура процессора. Блоки выборки программ, диспетчирования и декодирования команд могут каждый такт доставлять функциональным блокам до восьми 32-разрядных команд. Обработка команд производится в в путях A и B, каждый из которых содержит 4 функциональных устройства (L, S, M, и D) и 16 32-разрядных регистра общего назначения. Каждое ФУ одного пути почти идентично соответствующему ФУ другого пути.

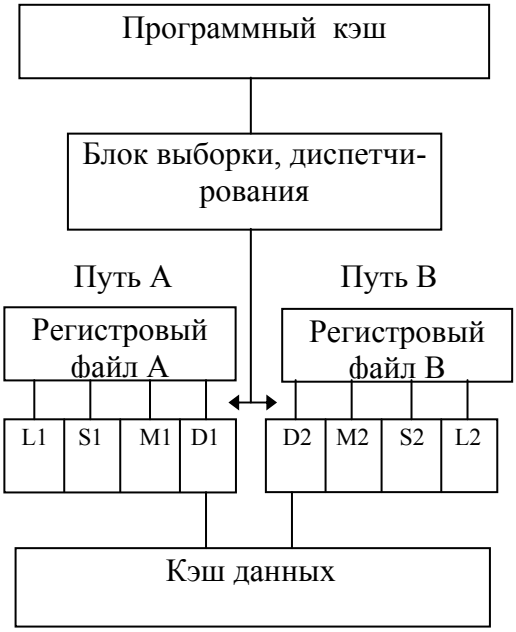


Рис.5..2. Структура микропроцессора C6

Каждое ФУ прямо обращается к регистровому файлу внутри своего пути, однако существуют дополнительные связи, позволяющие ФУ одного пути получать доступ к данным другого пути. Функциональные устройства описаны в таблице:

Функциональное устройство	Операции с фиксированной точкой
L	32/40-разрядные арифметические, логические и операции сравнения
S	32-разрядные арифметические, логические операции, сдвиги, ветвления,
M	16-разрядное умножение
D	32-разрядное сложение, вычитание, вычисление адресов, операции обращения к памяти

За один такт из памяти всегда извлекается 8 команд. Они составляют пакет выборки длиною 256 разрядов. В нем содержится 8 32-разрядных команд.

Выполнение отдельной команды частично управляется *p*-разрядом, расположенным в этой команде. *p* – разряды сканируюся слева направо (к более старшим адресам команд в пакете). Если в команде *i* *p*-разряд равен 1, то *i*+1 команда может выполняться параллельно с *i*-ой. В противном случае команда *i*+1 должна выполняться в следующем цикле, то есть после цикла, в котором выполнялась команда *i*. Все команды, которые будут выполниться в одном цикле, образуют исполнительный пакет. *Пакет выборки* и *исполнительный пакет* это различные объекты.

Исполнительный пакет может содержать до 8 команд. Каждая команда этого пакета должна использовать отдельное функциональное устройство. Исполнительный пакет не может пересекать границу 8 слов. Следовательно, последний *p*-разряд в пакете всегда устанавливается в 0 и каждый пакет выборки запускает новый исполнительный пакет. Имеется три типа пакетов выборки: полностью последовательные, полностью параллельные, и смешанные. В полностью последовательных пакетах *p*-разряды всех команд установлены в 0, следовательно, все команды выполняются строго последовательно. В полностью параллельных пакетах *p*-разряды всех команд установлены в 1 и все команды выполняются параллельно. Пример смешанного пакета выборки из восьми 32-разрядных команд приведен на рис.5.3.

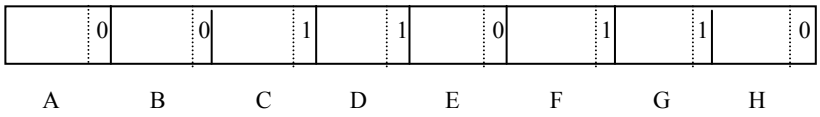


Рис.5.3. Частично последовательный пакет выборки

Состав исполнительных пакетов для этой команды представлен ниже:

Исполнителные пакеты (такты)	Команды		
1	A		
2	B		
3	C	D	E
4	F	G	H

Лекция 6. Технология MMX

Рассматриваемая в этой лекции технология MMX использует векторный, то есть крупноформатный параллелизм. Однако, эта технология реализована в виде одиночного процессора с собственной системой команд. В ней не используется коммутатор, не решаются сложные вопросы пространственного размещения и доступа к данным. Все это не позволяет по архитектуре отнести MMX процессоры к полноценным векторным процессорам и приближает их к процессорам с мелкозернистым параллелизмом.

В 1997 году компания Intel выпустила первый процессор с архитектурой SIMD с названием Pentium MMX (MultiMedia eXtension). При анализе участков с большим объемом вычислений выяснилось, что все приложения имеют следующие общие свойства, определившие выбор системы команд и структуры данных:

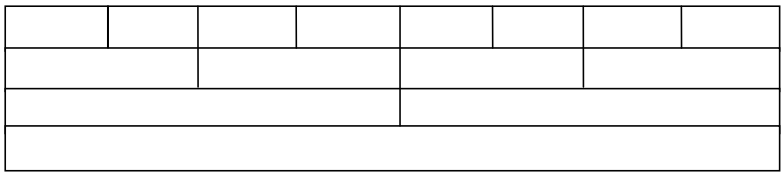
- небольшая разрядность целочисленных данных (например, 8-разрядные пикселы для графики или 16-разрядное представление речевых сигналов);
- небольшая длина циклов, но большое число их повторений;
- большой объем вычислений и значительный удельный вес операций умножения и накопления;
- существенный параллелизм операций в программах.

Это и определило новую структуру данных и расширение системы команд. Технологии, аналогичные MMX, затем были повторены в микропоцессорах многих ведущих микроэлектронных компаний.

Технология MMX выполняет обработку параллельных данных по методу SIMD. Для этого используются четыре новых типа данных, 57 новых команд и восемь 64-разрядных MMX-регистров.

Метод SIMD позволяет по одной команде обрабатывать несколько пар операндов (векторная обработка). Основой новых типов данных является 64-разрядный формат, в котором размещаются следующие четыре типа операндов:

- Упакованный байт (Packed Byte) – восемь байтов, размещенных в одном 64-разрядном формате;
- Упакованное слово (Packed Word) – четыре 16-разрядных слова в 64-разрядном формате;
- Упакованное двойное слово (Packed Doubleword) – два 32-разрядных двойных слова в 64-разрядном формате;
- Учетверенное слово (Quadword) – 64-разрядная единица данных.



B (Byte)
W (Word)
DW (Double Word)
Q (Quad Word)

Совместимость MMX с операционными системами и приложениями обеспечивается благодаря тому, что в качестве регистров MMX используются 8 регистров блока плавающей точки архитектуры Intel. Это означает, что операционная система использует стандартные механизмы плавающей точки также и для сохранения и восстановления MMX кода. Доступ к регистрам осуществляется по именам MM0-MM7.

MMX команды обеспечивают выполнение двух новых принципов: операции над упакованными данными и арифметику насыщения.

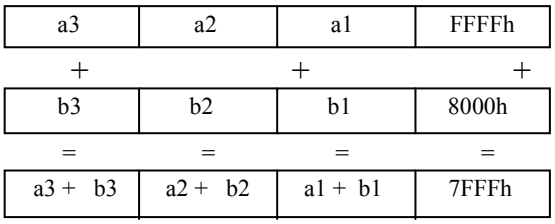
Арифметику насыщения легче всего определить, сравнивая с режимом циклического возврата. В режиме циклического возврата результат в случае переполнения или потери значимости обрезается. Таким образом перенос игнорируется. В режиме насыщения, результат при переполнении или потери значимости, ограничивается. Результат, который превышает максимальное значение типа данных, обрезается до максимального значения типа. В случае же, если результат меньше минимального значения, то он обрезается до минимума. Это является полезным при проведении многих операций, например, операций с цветом. Когда результат превышает предел для знаковых чисел, он округляется до 0x7F (0xFF для беззнаковых). Если значение меньше нижнего предела, оно округляется до 0x80 для знаковых чисел(0x00 для беззнаковых). Для примера с цветоделением это означает, что цвет останется чисто черным или чисто белым и удается избежать негативной инверсии

Ниже приводится таблица команд, распределенных по категориям. Если команда оперирует несколькими типами данных – байтами (B), словами (W), двойными словами (DW) или учетверенными словами (QW), то конкретный тип данных указан в скобках. Например, базовая мнемоника PADD (упакованное сложение) имеет следующие вариации: PADDB, PADDW и PADDD.

Все команды MMX оперируют над двумя операндами: операнд-источник и операнд-приемник. В обозначених команды правый операнд является источником, а левый – приемником. Операнд-приемник может также быть и вторым операндом-источником. Команды записывает в приемник результата.

Операнд-источник для всех команд MMX (кроме команд пересылок) может располагаться либо в памяти либо в регистре MMX. Операнд-приемник будет располагаться в регистре MMX. Для команд пересылок операндами могут также выступать целочисленные РОН или ячейки памяти. Все 57 MMX-команд разделенны на следующие классы: арифметические, сравнения, преобразования, логические, сдвига, пересылки и смены состояний. Основные форматы представлены в таблице 1 (не представлены команды сравнения, логические, сдвига и смены состояний). Приведем несколько примеров, иллюстрирующих выполнение команд MMX. В этих примерах используются 16-разрядные данные, хотя есть и модификации для 8- или 32-разрядных операндов.

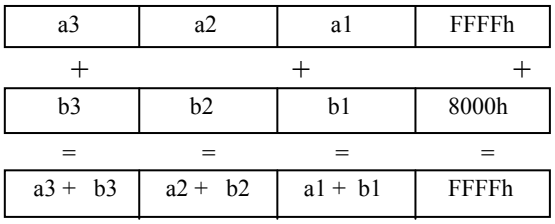
Пример 1. Пример представляет упакованное сложение слов без переноса. По этой команде выполняется одновременное и независимое сложение четырех пар 16-разрядных операндов. На рисунке самый правый результат превышает значение, которое можно представить в 16-разрядном формате. FFFFh + 8000h дали бы 17-разрядный результат, но 17-ый разряд теряется, поэтому результат будет 7FFFh.



Paddw (Packed Add Word)

Пример 2. Этотпример иллюстрирует сложение слов с беззнаковым насыщением. Самая правая операция дает результат, который не укладывается в 16 разрядов, поэтому имеет место насыщение. Это означает, что если сложение дает в результате переполнение или вычитание выражается в исчезновении данных, то в качестве результата используется наибольшее или наименьшее значение, допускаемое 16-разрядным форматом. Для беззнакового представления наибольшее и наименьшее значение соответственно будут FFFFh и 0x0000, а для знакового представления — 7FFFh и 0x8000. Это важно для обработки пикселов, где потеря переноса заставляла бы черный пиксел внезапно превращаться в белый, например, при выполнении цикла заливки по методу Гуро в 3D-графике. Особой здесь является команда PADDUS [W] – упакованное сложение слов беззнаковое с насыщением. Число FFFFh, обрабатываемое как беззнаковое (десятичное значение 65535),

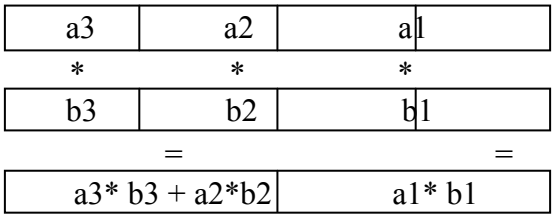
добавляется к 0x0000 беззнаковому (десятичное 32768) и результат насыщения до FFFFh – наибольшего представимого в 16-разрядной сетке числа.



Paddus [W] (Add Unsigne Saturation)

Для PADDUS не имеется никакого “разряда режима насыщения”, так как новый разряд режима потребовал бы изменения операционной системы.

Пример 3. Этот пример представляет ключевую команду умножения с накоплением, которая является базовой для многих алгоритмов цифровой обработки сигналов: суммирования парных произведений, умножения матриц, быстрого преобразования Фурье и т.д. Эта команда – упакованное умножение со сложением PMADD.



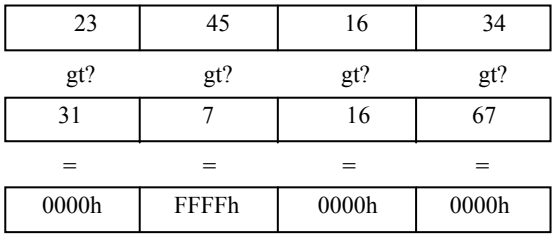
Pmadd (Packed Multiply Add)

$$\sum_{1}^{2} a_i \cdot b_i + \sum_{3}^{4} a_i \cdot b_i$$

Команда PMADD использует в качестве операндов 16-разрядные числа и вырабатывает 32-разрядный результат. Производится умножение четырех пар чисел и получаются четыре 32-разрядных результата, которые затем попарно складываются, вырабатывая два 32-разрядных числа. Этим и оканчивается выполнение PMADD.

Чтобы завершить операцию умножения с накоплением, результаты PMADD следует прибавить к данным в регистре, который используется в качестве аккумулятора. Для этой команды не используется никаких новых флагов условий, кроме того, она не воздействует ни на какие флаги, имеющиеся в архитектуре процессоров Intel.

Пример 4. Следующий пример иллюстрирует операцию параллельного сравнения. Сравниваются четыре пары 16-разрядных слов, и для каждой пары вырабатывается признак “истина” (FFFFh) или “ложь” (0000h).

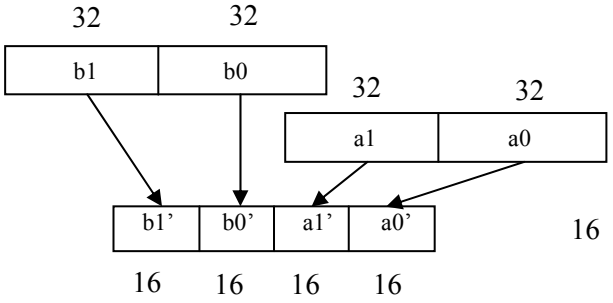


Greate Then

Результат сравнения может быть использован как маска, чтобы выбрать элементы из входного набора данных при помощи логических операций, что исключает необходимость использования ветвления или ряда команд ветвления. Способность совершать условные пересылки вместо использования команд ветвления является важным усовершенствованием в перспективных процессорах, которые имеют длинные конвейеры и используют прогнозирование ветвлений. Ветвление,

основанное на результате сравнения входных данных, обычно трудно предсказать, так как входные данные во многих случаях изменяются случайным образом. Поэтому исключение операций ветвления совместно с параллелизмом MMX-команд является существенной особенностью технологии MMX.

Пример 5. Пример иллюстрирует работу *команды упаковки.* Она принимает четыре 32-разрядных значения и упаковывает их в четыре 16-разрядных значения, выполняя операцию насыщения, если какое-либо 32-разрядное входное значение не укладывается в 16-разрядный формат результата. Имеются также команды, которые выполняют противоположное действие — распаковку, например, то есть преобразуют упакованные байты в упакованные слова.



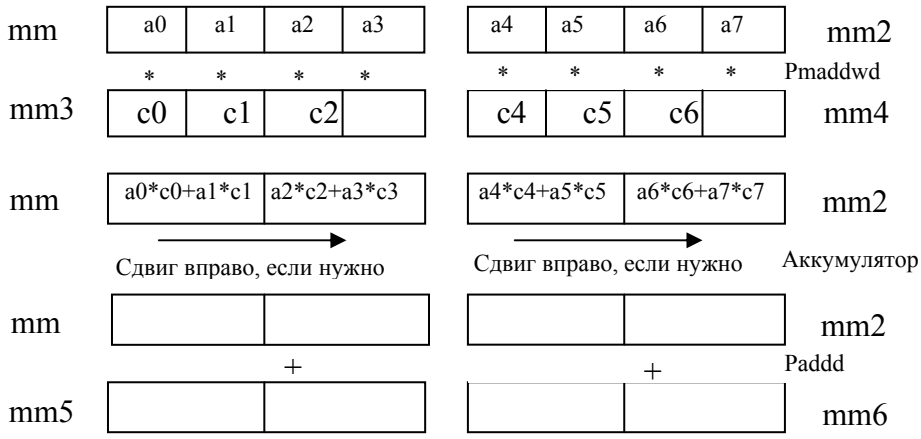
Эти команды особенно важны, когда алгоритму нужна более высокая точность промежуточных вычислений, как, например, в цифровых фильтрах при распознавании образов.

Такой фильтр обычно требует ряда умножений коэффициентов на соответствующие значения пикселов с последующей аккумуляцией полученных значений. Эти операции нуждаются в большей точности, чем та, которую может обеспечить 8-разрядный формат для пикселов. Решением проблемы точности является распаковка 8-разрядных пикселов в 16-разрядные слова, выполнение расчетов в 16-разрядной сетке и затем обратная упаковка в 8-разрядный формат для пикселов перед записью в память или дальнейшими вычислениями.

Рассмотрим некоторые примеры использования технологии MMX для кодирования базовых прикладных операций.

Пример 6. Точечное произведение векторов (Vector Dot Product) является базовым алгоритмом в обработке образов, речи, видео или акустических сигналов.

Приводимый ниже пример показывает, как команда PMADD помогает сделать алгоритм более быстрым. Команда PMADD позволяет выполнить сразу четыре умножения и два сложения для 16-разрядных операндов.Для завершения умножения с накоплением требуется еще команда PADD.

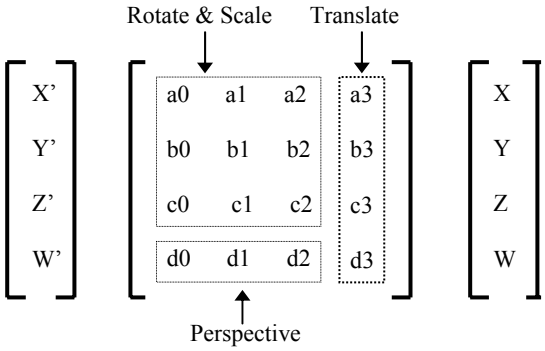


Pmadd – 2
Padd – 2
Сдвиг – 2
Move – 2
Прочие – 5

Итого – 13
вместо 40

Если предположить, что 16-разрядное представление входных данных является удовлетворительным по точности, то для получения точного произведения вектора из восьми элементов требуется восемь MMX-команд: две команды PMADD, две команды PADD, два сдвига (если необходимо) и два обращения в память, чтобы загрузить один из векторов (другой загружен командой PMADD, поскольку она выполняется с обращением в память). С учетом вспомогательных команд на этот пример для технологии MMX требуется 13 команд, а без нее — 40 команд. Следует также учесть, что большинство MMX-команд выполняется за один такт, поэтому выигрыш будет еще более существенным.

Пример 7. Пример на умножение матриц (Matrix Multiply) связан с 3D-играми, количество которых увеличивается каждый день. Типичная операция над 3D-объектами заключается в многократном умножении матрицы размером 4x4 на четырехэлементный вектор. Вектор имеет значения X, Y, Z и информацию о коррекции перспективы для каждого пиксела. Матрица 4x4 используется для выполнения операций вращения (rotate), масштабирования (scale), переноса (translate) и коррекции информации о перспективе для каждого пиксела. Эта матрица 4x4 используется для многих векторов.

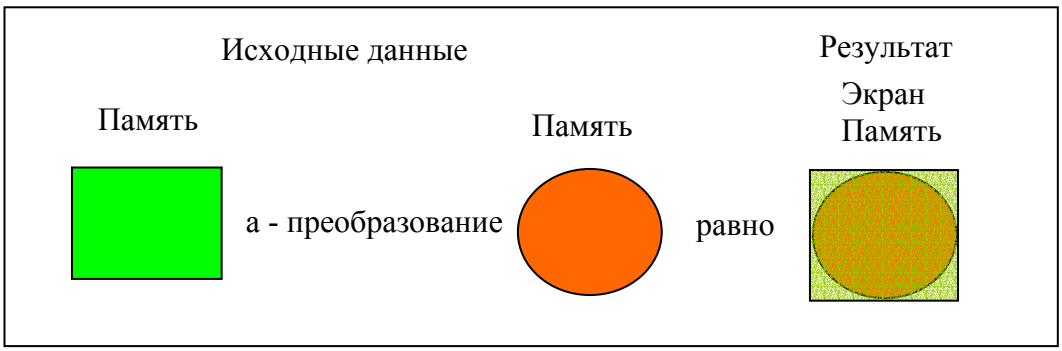


$$X' = a0*x + a1*y + a2*z$$

Приложения, которые работают с 16-разрядным представлением исходных данных, могут широко использовать команду PMADD. Для одной строки матрицы нужна одна команда PMADD, для четырех строк – четыре команды. Более подробный подсчет показывает, что умножение матриц 4x4 требует 28 команд для технологии MMX, а без нее – 72 команды.

Пример 8. Набор команд MMX предоставляет графическим приложениям благоприятную возможность перейти от 8 или 16-разрядного представления цвета к 24-разрядному, или “истинному” цвету, особенностью которого является большой реализм графики, например для игр. Во многих случаях это может быть сделано за то же время, что и для 8-разрядного представления. При 24- и 32-разрядном представлении красный, зеленый и голубой цвета представлены соответственно 8-разрядными значениями.

Наиболее успешно операции с 24-разрядным представлением цвета используются в композиции образов и альфа-смешивании (alpha blending).



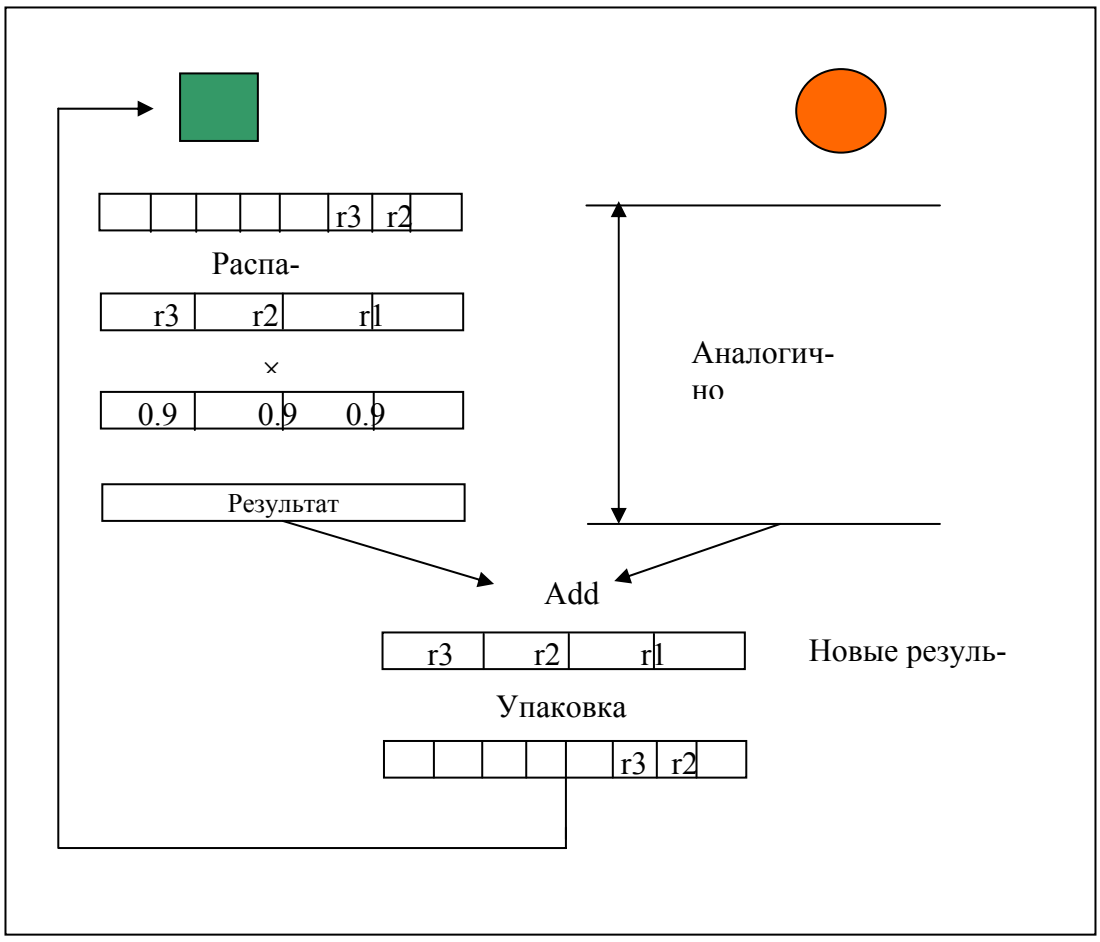
Предположим, что необходимо плавно преобразовать изображение зеленого квадрата в красный круг, которые представлены в 24-разрядном цвете. Причем, для представления каждого из трех цветов отводится один байт. Дале рассмотрим только преобразование для одного цвета.

Основой преобразования является простая функция, в которой *альфа* определяет интенсивность изображения цветка. При полной интенсивности изображения цветка значение *альфа* при его 8-разрядном представлении будет FFh или 255. Если вставить 255 в уравнение преобразования, то интенсивность каждого пиксела квадрата будет 100%, а круга – 0%. Введем обозначения: $П_{рез}$ – пиксел результата, $П_{квадр}$ – пиксел квадрата и $П_{круг}$ – пиксел круга. Тогда уравнение для вычисления результирующего значения пиксела будет таким:

$$П_{рез} = П_{круг}*(альфа/255) + П_{квадр}*(1-(альфа/255))$$

$$альфа = 1…255$$

Ниже представлена программа альфа-преобразования.В этом примере принято, что 24 - разрядные данные организованы так, что параллельно обрабатываются четыре пиксела одного цветового плана, то есть образ разделен на индивидуальные цветовые планы: красный, зеленый и голубой. Сначала обрабатываются первые четыре значения красного цвета, а после них обработка выполняется для зеленого и голубого планов. За счет этого и достигается ускорение при использовании технологии MMX.



Команда распаковки принимает первые 4 байта для красного цвета, преобразует их в 16-разрядные элементы и записывает в 64-разрядный MMX регистр. Значение альфа, которое вычисляется всего один раз для всего экрана, является другим операндом. Команда умножает два вектора параллельно. Таким же образом создается промежуточный результат для круга. Затем два промежуточных результата складываются и конечный результат записывается в память.

Если использовать для представления образов экран с разрешением 640x480 и все 255 ступеней значения *альфа*, то преобразование квадрата в круг потребует выполнения для технологии ММХ 525 млн. операций, а без нее - 1.4 млрд. операций.

Альфа-смешивание позволяет разработчикам игр реалистично представить многие ситуации, например, движение гоночного автомобиля в тумане или смоге, рыб в воде и другое. В этих случаях значение альфа не обязательно будет одинаковым для всего экрана, но от этого принцип обработки не меняется. Наиболее успешно операции с 24-разрядным представлением цвета используются в композиции образов и альфа-смешивании. Техника обработки изображений с помощью альфа-смешивания позволяет создавать различные комбинации для двух объектов А и Б, как представлено ниже в таблице.

Состав операций для альфа-смешивания для двух объектов

Комбинация	Плавное преобразование A в B A*alpha (A)+B*(1-alpha(A))
A over B	Прозрачный образ, накладываемый на фон A+(B*(a-alpha(A))
A in B	Образ А есть там, где В непрозрачен A*alpha(B)
A out B	Образ А есть там, где В прозрачен A*(1-alpha(B))
A top B	(A in B) over B (A*alpha(B) + (B*(1-alpha(A)))
A xor b	(B*(1-alpha(A))) + (A*(1-alpha(B)))

Если подвести итог сказанному, то становится очевидным, что технология ММХ значительно увеличивает возможности мультимедийных и коммуникационных приложений, обеспечивая параллельную обработку для новых типов данных.

Лекция 7. Векторно-конвейерные ЭВМ

7.1. Классификация Флинна

Флинн (Flynn, 1970) ввел два рабочих понятия: поток команд и поток данных. Под потоком команд упрощенно понимают последовательность выполняемых команд одной программы. Поток данных – это последовательность данных, обрабатываемых одним потоком команд. На этой основе Флинн построил свою классификацию параллельных ЭВМ с крупнозернистым параллелизмом:

1) *ОКОД* (одиночный поток команд – одиночный поток данных) или SISD (Single Instruction – Single Data). Это обычные последовательные ЭВМ, в которых выполняется одна программа.

2) *ОКМД (одиночный поток команд – множественный поток данных)* или SIMD (Single Instruction – Multiple Data). В таких ЭВМ выполняется единственная программа, но каждая ее команда обрабатывает много чисел. Это соответствует векторной форме параллелизма.

3) *МКОД (множественный поток команд – одиночный поток данных)* или MISD (Multiple Instruction – Single Data). Здесь несколько команд одновременно работает с одним элементом данных. Этот класс не нашел применения на практике.

4) *МКМД (множественный поток команд – множественный поток данных)* или MIMD (Multiple Instruction – Multiple Data). В таких ЭВМ одновременно и независимо друг от друга выполняется несколько программных ветвей, в определенные промежутки времени обменивающихся данными. Такие системы обычно называют многопроцессорными.

Таким образом, далее будут рассмотрены только два класса параллельных ЭВМ: SIMD и MIMD. SIMD –ЭВМ в свою очередь делятся на два подкласса: векторно-конвейерные ЭВМ и процессорные матрицы.

7.2. Арифметические конвейеры

На рис.7.1 представлен принцип постороения системы с арифметическим конвейером. Обычно в конвейере команд самым медленным звеном («узким» местом трубопровода) является этап исполнения арифметических операций EXE. Например, время выполнения некоторых операций в АЛУ в тактах таково:

```
+R1, R2      1
+R1, [M]     2
+[M], R1     3
*R1,R2     11 тактов
```

Существуют способы разбить этот этап на несколько более мелких арифметических этапов, снабдив каждый из них аппаратурой. Это и будет арифметический конвейер в составе конвейера команд. Теперь скорость всего конвейера определяется тактом арифметического конвейера $\Delta_{ак}$.

Ниже приведены некоторые способы построения арифметических конвейеров. На рис.7.2 представлена схема конвейера для сложения потока чисел с плавающей запятой, а на рис. 7.3 – для умножения чисел. Такие конвейеры можно построить практически для всех команд.

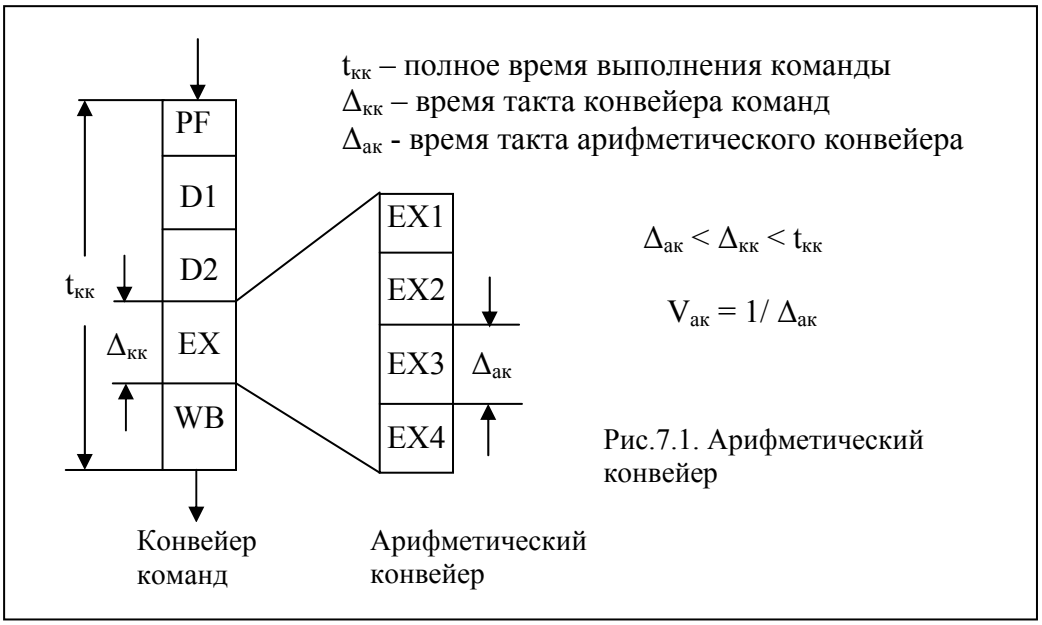


Рис.7.1. Арифметический конвейер

Рис.7.2. Арифметический конвейер для сложения чисел с плавающей запятой

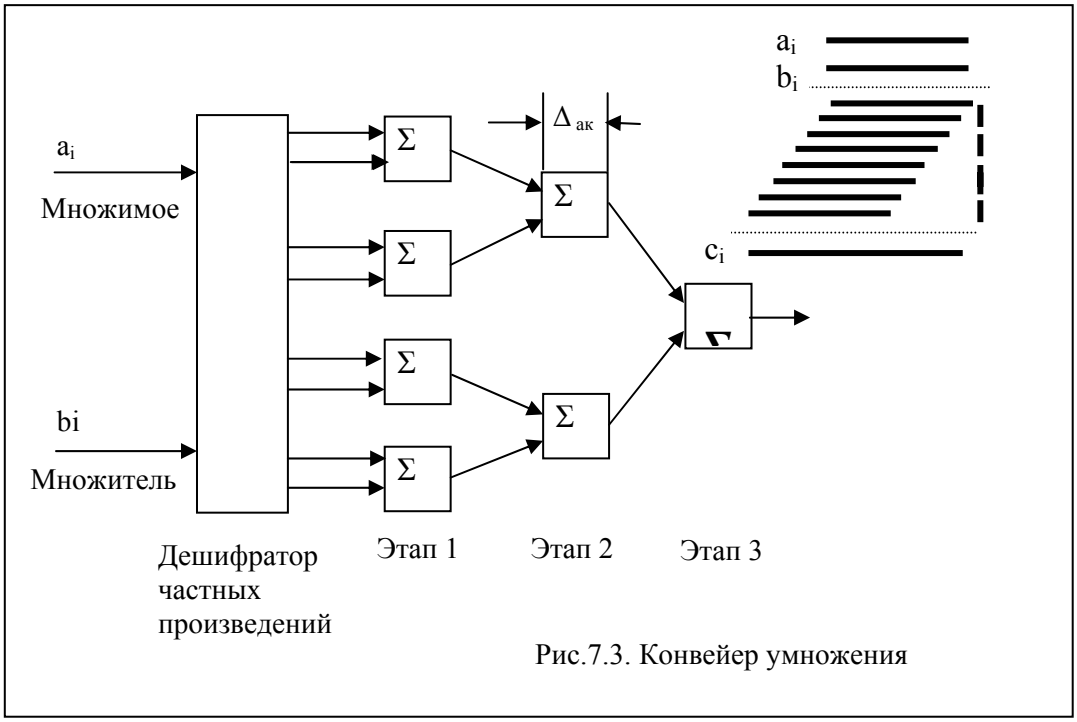


Рис.7.3. Конвейер умножения

7.3. Векторно-конвейрная ЭВМ CRAY.

Впервые арифметические конвейеры были использованы для целей обработки числовых векторов в ЭВМ STAR-100, запущенной в США в 1973 г., а затем на этом принципе была построена знаменитая ЭВМ . ЭВМ CRAY стала родоначальником множества векторно-конвейерных ЭВМ различных компаний и разного быстродействия. Однако, принцип работы всех этих машин вкладывается в организацию CRAY-1, как “скрипка в футляр”.

ЭВМ CRAY [10] состоит из скалярного и векторного процессоров (рис.7.4). Скалярный процессор выбирает из памяти и выполняет скалярные и векторные команды, но скалярные – целиком, а арифметическую часть векторных команд передает в векторный процессор. В соответствии с законом Амдала скалярный процессор должен обладать повышенным быстродействием.

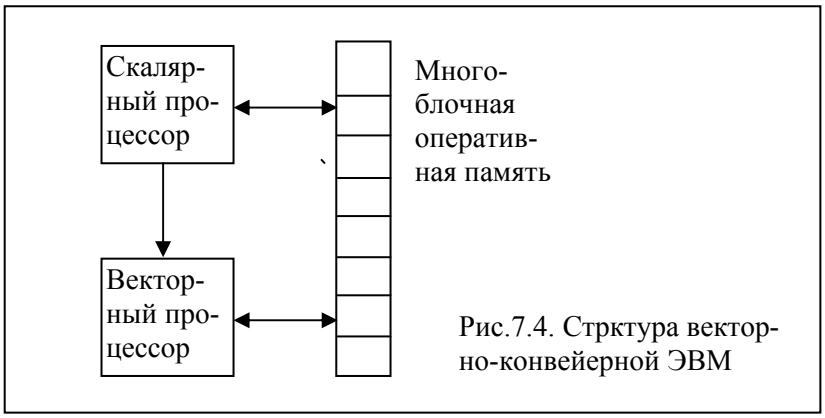


Рис.7.4. Стрктура вектор-но-конвейерной ЭВМ

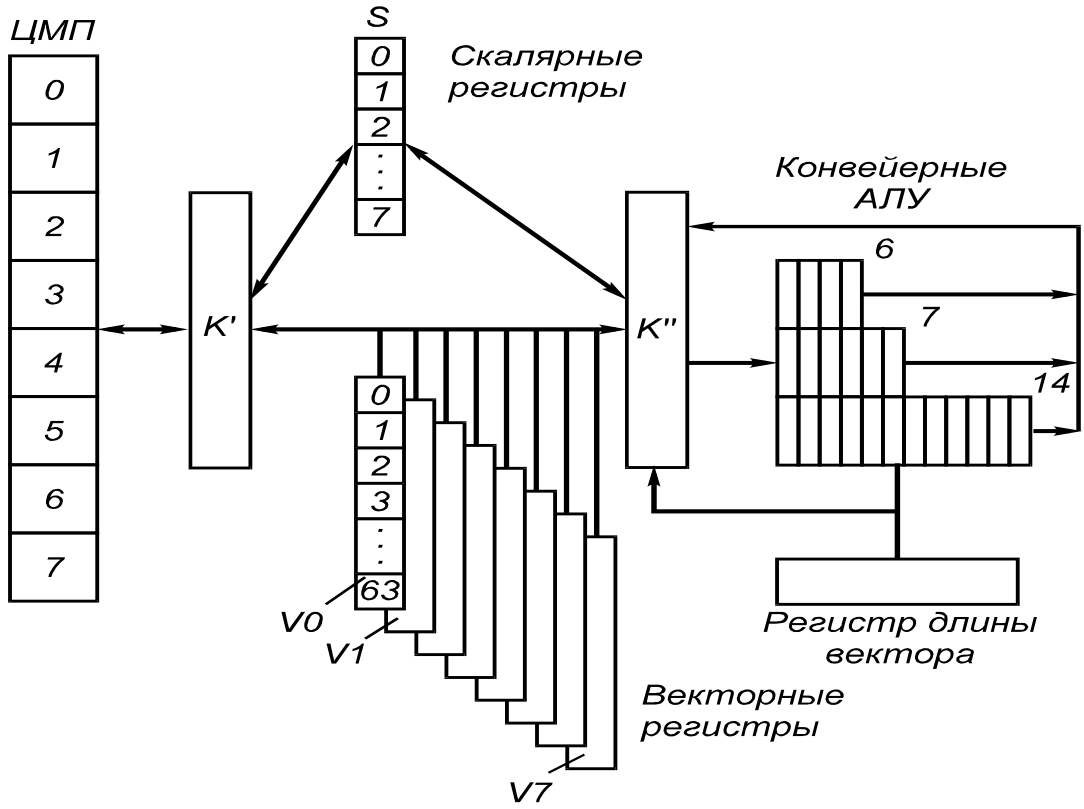


Рис.7.5 Структура векторного конвейерного процессора

Схема векторного процессора ЭВМ представлена на рис.7.5. Будем считать, что вектор — это одномерный массив, который образуется из многомерного массива, если один из индексов не фиксирован и пробегает все значения в диапазоне его изменения. В некоторых задачах векторная форма параллелизма представлена естественным образом. В частности, рассмотрим задачу перемножения матриц на векторном процессоре.

Для перемножения матриц на последовательных ЭВМ неизменно применяется гнездо из трех циклов:

```
DO 1 I = 1, L
DO 1 J = 1, L
DO 1 K = 1, L
1  C(I, J) = C(I, J) + A(I, K) ∗ B(K, J)
```

Внутренний цикл может быть записан в виде отрезка программы на фортраноподобном параллельном языке:

```
R (∗) = A (I, ∗) ∗ B (∗, J)                    (7.1)
C (I, J) = SUM R (∗)
```

Здесь R(∗), A(I, ∗), D(∗, J) — векторы размерности L; первый оператор представляет бинарную операцию над векторами, а второй — унарную операцию SUM суммирования элементов вектора.

Главная особенность векторного процессора — наличие ряда *векторных регистров V*, каждый из которых позволяет хранить вектор длиною до 64 слов. Это своего рода РОН, значительно ускоряющие работу векторного процессора.

Рассмотрим, как будет выполняться программа (1) в векторном процессоре. На условном языке ассемблерного уровня программа может быть представлена следующим образом:

```
1  LD L, Vi, A
2  LD L, Vj, B
3  MP Vk, Vi, Vj                               (7.2)
4  SUM Sn, Vk
```

Операторы 1 и 2 соответствуют загрузке слов из памяти с начальными адресами A и B в регистры Vi, Vj; оператор 3 означает поэлементное умножение векторов с размещением результата в регистре Vk; оператор 4 — суммирование вектора из Vk с размещением результата в Sn.

Соответственно этой программе векторные регистры сначала потактно заполняются из ЦМП, а затем слова из векторных регистров потактно (одна пара слов за такт) передаются в конвейерные АЛУ, где за каждый такт получается один результат.

Рассмотрим характеристики быстродействия векторного процессора на примере выполнения команды MP Vi, Vj, Vk. Число тактов, необходимое для выполнения команды, равно: $r = m_* + L$, где m_* — длина конвейера умножения.

Поскольку на умножение пары операндов затрачивается $k = (m_* + L)/L$ тактов, то быстродействие такого процессора

$$V = 1/(K\Delta t) = \frac{L}{(m_* + L)\Delta t},$$

где Δt — время одного такта работы конвейера.

Быстродействие конвейера зависит от величины L (рис. 7.6, кривая 1). При $L > m_*$ величина $K = 1$ и $V = 1/\Delta t$. Обычно для векторных процессоров стараются сделать Δt малым, в пределах 1...2 нс, поэтому быстродействие при выполнении векторных операций может достигать 500...100 млн оп/с.

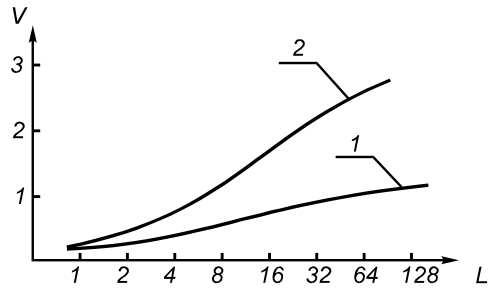


Рис.7.6. Зависимость быстродействия векторного процессора от длины вектора: $m_{\text{ЦМП}} = 4$; $m_* = 7$; $m_+ = 6$

Важной особенностью векторных конвейерных процессоров, используемой для ускорения вычислений, является механизм зацепления. *Зацепление* — такой способ вычислений, когда одновременно над векторами выполняется несколько операций. В частности, в программе (2) можно одновременно производить выборку вектора из ЦМП, умножение векторов, суммирование элементов вектора. Поэтому программу можно переписать следующим образом:

LD L, Vi, A
ЗЦ Sn, Vi, B

Здесь команда зацепления (ЗЦ) задает одновременное выполнение операций в соответствии со схемой соединений (рис.7.7).

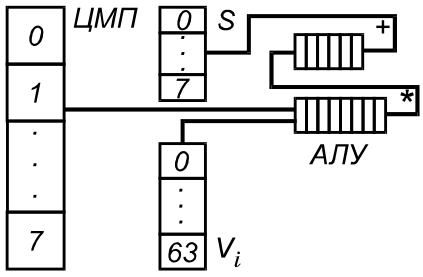


Рис.7.7. Выполнение операции зацепления в конвейерном процессоре

Для команды ЗЦ получаем:

$$r = m_{ЦМП} + m_* + m_+ + L,$$
$$K = (m_{ЦМП} + m_* + m_+ + L)/L,$$
$$V = n/(K\Delta t),$$

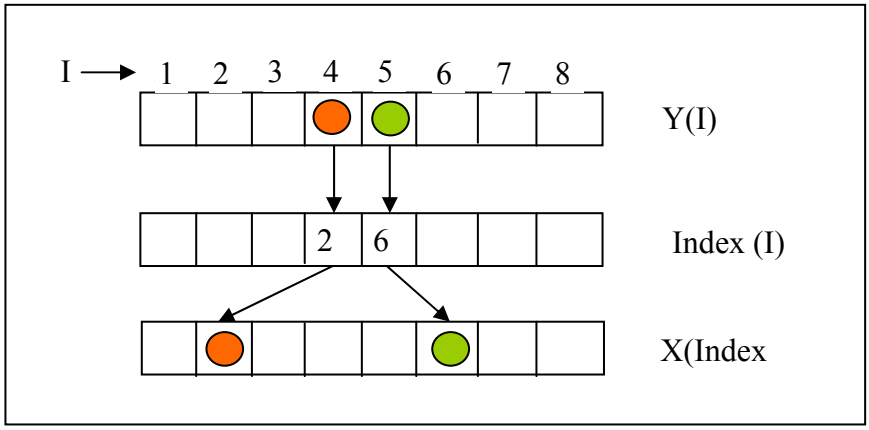
где n — число одновременно выполняемых операций. В случае команды ЗЦ $n = 3$ (рис. 7.6, кривая 2). При $L >> m_i$ и $\Delta t = 1...2$ нс в зацеплении быстродействие равно 1500...3000 млн оп/с.

Такое быстродействие достигается не на всех векторных операциях. Для векторных ЭВМ существуют “неудобные” операции, в которых ход дальнейших вычислений определяется в зависимости от результата каждой очередной элементарной операции над одним или парой операндов. В подобных случаях L приближается к единице. К таким операциям относятся операции рассылки и сбора, которые можно определить следующими отрезками фортран-программ:

1) рассылка

```
      DO 1 I = 1, L
1     X(INDEX (I)) = Y (I)
```

На рисунке показано, что изменение индекса I с постоянным шагом 1 для Y (I) приводит при обращении к X(INDEX (I)) к неравномерному рассеянию адресов памяти, поэтому конвейер памяти не может работатьэффективно. Это же относится и к следующей операции сбора.



2) сбор

```
DO 1 I = 1, L
1    Y (I) = X (INDEX (I))
```

Названные операции имеются в задачах сортировки, быстрого преобразования Фурье, при обработке графов, представленных в форме списка, и во многих других задачах.

Скалярный процессор

Ускорение векторных вычислений согласно закону Амдала очень чуствительно к времени выполнения скалярных операций. Это время можно уменьшить двояко:

- Уменьшить количество скалярных операций, что не всегда возможно
- Увеличить быстродействие скалярного процессора. Именно это и сделано в CRAY. Здесь скалярный процессор выполнен по принципу потока данных (DF).

Рассмотрим структуру и функционирование *скалярного конвейерного процессора* в целом (рис.7.8). На рисунке обозначено:

- ЦМП — центральная многоблочная память;
- БАО — буфер адресов операндов;
- АЛУ — конвейеризованные арифметико-логические устройства для сложения, умножения и деления чисел с плавающей запятой;
- К’, К’’— коммутаторы памяти и АЛУ соответственно;
- БС РОН — блок состояния РОН;
- УПК — указатель номера пропущенной команды;
- СчК — счетчик команд;
- 1— шина адреса команд;
- 2— шина управления выполнением команд обращения к памяти;
- 3 — шина заполнения БК;
- 4 — шина смены состояний РОН;
- 5 — шина управления выполнением регистровых команд

Процессор содержит несколько конвейерных АЛУ. Для разных операций АЛУ имеют различную длину конвейера (на рис.7.8 она равна 6, 7 и 14 позициям). В процессоре используются команды двух классов: команды обращения в память и регистровые команды для работы с РОН. Бу-

фер команд имеет многостраничную структуру, что позволяет во время работы УУ с одной страницей производить заранее смену других страниц.

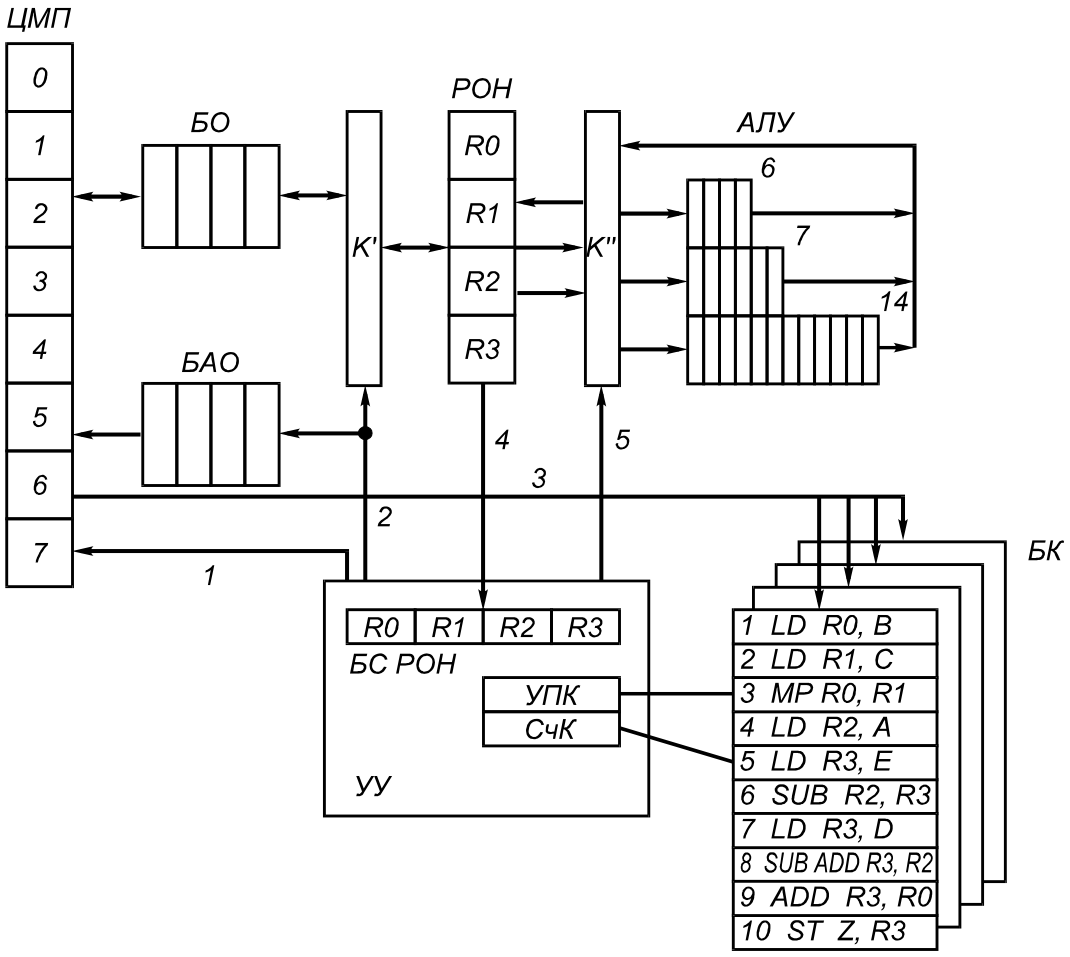


Рис.7.8 Организация скалярного конвейерного процессора:

Для изучения работы процессора (см. рис.7.8) использован отрезок программы, соответствующий вычислению выражения

$$Z = A - (B * C + D) - E.$$

В программе: *LD* — команда загрузки операнда из памяти в регистр; *MP*, *SUB, ADD* — команды умножения, вычитания и сложения соответственно; *ST* — команда записи операнда из регистра в память.

Состояние некоторых регистров при выполнении программы показано в табл.1. В последней колонке таблицы приведен порядок запуска команд на исполнение. Некоторые команды могут опережать по запуску команды, находящиеся в программе выше запущенной. Например, команды 4 и 5 выполняются ранее команды 3. Это возможно благодаря наличию в программе локального параллелизма и нескольких АЛУ в структуре процессора. Однако подобные “обгоны” не должны нарушать логики исполнения программы, задаваемой ее ИГ (рис.7.9). Любая операция согласно рисунку может быть запущена только после того, как подготовлены соответствующие операнды. Это достигается путем запрета доступа в определенные РОН до окончания операции, в которой участвуют данные РОН. Состояния РОН отражены в специальном БС РОН.

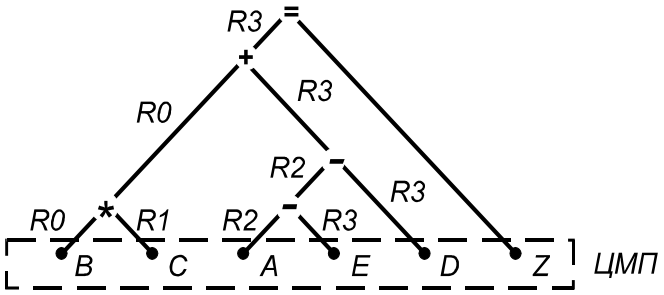


Рис. 7.9. Информационный граф программы: Ri — регистр общего назначения

В табл.7.1 приведено также описание нескольких тактов работы процессора. Принято, что выборка операнда из ЦМП занимает четыре такта, сложение – 6, а умножение – 7 тактов. Кроме того, считается, что за один такт процессора устройство управления запускает на исполнение одну команду или просматривает в программе до четырех команд.

Таблица 7.1.Порядок исполнения программы в скалярном конвейере

NN такта	Состояние РОН				Номер команды
	*R*0	*R*1	*R*2	*R*3	
1	4	-	-	-	1
2	3	4	-	-	2
3	2	3	4		4
4	1	2	3	4	5
5	*x*	1	2	3	-
6	7	-	1	2	3
7	6	-	*x*	1	-
8	5	-	6	*x*	6
9	4	-	5	4	7
10	3	-	4	3	-

Сделаем пояснения к таблице.

Такт 1. Анализ БС РОН показывает, что все РОН свободны, поэтому команда 1 запускается для исполнения в ЦМП. В столбец *R*0 записывается 4, что означает: *R*0 будет занят четыре такта. После исполнения каждого такта эта величина уменьшается на единицу. В структуре процессора занятость *R*0 описывается установкой разряда R0 БС РОН в 1, а затем сброс *R*0 в 0 по сигналу с шины 4, который появляется в такте 4 после получения операнда из памяти.

Такт 2. Запускается команда 2 и блокируется регистр *R*1.

Такт 3. Просматривается команда 3, она не может быть выполнена, так как после анализа БС РОН нужные для ее исполнения регистры *R*0 и *R*1 заблокированы. Команда 3 пропускается, а ее номер записывается в УПК. Производится анализ условий запуска следующей (по состоянию СчК) команды. Команда 4 может быть запущена и запускается.

Такт 4. Просмотр БК начинается с номера команды, записанной в УПК. Команда 3 не может быть запущена, поэтому запускается команда 5.

Такт 5. Команда 3 не может быть запущена, так как занят регистр *R*1, однако регистр *R*0 освободился и будет использоваться командой 3, он снова блокируется (символ *x*). Просмотр четырех следующих команд показывает, что они не могут быть запущены, поэтому в такте 5 для исполнения выбирается новая команда.

Такт 6. Запускается команда 3.

В дальнейшем процесс происходит аналогично. Можно заметить, что за 10 тактов, описанных в табл. 2.1, в процессоре запущено семь команд, что соответствует $10/7 \approx 1{,}5$ такта на команду.

Лекция 8. Коммутаторы

8.1. Сети коммутации

Коммутатор (сеть коммутации) представляет собой совокупность следующих элементов и соединений между ними:

- два набора узлов, которые называются входными и выходными. Каждый входной узел имеет один вход и несколько выходов, каждый выходной узел имеет несколько входов и один выход.
- Каждый узел (входной или выходной) имеет кроме того настроечный вход, на который подается код настройки данного узла, в соответствии с которым определенный вход данного узла подключается на заданный кодом выход.
- выходы всех входных узлов соединены с входами выходных узлов по определенной схеме (рисунок соединений), которая определяет возможности данного коммутатора.

Коммутатор является аппаратной средой, в которой реализуются программные связи, задаваемые совокупностью настроек всех узлов. Связи различного вида возникают между процессорами в ходе выполнения вычислительного алгоритма. Не всякий коммутатор в состоянии реализовать требуемые виды связей.

Коммутатор плюс программные средства, обеспечивающие передачу данных в локальной или глобальной компьютерной сети в соответствии с протоколами TCP/IP, называется коммуникатором [2]. В этой лекции коммуникаторы не рассматриваются.

Всего между N входами и M выходами существует N^M различных связей, включая парные связи (“один — одному”), связи “один — многим”, смешанные связи. Сеть, осуществляющая N^M соединений, называется *обобщенной соединительной сетью* (рис.8.1). Если сеть осуществляет только парные связи, она называется *соединительной*, в ней возможно $N!$ соединений (рис.8.1,б, в). В соединительных сетях любой допустимый набор связей (перестановка) выполняется за один такт и без конфликтов.

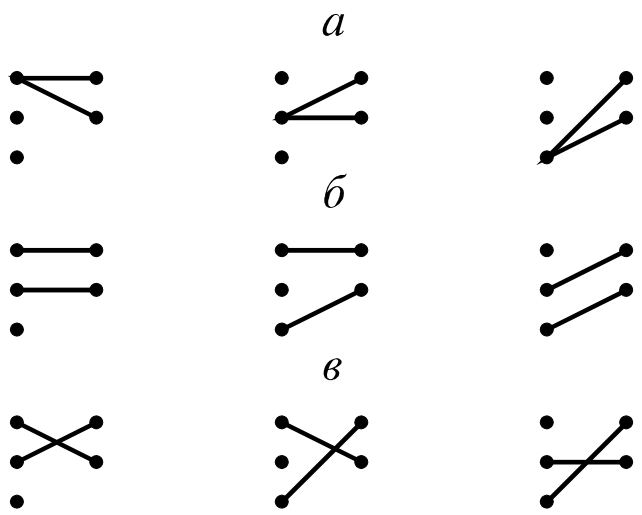


Рис.8.1 Варианты соединений в обобщенной соединительной сети для N = 3, M = 2: *а* — связи “один — многим”; *б*, *в* — парные связи

Для процессорных матриц (ПМ) соединительные сети имеют большее значение, чем обобщенные соединительные сети. Обобщенная соединительная сеть, реализующая N^M соединений, изображается в виде двудольного графа (рис. 8.2,а). Она представляет собой решетку, на пересечениях которой осуществляются необходимые замыкания (рис. 8.2,б). Такая решетка называется координатным переключателем и имеет существенный недостаток: объем оборудования в ней пропор-

ционален $N \times M$, поэтому координатные переключатели используются для ПП размером не более 16...32 процессора.

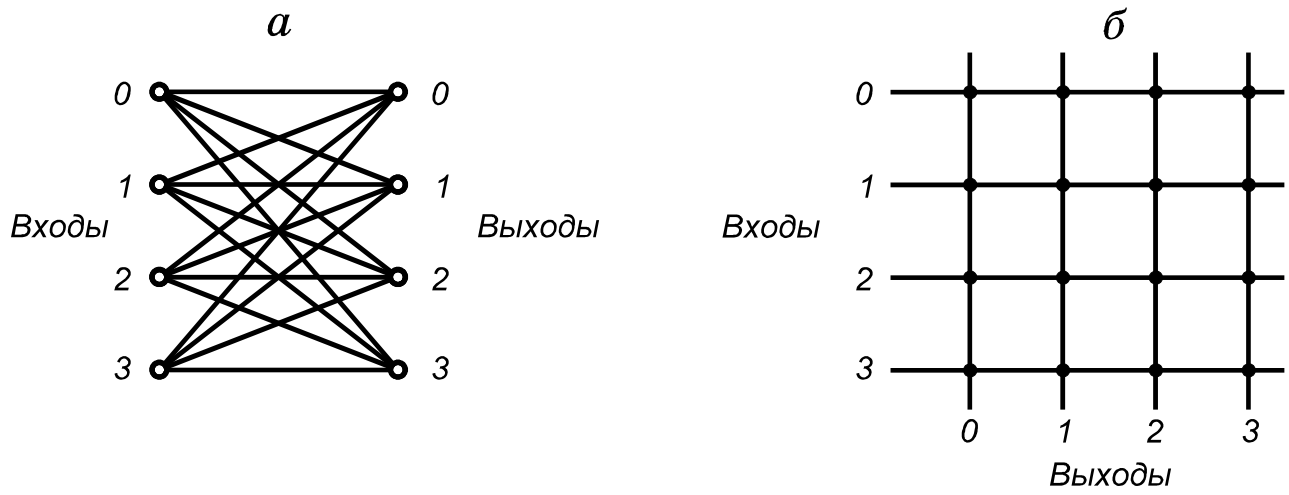


Рис. 8.2. Два представления координатного переключателя размером 4 × 4:
а — в виде двудольного графа; *б* — в виде решетки связей. Термины вход и выход относится только к коммутатору.

Коммутатором, противоположным по своим свойствам полному координатному соединителю, является общая шина UNIBUS — общая магистраль. Этот коммутатор (рис.8.3) способен за один такт выполнить соединение только между одним входом и одним выходом в отличие от координатного соединителя, который за один такт выполняет все соединения.

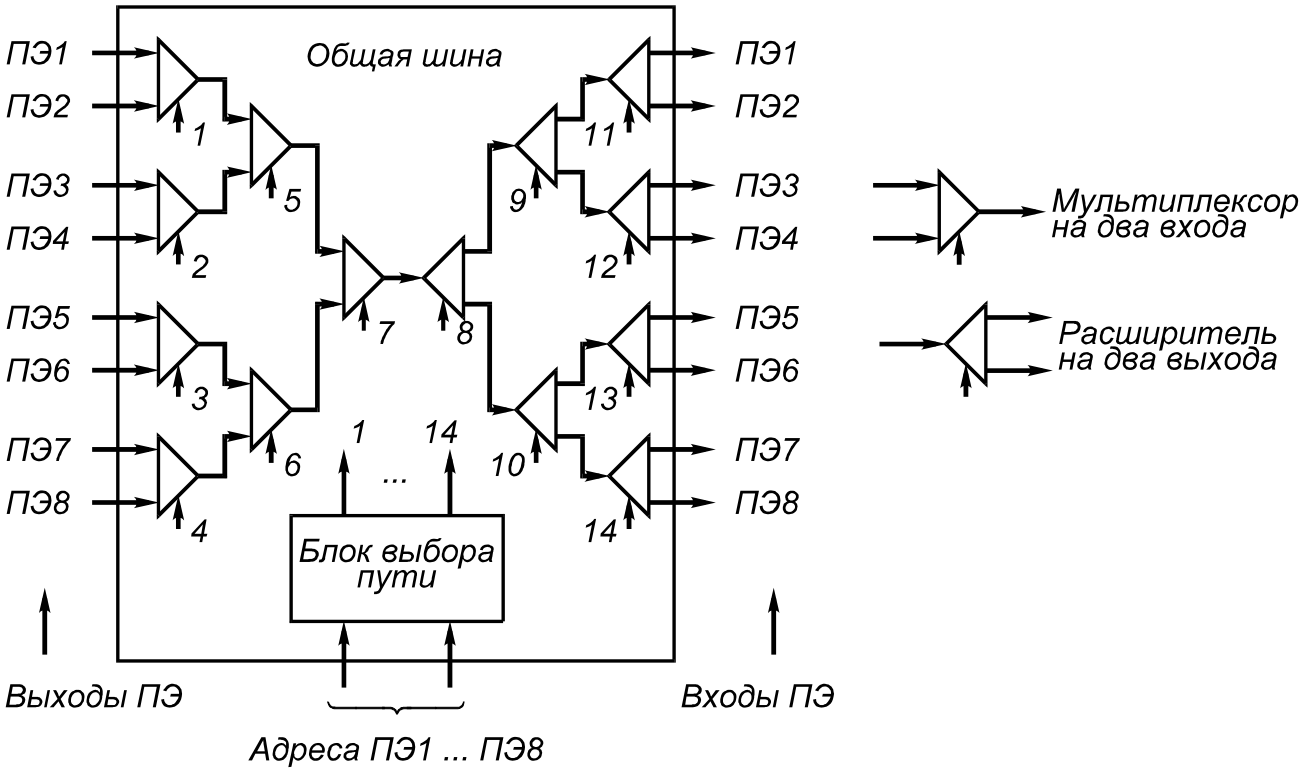


Рис.8.3. Коммутатор типа “общая шина”

Рассмотрим три варианта изображения некоторой многоэлементной связи:

- С помощью рисунка (рис.8.4). Это наиболее наглядный вид изображения связей, однако он не может быть использован в программе.

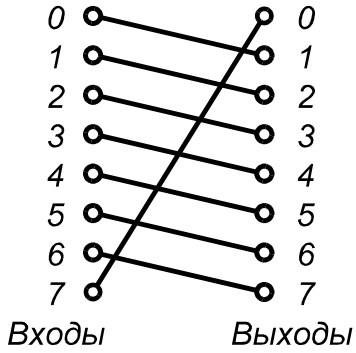


Рис.8.4. Изображение связей в виде схемы соединений

- С помощью таблицы (пакета) связей:

Входы	0	1	2	3	4	5	6	7
Выходы	1	2	3	4	5	6	7	0

Здесь позиция целого числа в таблице означает номер входа, а число в этой позиции — номер выхода, на который замкнута данная связь.

- С помощью закона, позволяющего на основании номеров (адресов) входов вычислять номера (адреса) выходов. Такие связи называются *перестановками.*

В зависимости от характера различают регулярные и распределенные случайным образом связи. Для *регулярных связей* используются перестановки. Они, как правило, используются при решении систем уравнений, выполнении матричных операций, вычислении быстрого преобразования Фурье и т. д. *Случайно распределенные связи* возникают при обработке нечисловой информации (графов, таблиц, текстов) и могут задаваться только в виде таблицы случайных связей.

Рассмотрим некоторые виды перестановок, упоминаемые в литературе по параллельным ЭВМ.

1. Перестановки с замещением (рис.8.5). Закон получения номера выхода следующий:

$$\{b_n, ..., b_k, ..., b_1\}_j = \{ a_n, ..., \bar{a}_k, ..., a_1\}_i,$$

где b_k, a_k — k-е двоичные разряды номера i-го входа и j-го выхода связи; $\bar{a}_k$ — двоичная инверсия k-го разряда.

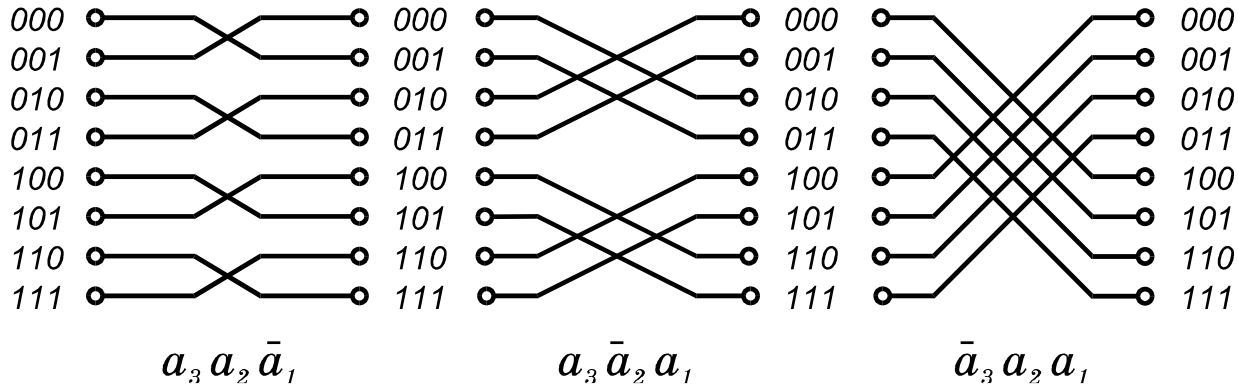


Рис. 8.5. Перестановки с замещением

2. Перестановки с полной тасовкой (рис.8.6). Они соответствуют циклическому сдвигу влево всех разрядов или группы разрядов номера нужного входа. Полученный двоичный результат дает номер соответствующего выхода.

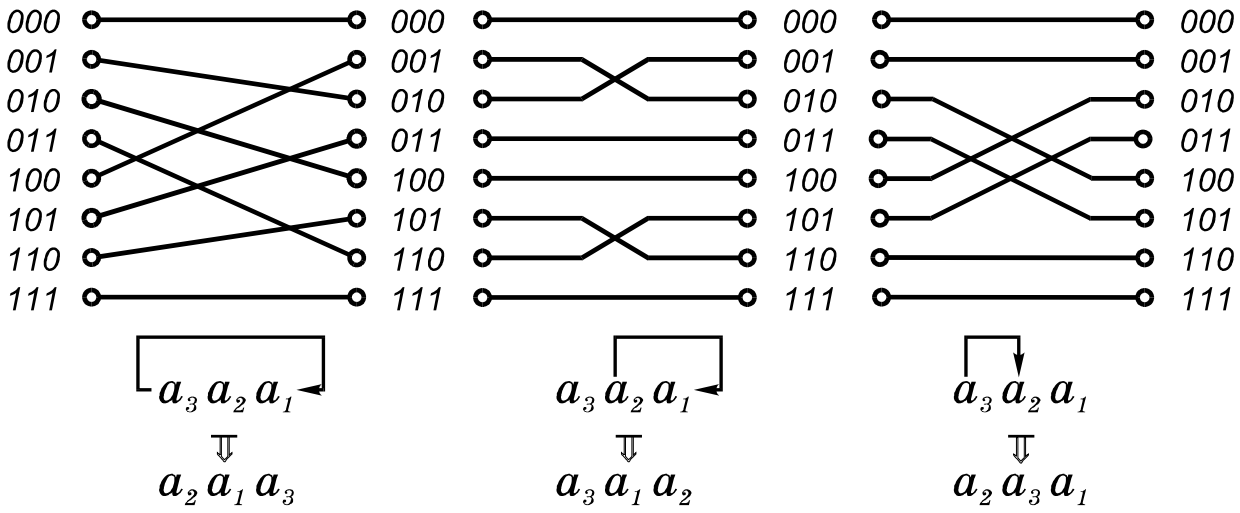


Рис.8.6. Перестановки с полной тасовкой

3. Перестановки со смещением (рис.8.7). Они выполняются по формуле

$$i_B = (i_A + r)_N ,$$

где r — величина сдвига; N — число входов. Вычисления производятся по модулю N. Для случая $i_B = (i_A + 1)_4$ в разрядном представлении номеров входов и выходов рассматриваются только два младших разряда.

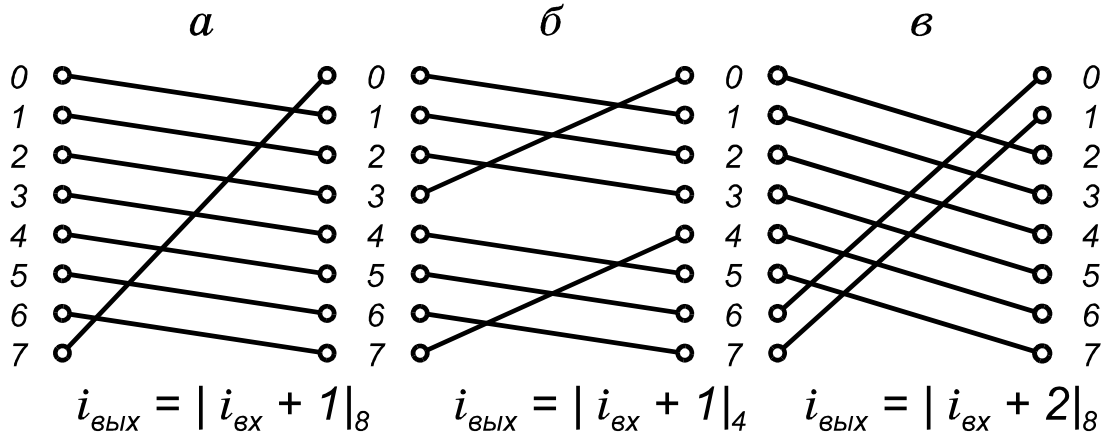


Рис.8.7. Престановки со смещением

Если в некотором коммутаторе предусмотрено (конструктивно) использование m перестановок, т. е.

$$КМ = \{П_1, П_2, ..., П_m\},$$

где $П_k$ — перестановка типа k, то для задания кода любой перестановки в КМ для использования необходимо управляющее слово с разрядностью $\log_2 m$.

Координатный переключатель в состоянии реализовать за один такт все виды связей как регулярных, так и случайных. Однако объем оборудования такого коммутатора растет слишком быстро. Поэтому надо искать другие типы коммутаторов, имеющих несколько меньше возможностей по числу устанавливаемых связей, но требующих значительно меньшего объема оборудования.

8.2. Коммутаторы типа среда

Среда — коммутатор, каждый узел которого связан только со своими ближайшими соседями. В коммутаторе с односторонним сдвигом (рис.8.8, а) каждый ПЭ связан с двумя соседними. В общем случае коммутатор (см. рис.9.1, а) в состоянии выполнить за N тактов любую перестановку, где N — число ПЭ. При этом каждую единицу передаваемой информации можно снабдить номером процессора приемника. По мере сдвигов, когда информация достигнет соответствующего ПЭ, она должна “осесть” в этом ПЭ.

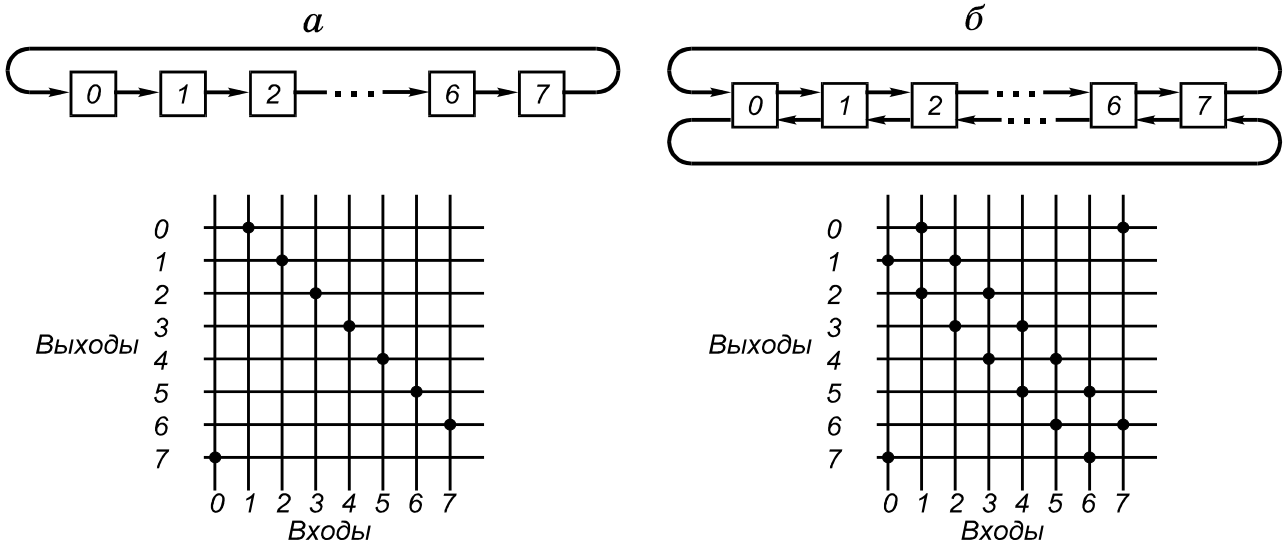


Рис.8.8. Структура кольцевых коммутаторов

Коммутатор с односторонним сдвигом (см. рис.8.8,а) представлен в сетке координатного переключателя. Точки на пересечениях означают, что все эти соединения могут осуществляться одновременно. Такое представление коммутаторов удобно для теоретических исследований. По данному представлению видно, что коммутатор с односторонним сдвигом — часть полного координатного переключателя и поэтому не является соединительной сетью.

Любая совокупность перестановок в коммутаторе с двусторонним сдвигом (рис.8.8, б) будет выполнена за более короткое время, чем в коммутаторе на рис.8.8,а. В частности, чтобы передать единицу информации по связи ПЭ0 → ПЭ7, в первом коммутаторе требуется семь тактов, во втором — один такт. Другими словами, максимальное расстояние между узлами в одностороннем кольце примерно пропорционально N, а в двухстороннем – N/2, где N- число узлов.

В двумерном коммутаторе каждый ПЭ соединен двусторонними связями с четырьмя соседними (справа, слева, снизу, сверху) (рис.8.9,а). Внешние связи могут в зависимости от решаемой задачи программно замыкаться по-разному: вытягиваться в цепочку (рис.8.9,б), замыкаться в виде вертикально (рис.8.9,в) и горизонтально (рис.8.9,г) расположенных цилиндров, в виде тороида (рис.8.9,д). Коммутационный элемент (КЭ) здесь обычно соединен с оборудованием ПЭ, и объем оборудования всего коммутатора растет пропорционально числу процессоров N.

Коммутатор с матричной схемой соединений выполняет некоторый набор перестановок быстрее, чем рассмотренные выше коммутаторы с кольцевыми схемами.

Под кубическим коммутатором понимается пространственная трехмерная структура, где каждый процессор соединен с шестью соседними: четырьмя в своей плоскости и двумя в прилегающих плоскостях.

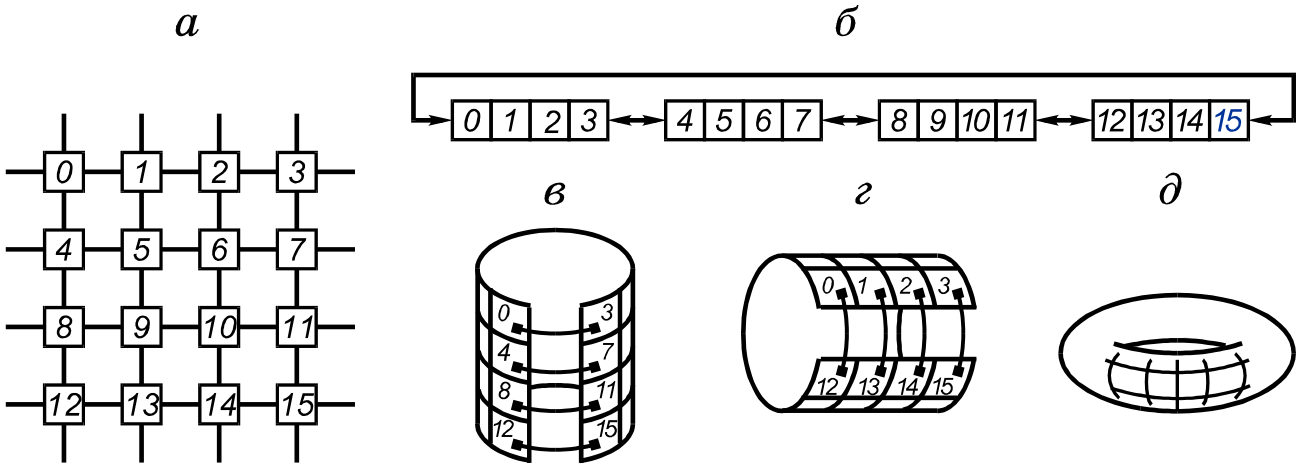


Рис.8.9. Коммутатор с матричной схемой соединения

В общем случае все среды можно назвать *n*-мерными кубами, тогда кольцевой коммутатор имеет размерность $n = 1$, а матрица — $n = 2$ и т. д. Большое распространение в параллельной вычислительной технике получили среды с размерностями 2 и 3.

Чем больше *n* при неизменном числе процессоров, тем больше радиус связей и тем меньше *D*. Под радиусом связи понимается число процессоров, напрямую связанных с данным ПЭ.

Величина *n* может быть и больше трех. Покажем общий прием построения *n*-кубов произвольной размерности (рис.8.10).

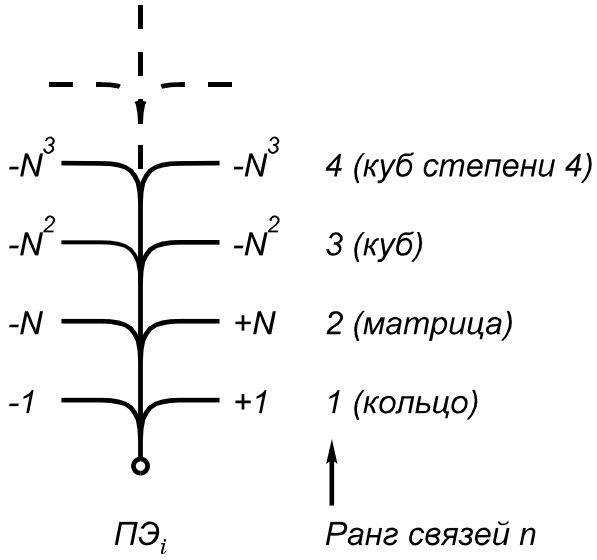


Рис.8.10. Принцип построения коммутаторо произвольной размерности

Для примера рассмотрим возможные связи элемента 9 матрицы, изображенной на рис. 8.9, а. Этот элемент в своей строке связан с элементами 8 и 10, адреса которых отличаются на -1 и +1. Если бы данная строка составляла независимый коммутатор с размерностью 4 ($n = 1$, $N = 4$), то

связи любого ПЭi такого коммутатора описывались бы рангом 1. Если элемент 9 находится в матрице, то его связи с элементом 5 верхней строки и элементом 13 нижней строки отличаются на $+N$ и $-N$, где $N = 4$ — размер строки. Следовательно, связи элемента ПЭi матричного коммутатора описываются рангом 2 (см. рис.8.10).

Если бы элемент 9 находился в кубе, то номера смежных элементов соседних плоскостей отличались бы на $\pm N^2$, в кубе степени 4 — на $\pm N^3$ и т. д., что и показано на рис. 8.10. Такая методика позволяет построить куб любой степени и вычислить величину D.

8.3. Каскадные коммутаторы

Многокаскадные коммутаторы помогают создать более дешевые варианты соединительных и обобщенных соединительных сетей. Координатный переключатель $N \times N$ можно свести к двум ($N/2 \times N/2$) переключателям с замещением В свою очередь переключатели $N/2 \times N/2$ могут быть разложены на более мелкие подобным же образом. Пример сети, приведенной до уровня стандартных КЭ размерностью 2x2, дан на рис.8.11. По определению, это соединительная сеть (сеть Бенеша), обеспечивающая $N!$ возможных перестановок, каждая из которых выполняется за один такт работы сети. В такой сети отсутствуют конфликты для любых парных соединений.

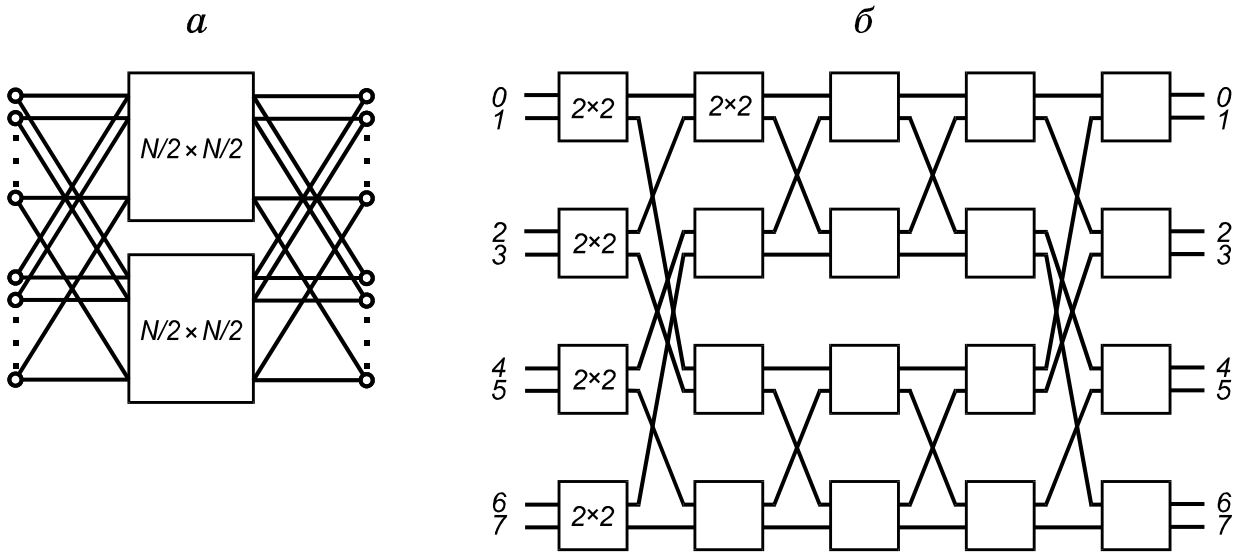


Рис.8.11. Способ построения сети Бенеша

Стандартный КЭ размерностью 2×2 и возможные в нем соединения показаны на рис.8.12. Иногда для построения больших сетей применяются элементы 4×4 или 8×8. Основной недостаток сети Бенеша — большое количество элементов, необходимых для ее построения.

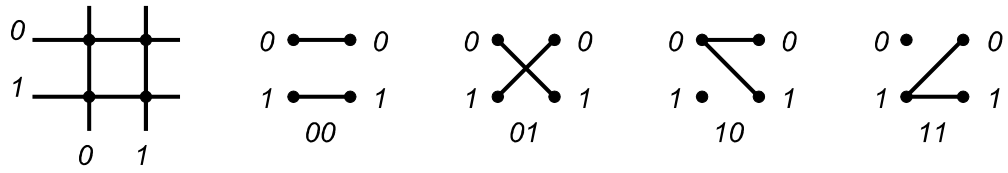


Рис. 8.12. Коммутационный элемент размерностью 2×2 и допустимые перестановки. Внизу приведены коды нумерации перестановок

Вследствие этого на практике используются коммутаторы со значительно меньшим объемом оборудования, не являющиеся соединительными сетями в полном смысле, однако обеспечивающие широкий класс перестановок. В таких коммутаторах возможны конфликты, которые разрешаются последовательно во времени. Примером конфликта может быть ситуация, когда для двух входов требуется одна и та же выходная линия.

Широко используются неполные соединительные сети: омега-сеть, сеть Бенеша, *R*-сеть и др. Омега-сеть относится к типу сетей тасовки с замещением.

Коммутационные сети могут работать в двух режимах: коммутации каналов и коммутации сообщений. В первом случае сначала с помощью адресной системы устанавливается прямой тракт между парой соединяемых точек, а затем по этому тракту передается информация. Такой вид связи используется, например, в телефонных сетях. Во втором случае каждая передаваемая единица информации снабжается адресом требуемого выхода и перемещается от узла n_i к узлу m_{i+1}, где i — номер каскада; n и m — номера коммутационных узлов в каскадах. После того как информация принята узлом m_{i+1}, линия, соединявшая n_i и m_{i+1}, освобождается для других соединений. Узел m_{i+1} по адресной части сообщения определяет дальнейший маршрут сообщения.

В матрицах процессоров используется как коммутация каналов, так и коммутация сообщений. Оценим основные характеристики коммутаторов, описанных выше. Оценка будет производиться для случая коммутации сообщений и перестановок двух видов: регулярных и случайных. Существует множество вариантов регулярных перестановок, и довольно трудно аналитически или методом имитационного моделирования определить число тактов, необходимое для реализации некоторой тестовой группы перестановок. Поэтому в качестве теста принята единственная, но очень широко используемая в процессорных матрицах операция — сдвиг на некоторое число разрядов. Минималь ая дистанция сдвига равна единице, а максимальная совпадает с величиной максимального межпроцессорного расстояния *D*, в связи с чем за среднюю величину сдвига принято *D*/2 (см. табл.8.1). Конечно, это только качественная оценка скорости выполнения регулярных операций коммутации.

Таблица 8.1 Основные характеристики коммутаторов различных типов

Тип коммутатора	Время реализации пакета, такты		Число КЭ
	Регулярные перестановки	Случайный пакет	
Общая шина	N/2	N	$N/2*\log_2 N$
Кольцо (ранг 1)	*N*/4	*N*/2	*N*
Матрица (ранг 2)	$\sqrt{N}/2$	$\sqrt{N}$	*N*
Куб (ранг 3)	$3\sqrt[3]{N}/4$	$\sqrt[3]{N}$	*N*
Омега-сеть	1	$\log_2 N$	$N/2*\log_2 N$
Полный координатный переключатель	1	1	N^2

Для оценки времени реализации пакетов случайных связей обычно проводят имитационное моделирование работы коммутатора при подаче на вход наборов, распределенных по равномерному закону. При этом конфликты разрешаются последовательно в соответствии с некоторым приоритетом. Результаты такого моделирования и число КЭ в коммутаторе каждого типа приведены в табл.8.1.

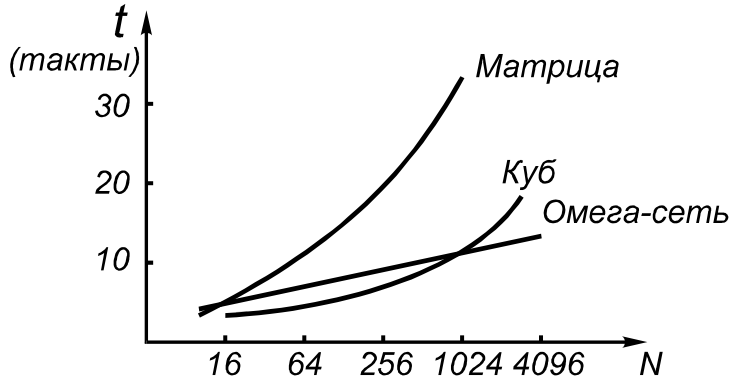


Рис.8.13.Время реализации случайного пакета связей для различных комутаторов

Все вышесказанное позволяет сделать следующие выводы:

- коммутаторы типа полного координатного переключателя не могут использоваться для больших N из-за значительного объема оборудования, а общая шина — из-за большого времени реализации сообщения;
- порядок быстродействия сред на регулярных и нерегулярных пакетах связей одинаков и равен $\sqrt[n]{N}$, где n — ранг среды, а для каскадных коммутаторов подчиняется логарифмическому закону;
- омега-сеть превосходит в большинстве случаев среды на регулярных пакетах. Для качественного сравнения сред и многокаскадных коммутаторов построен график (рис.8.13), из которого следует, что выбор того или иного типа коммутатора должен производиться строго в зависимости от типа ЭВМ, класса решаемых задач (виды перестановок) и размера ПП.

Рассмотрим некоторые вопросы управления в коммутационных сетях. Для примера разберем управление в омега-сети для двух случаев: реализации регулярных перестановок и пакетов случайных связей.

Коммутатор для реализации регулярных перестановок (рис.8.14,а) состоит из коммутационного поля и блока дешифрации и управления (ДУ). Выходы блока ДУ с номерами 0...11 подключены к соответствующим адресным входам коммутационных элементов (чтобы не загромождать рисунок, эти линии не показаны).

Если при реализации некоторой перестановки блок ДУ обнаруживает, что в некоторых КЭ возможны конфликты, блок ДУ вырабатывает два или более КСН, которые реализуются в коммутационном поле последовательно.

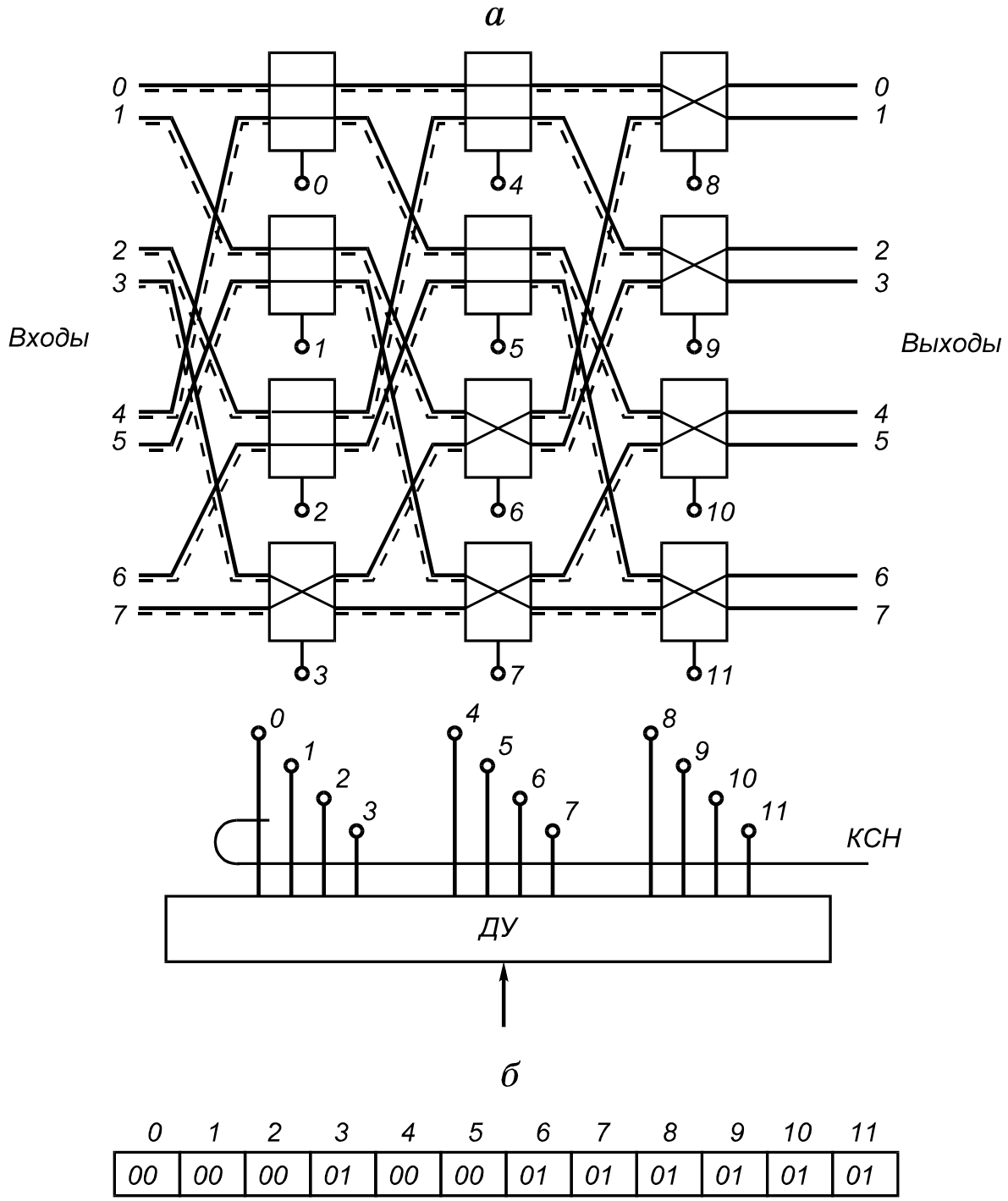


Рис. 8.14. Структура коммутатора для выполнения регулярных перестановок:
а — схема коммутатора; *б* — кодовое слово настройки, использованной в примере перестановки

При выполнении команды коммутации код коммутации подается на вход блока ДУ. Размерность кодового номера перестановки невелика, зависит от числа реализуемых перестановок m и равна $\log_2 m$. Блок ДУ на основе кодового номера перестановки вырабатывает кодовое слово настройки (КСН), размер которого равен $(N / p)C_p^p \log_p N$, где N — число процессоров; p — размерность КЭ; N/p — число КЭ в одном каскаде; C_p^p — число перестановок в КЭ размерностью $p \times p$; $\log_p N$ — число каскадов. Например, КСН для коммутатора на 128 ПЭ при $p = 2$ содержит 1792 разряда, что сильно затрудняет создание коммутаторов с большой размерностью. Кодовые слова

настройки могут храниться в памяти блока ДУ либо вычисляться при выполнении команды коммутации. В первом случае требуется большой объем памяти, во втором время вычисления будет слишком велико. Все КЭ (см. рис.8.14,а) замкнуты в соответствии с КСН (см. рис.8.14,б), которое обеспечивает смещение данных на один ПЭ.

После настройки коммутационного поля с помощью блока ДУ производится передача данных за один такт. Такой вид функционирования соответствует коммутации каналов.

Как правило, вид перестановок заранее не известен и генерируется в процессе решения задачи. Так происходит, например, при определении кратчайшего пути в графе, если каждая вершина графа расположена в отдельном ПЭ и представлена в виде списка вершин. Тогда выборка очередного слоя из списка связей (по одной из каждого процессора) приводит к образованию на входе коммутатора случайно распределенного пакета связей.

В этом случае каждая единица информации, передаваемая любым процессором, состоит из двух частей: А, И. Здесь А — адрес (номер) ПЭ, куда надо передать информацию И. Адрес может передаваться по отдельным шинам (штриховые линии на рис. 8.14, а) или по одним и тем же шинам с разделением во времени: сначала адрес, затем информация. В каждом ПЭ принятый адрес анализируется, чтобы определить на какой выход КЭ должна быть дальше передана информация. Таким образом, в КЭ дешифратор должен дополнительно выполнять функцию анализа принятого адреса. Кроме того, для случайных пакетов конфликты являются обычной ситуацией, поэтому коммутационный элемент должен иметь память для хранения принятой информации, так как неизвестно, будет принятая информация передана в настоящем такте или послана с задержкой. В таком коммутаторе некоторые связи из случайного пакета могут быть реализованы раньше, чем другие, хотя процесс коммутации на входе был начат для всех связей одновременно.

Описанный способ функционирования соответствует коммутации сообщений. Здесь управление является децентрализованным и распределено по коммутационным элементам.

Лекция 9. Процессорные матрицы

Понятие “процессорная матрица” подчеркивает тот факт, что параллельные операции выполняются группой одинаковых ПЭ, объединенных коммутационной сетью и управляемых единым ЦУУ, реализующим единственную программу. Следовательно, ПМ являются представителями класса ОКМД-ЭВМ. Процессорная матрица может быть присоединена в качестве сопроцессора к БМ. В данном случае БМ обеспечивает ввод-вывод данных, управление графиком работы ПМ, трансляцию, редактирование программ и некоторые другие вспомогательные операции.

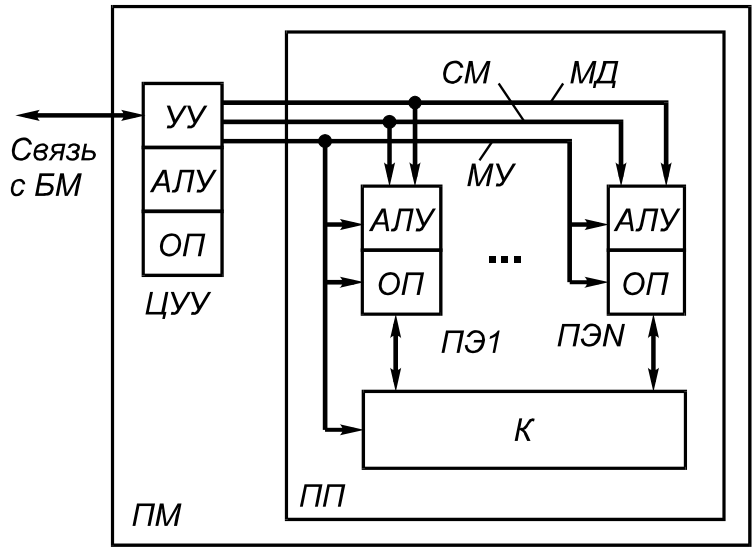


Рис.9.1 Схема ПМ

Каждый ПЭ имеет собственную (локальную) память. Известно, что более половины обращений за информацией приходится на долю локальной памяти, поэтому в таких системах к быстродействию коммутатора предъявляются относительно низкие требования. Схема ПМ с локальной памятью приведена на рис.9.1. Центральное устройство управления является полноценным процессором для выполнения единственной программы, однако в этой программе используются команды двух типов: скалярные и векторные. Скалярные команды начинаются и заканчиваются в ЦУУ, векторные команды начинаются в ЦУУ, а продолжаются в ПП.

Процессорное поле содержит ряд ПЭ, каждый из которых является частью полного процессора, т. е. содержит АЛУ и память, но функции УУ в ПЭ весьма ограничены. В большинстве случаев управление заключается в том, что отдельный ПЭ может выключаться в связи с требованиями решаемой задачи на время выполнения одной или нескольких команд. Число ПЭ колеблется от нескольких десятков (в этом случае процессоры делаются мощными) до нескольких сотен и даже тысяч штук. Центральное устройство управления связано с ПП тремя магистралями:

- по магистрали данных МД из ЦУУ в ПП передаются общие адреса и данные, а из отдельных ПЭ в ЦУУ проводится выборка необходимых единиц информации;
- по системной магистрали СМ, которая имеет одну шину на каждый ПЭ, в ПП передается (или из ПП считывается) вектор активности процессорных элементов;
- по магистрали управления МУ из ЦУУ всем ПЭ передаются одни и те же микрокоманды для одновременного исполнения.

Для большинства ЭВМ магистраль МД имеет 32...64 разряда, а магистраль МУ - 30...80 разрядов; разрядность магистрали СМ равна числу ПЭ.

Как в ЦУУ, так и в ПЭ имеются регистры РОН, поэтому форматы и типы скалярных и векторных команд такие же, как и в последовательных ЭВМ: регистр - регистр (RR), регистр - память (RX) и др.

Векторные команды можно разделить на пять основных групп: арифметико-логические команды; команды пересылки между ЦУУ и ПП; команды коммутации; команды групповых условных переходов; команды активации процессорных элементов.

Рассмотрим функционирование ПП при выполнении команд этих групп.
Пусть имеется следующий оператор:

$$Z(*) = A1*A2(*),$$

которому соответствует следующий фрагмент программы:

```
1 S  ЧТ     5, A1
2 V  ПС     5, 3
3 V  ЧТ     2, A2
4 V  УМН 2, 3
5 V   ЗП     Z,2
```

Здесь S и V — скалярная и векторная команды соответственно.

По команде 1 в ЦУУ из локальной ОП (адрес A1) считывается константа Z1 и записывается в регистр 5. По команде 2 константа *Z*1 из регистра 5 ЦУУ передается в регистр 3 всех ПЭ по МД. По команде 3 все ПЭ считывают из локальной памяти (адрес A 2) константу Z 2 и помещают в регистр 2. Наконец, по команде 4 все ПЭ умножают содержимое регистров 2 и 3 и результат помещают в регистр 2, т.е. выполняется операция умножения на константу.

В ПМ *индивидуальное управление* каждым ПЭ зависит от выключения этих ПВ на период выполнения одной (нескольких) задачи или команды в зависимости от исходных данных, внутреннего состояния ПЭ и некоторых внешних условий. Таким включением и выключением занимается двухуровневая система активации.

Первый уровень активации — реконфигурация — позволяет отключать ненужные или неисправные ПЭ на период выполнения нескольких задач и дает возможность проводить вычисления при наличии неисправных ПЭ. Триггер конфигурации имеется в каждом ПЭ. После выполнения тестовых программ процессорного поля в триггеры конфигурации исправных ПЭ из ЦУУ по СМ заносятся единицы, а в триггеры неисправных ПЭ — нули.

Процессорные элементы с нулевым значением триггера конфигурации не участвуют в вычислениях: в них не выполняются команды, поступающие из ЦУУ, становится невозможным считывание и запись в ОП таких ПЭ, состояние их триггеров активации не влияет на команды векторного условного перехода.

Кроме триггера конфигурации в каждом ПЭ есть несколько (до 16) триггеров активации. Триггер основной активности A0 управляет активностью ПЭ: если A0 находится в состоянии 1, то данный ПЭ активен, если в состоянии 0 — пассивен. Вспомогательные триггеры A1...A7 запоминают активность ПЭ перед различными внутренними циклами, если эта активность меняется; после выхода из цикла нужно восстановить исходное значение активности. Триггеры A8...A15 отражают результат вычислений, проводимых в ПЭ: A8 — равенство результата нулю; A9 — знак результата; A10 — перенос; A11 — арифметическое переполнение; A12 — исчезновение порядка; A13 — переполнение порядка; A14 — некорректное деление; A15 — служебный триггер.

Активация бывает внешней и внутренней. В первом случае состояние триггеров активности каждого ПЭ устанавливается со стороны ЦУУ по СМ с помощью команд вида V ВША Ri, Aj. Здесь код ВША означает, что выполняется команда внешней активации, по которой содержимое 16 разрядов регистра Ri ЦУУ поразрядно записывается во всех ПЭ в регистры активации с номером j. Если j = 0, то это означает, что производится загрузка триггеров основной активности, которая может привести к выключению некоторых ПЭ, когда в соответствующих разрядах находятся нули.

Команда V ЧТА Ri, Aj выполняет обратную операцию: считывание состояния триггеров активации в разряды регистра Ri ЦУУ.

Коммутатор в ПП (см. рис. 2.32) может управляться командами двух типов:

V КМ1 Ri, Rj, Rk, F
V КМ2 Ri, Пk, F

По команде КМ1 каждый передающий процессор посылает содержимое своего регистра Ri в регистр Rk процессора-приемника, номер которого указан в регистре Rj процессора-передатчика. При выполнении команды содержимое Ri и Rj передается в коммутатор, прокладывающий по адресу из Rj нужный маршрут и затем передающий информацию из Ri. Детальный алгоритм выполнения команды коммутации зависит от типа используемого коммутатора. Функция F в команде КМ1 указывает на разновидность команды: запись или считывание данных. В случае считывания требуется обеспечить еще и обратную передачу информации.

Здесь описана только передача информации между регистрами. Если требуется произвести обмен данными, хранящимися в локальной памяти, то предварительно необходимые данные должны быть с помощью команд обращения в память приняты в нужные регистры.

Команды коммутации с адресным вектором используются там, где заранее не известен вид перестановки, т. е. при обработке сложных структур данных (графов, деревьев, таблиц и т. д.).

Однако не всякий коммутатор пригоден для эффективной реализации команд типа КМ1. Такой коммутатор должен допускать работу в асинхронном режиме, внутренними средствами разрешать конфликты, если они возникают. Время передачи пакета случайных связей должно находиться в допустимых пределах. Назовем подобный коммутатор универсальным (УК). Для небольших размеров ПП ($N = 16...32$) в качестве УК может использоваться полный координатный переключатель, а для $N > 32$ должны использоваться многокаскадные коммутаторы.

В УК изменение конфигурации путем разделения ПП на независимые части или по причине неисправностей процессоров не вызывает трудностей, так как для продолжения решения задачи необходимо только переустановить адресные таблицы. Это обеспечивает повышенную живучесть ПП. В машинах с реконфигурацией число ПЭ задается как параметр. Это означает, что в библиотеках стандартных программ число процессоров не определено и конкретизируется в процессе трансляции пользователем или ОС. В синхронных же сдвиговых коммутаторах типа двумерной среды в ILLIAC-IV выход из строя одного узла разрушает всю систему сдвигов.

При выполнении команд типа КМ1 возможна ситуация, когда несколько ПЭ обращаются к одному, т.е. производится сбор информации (коллекция). В коллекционных командах коммутатор должен самостоятельно обеспечить последовательную запись или выдачу требуемых данных.

Команда типа КМ2 означает, что на коммутатор из ЦУУ через МД передается код перестановки Пk, в соответствии с которым настраивается коммутатор и производится бесконфликтная передача информации из регистров Ri или считывание через коммутатор в регистры Ri. Эффективность вычислений в первую очередь зависит от затрат времени на коммутацию и выборку информации. Время выполнения команды коммутации сравнимо со временем выполнения простых команд (логические команды, целочисленное сложение и т.д.), поэтому в большинстве случаев не коммутация, а *размещение данных* влияет на время выполнения программы.

Рассмотрим два варианта размещения квадратной матрицы в ПП (рис.9.2). Предположим, что для решения некоторой задачи, например, умножения матриц, требуется параллельная выборка как строк, так и столбцов. Стандартное, принятое для большинства языков и ЭВМ размещение “по столбцам” (см. рис.9.2,а) позволяет выбирать в АЛУ из ОП параллельно только строки, а элементы столбца расположены в одной ОП*i* и могут выбираться только последовательно. В результате при таком размещении параллелизм процессорного поля не будет использован в полной мере, поэтому для подобных задач характерно специальное размещение информации. При “скошенном” размещении матрицы (см. рис.9.2,б) за один такт могут быть выбраны как строка, так и столбец. Но при выборе столбца должна использоваться “фигурная” выборка, когда каждый ПЭi обращается в ОПi по собственному адресу, вычисляемому в зависимости от номера ПЭ.

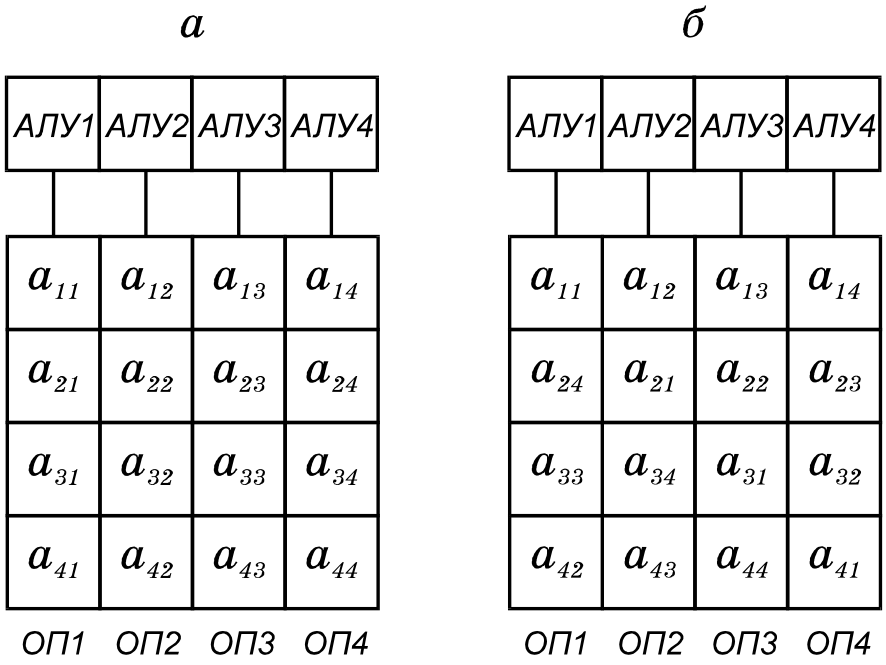


Рис.9.2. Размещение матрицы в памяти процессорного поля

Размещение информации влияет и на методы программирования. Рассмотрим операцию сложения двух векторов (рис. 9.3) для $L = N$ и $L > N$, где L — длина вектора; N — число процессоров. Программы будут выглядеть различно (рис.10.3):

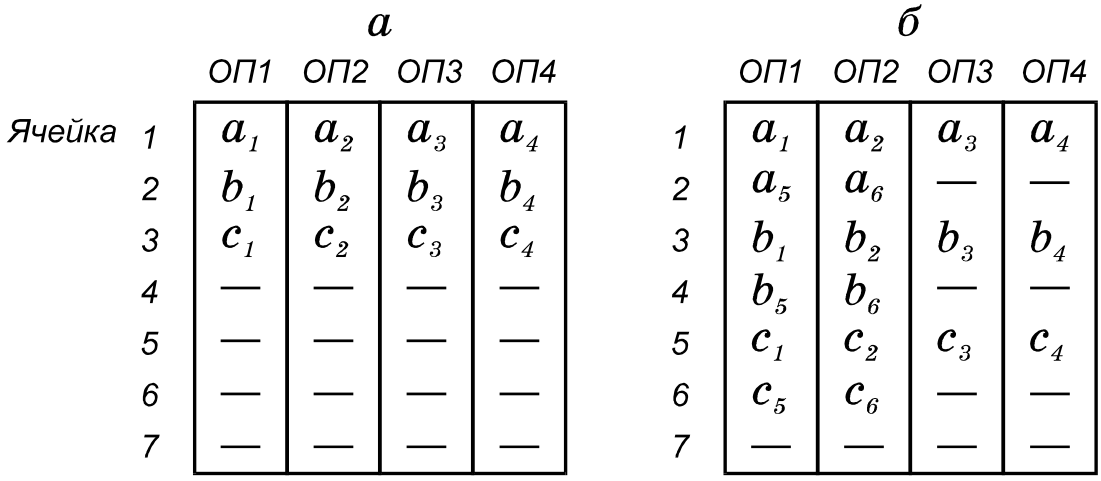


Рис.9.3. Многослойное размещение коротких (а) и длинных (б) векторов

а) V ЧТ 1, $Я_1$ (считывание среза ячеек $Я_1$ памяти в регистры 1)
V СЛ 1, $Я_2$ (сложение содержимого регистров 1 со срезом ячеек $Я_2$ памяти)
V ЗП 1, $Я_2$ (запись содержимого регистров 1 в срез ячеек $Я_3$ памяти)

б) V ЧТ 1, $Я_1$
V СЛ 1, $Я_3$
V ЗП 1, $Я_5$

V МАСК 5, А0 (из регистра 5 ЦУУ передана маска кода активации
1100, запрещающая обращение к ОП3, ОП4)
V ЧТ 1, $Я_2$
V СЛ 1, $Я_4$
V ЗП 1, $Я_6$

Таким образом, несоответствие размеров *L* и *N* меняет не только длину, но и состав операторов программы. На программах на параллельных ЯВУ это не отражается: обе программы будут представлены в виде одного и того же оператора

$$C(*) = A(*) + B(*)$$

Поскольку наиболее эффективные программы могут быть написаны только на ассемблере, то в языках подобного уровня следует предусматривать средства, позволяющие вне зависимости от соотношения *L* и *N* получать одинаковые программы. Так как это не всегда возможно, то из-за необходимости учитывать расположение данных программирование на ассемблере для ПМ становится намного сложнее, чем для последовательных ЭВМ.

Некоторые операции в ПМ программируются проще и выполняются быстрее, чем в параллельных ЭВМ других типов. Например, операция сборки

$$V = V0\ (INDEX\ (i)),\ i = 1, 2, ..., n$$

состоит в том, что из вектора V0 выбираются числа в порядке, указанном в целочисленном массиве INDEX(i) и помещаются в вектор V1. В конвейерной ЭВМ данная операция выполняется в скалярной части (и поэтому медленно). В ПМ операция сборки отображается в одну команду коммутации согласно :

КМ1 Ri, Rj, Rk, F

Здесь в регистре Ri хранится вектор V0, в регистре Rj — массив INDEX(i), в регистр Rk будет записан вектор V1. Поскольку все операции в команде КМ1 выполняются параллельно, то время выполнения этой команды при $L = N$ будет соответствовать одному циклу работы коммутатора.

Для многих параллельных ЭВМ принципиальное значение имеет команда SUM V, т. е. операция суммирования элементов некоторого вектора V. В процессорных матрицах с УК она выполняется в течение нескольких микротактов, которые строятся на основе команды КМ1. В каждом микротакте реализуется следующая операция (работают все ПЭi):

V СЛ Ri, Rj

Здесь операция СЛ выполняется в каждом ПЭ*l* и производит сложение двух чисел, одно из которых расположено в регистре Ri данного ПЭk, а другое — в регистре Ri, но в ПЭ*l*. В регистре *R*j процессора ПЭk указан номер процессора ПЭl. Если в регистре Rj процессора ПЭk записан нуль, то этот процессор не производит суммирование, а выборка операнда из него через коммутатор не запрещается.

Операция SUM V для $N = 8$ (рис.9.4) выполняется за три такта $(\log_2 8 = 3)$ и для каждого такта по вертикали указано содержимое адресного регистра Rj для любого ПЭ.

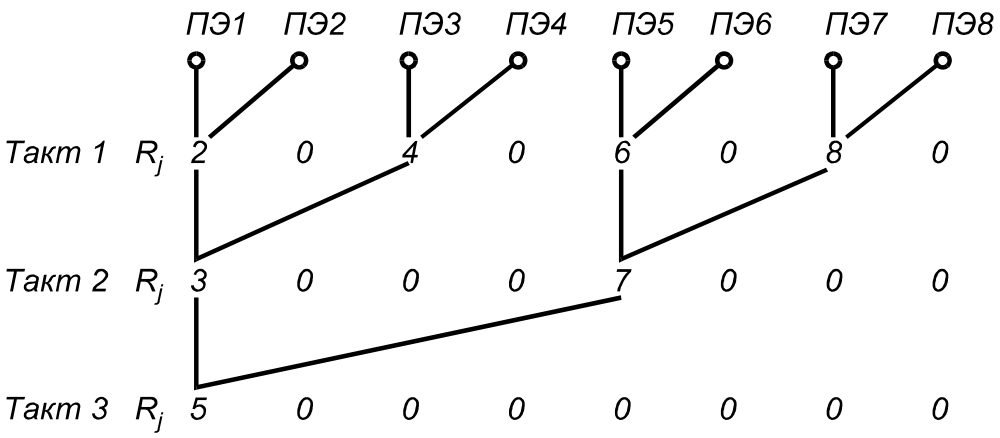


Рис.9.4. Выполнение операции SUM V в ПМ

Эту операцию можно представить следующей программой:

DO 1 N=1, LOG_2 (L)
Маскирование части процессорных элементов
V KM1 R_i, R_j, R_k, F
V СЛ R_s, R_k
CONTINUE

Здесь R_s – накапливающий регистр процессорного элемента.

Наиболее известным представителем процессорных матриц является ЭВМ ILLIAC-IV.

Лекция 10. Параллельные системы с разделяемой памятью (SMP)

10.1. Типы многопроцессорных систем.

В основе МКМД-ЭВМ лежит традиционная последовательная организация программы, расширенная добавлением специальных средств для указания независимых, параллельно исполняемых программных фрагментов. Такими средствами могут быть операторы FORK и JOIN, скобки parbegin ... parend, оператор DO FOR ALL и др. МКМД-ЭВМ имеет две разновидности: ЭВМ с общей и индивидуальной памятью. Структура этих ЭВМ представлена на рис.10.1.

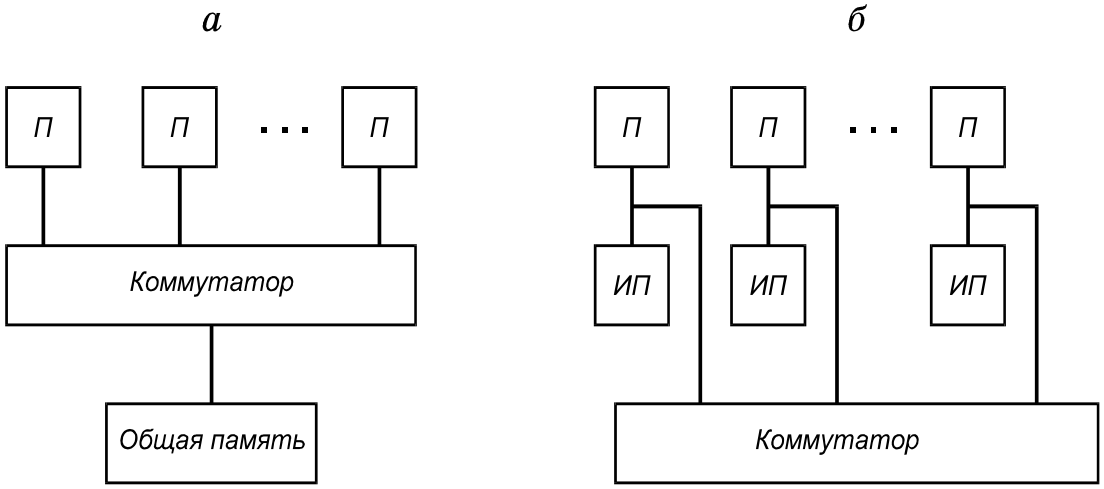


Рис.10.1. Структура ЭВМ с разделяемой (*а*) и индивидуальной (*б*) памятью. Здесь: П – процессор, ИП - индивидуальная память.

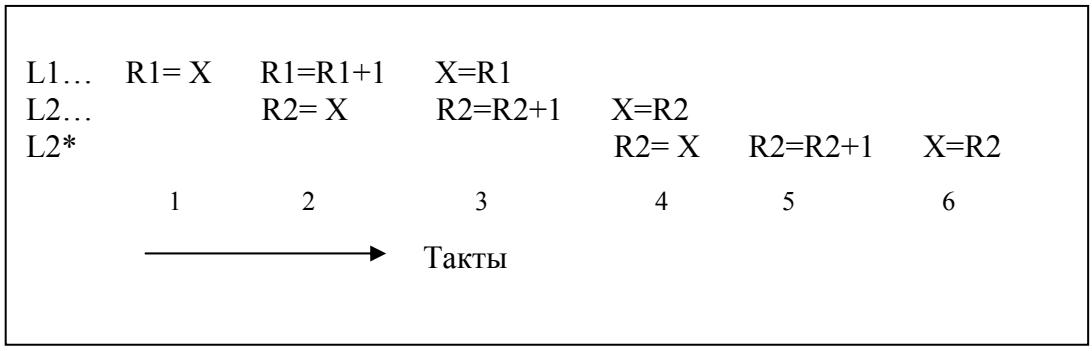
Главное различие между МКМД-ЭВМ с разделяемой и индивидуальной (локальной, распределенной) памятью состоит в характере адресной системы. В машинах с разделяемой памятью адресное пространство всех процессоров является единым, следовательно, если в программах нескольких процессоров встречается одна и та же переменная *X*, то эти процессоры будут обращаться в одну и ту же физическую ячейку общей памяти. Это вызывает как положительные, так и отрицательные последствия:

- Наличие разделяемой памяти не требует физического перемещения данных между взаимодействующими программами, которые параллельно выполняются в разных процессорах. Это упрощает программирование и исключает затраты времени на межпроцессорный обмен.
- Поскольку при выполнении команд каждым процессором необходимо обращаться в разделяемую память, то требования к пропускной способности коммутатора этой памяти чрезвычайно высоки, что и ограничивает число процессоров в системах с общей памятью величиной 10...20.
- Несколько процессоров могут одновременно обращаться к общим данным и это может привести к получению неверных результатов. Чтобы исключить такие ситуации, необходимо ввести систему синхронизации параллельных процессов, что усложняет механизмы операционной системы. Это показано на рисунке ниже.

Пусть два процесса (процессора) L1 и L2 выполняют операцию прибавления 1 в ячейку X, причем, во времени эти операции выполняются независимо:

L1 =….X: = X + 1;……..
L2 =….X: = X + 1;……..

Такие вычисления могут соответствовать, например, работе сети по продаже билетов, когда два терминала сообщают в центральный процессор о продаже одного билета каждый. На центральном процессоре выполнение каждой операции заключается в следующем: чтение сожержимого X в регистр R1, прибавление единицы, запись содержимого R1 в ячейку X. Пусть во времени на центральном процессоре операции по тактам расположились следующим образом и начальное значение X = 0.



В результате неудачного размещения в такте 2 из ячейки X читается значение 0 до того, как процесс L1 записал туда единицу. Это приводит к тому, что в такте 4 в ячейку X будет вместо двух записана единица. Чтобы избежать таких ситуаций, нужно запрещать всем процессам использовать общий ресурс (ячейка X), пока текущий процесс не закончит его использование. Это называется синхронизацией. Такая ситуация показана в строке L2*.

МКМД-ЭВМ с разделяемой памятью называют симметричными мультипроцессорами СМП (SMP – Symmetrical Multiprocessor).

В системах с индивидуальной памятью каждый процессор имеет независимое адресное пространство и наличие одной и той же переменной *X* в программах разных процессоров приводит к обращению в физически разные ячейки индивидуальной памяти этих процессоров. Это вызывает физическое перемещение данных между взаимодействующими программами в разных процессорах, однако, поскольку основная часть обращений производится каждым процессором в собственную память, то требования к коммутатору ослабляются и число процессоров в системах с распределенной памятью и коммутатором типа гиперкуб может достигать нескольких сотен и даже тысяч.

В микропроцессорной технике большое распространение получили МКМД-ЭВМ с индивидуальной памятью ввиду относительной простоты их архитектуры. Наиболее последовательно идея ЭВМ с индивидуальной памятью отражена в транспьютерах и транспьютероподобных микропроцессорах.

10.2. Программирование для SMP (Open MP)

Интерфейс OpenMP является стандартом для программирования на масштабируемых SMP-системах с разделяемой памятью. В стандарт OpenMP входят описания набора директив компилятора, переменных среды и процедур. За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы.

Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки, текст которых приведен в спецификациях. В OpenMP любой процесс состоит из нескольких *нитей управления*,

которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. В простейшем случае, процесс состоит из одной нити.

Обычно для демонстрации параллельных вычислений используют простую программу вычисления числа π. Рассмотрим, как можно написать такую программу в OpenMP. Число π можно определить следующим образом:

$$\int_0^1 \frac{1}{1+x^2} dx = \text{arctg}(1)\text{-arctg}(0)= \pi/4.$$

Вычисление интеграла затем заменяют вычислением суммы :

$$\int_0^1 \frac{4}{1+x^2} dx = \frac{1}{n} \sum_{i=1}^{n} \frac{4}{1+x_i^2},$$

где: $x_i = \frac{1}{n} \cdot \left(i - \frac{1}{2} \right)$

В последовательную программу вставлены две строки (директивы), и она становится параллельной.

```
program compute_pi
  parameter (n = 1000)
  integer i
  double precision w,x,sum,pi,f,a
  f(a) = 4.d0/(1.d0+a*a)
  w = 1.0d0/n
  sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+:sum)
  do i=1,n
     x = w*(i-0.5d0)
     sum = sum + f(x)
  enddo
  pi = w*sum
  print *,'pi = ',pi
  stop
  end
```

Программа начинается как единственный процесс на головном процессоре. Он исполняет все операторы вплоть до первой конструкции типа PARALLEL. В рассматриваемом примере это оператор PARALLEL DO, при исполнении которого порождается множество процессов с соответствующим каждому процессу окружением. В рассматриваемом примере окружение состоит из локальной (PRIVATE) переменной x, переменной sum редукции (REDUCTION) и одной разделяемой (SHARED) переменной w. Переменные x и sum локальны в каждом процессе без разделения между несколькими процессами. Переменная w располагается в головном процессе. Оператор редукции REDUCTION имеет в качестве атрибута операцию, которая применяется к локальным копиям параллельных процессов в конце каждого процесса для вычисления значения переменной в головном процессе. Переменная цикла i является локальной в каждом процессе, так как именно с уникальным значением этой переменной порождается каждый процесс. Параллельные процессы завершаются оператором END DO, выступающим как синхронизирующий барьер для порожденных процессов. После завершения всех процессов продолжается только головной процесс.

Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов "!$OMP", поэтому приведенная выше программа может без изменений выполняться на последовательной ЭВМ в обычном режиме.

Рассматриваемый ниже многоядерный кристалл Cell включает несколько уровней параллелизма, однако, по верхнему уровню Cell можно отнести к классу машин с параллелизмом типа SMP. Эти и объясняется размещение Cell в этой лекции, хотя его можно и оспорить.

10.3. Многоядерность, CELL

(по тексту статьи «Процессор: шаг в будущее», автор: Дмитрий Мороз, 01.03.2005)

Философия Cell. Принципы, заложенные в архитектуру нового чипа, были разработаны в начале 2000 года инженерами IBM. Идея массового параллелизма, на основе которой работает Cell, была заложена в так называемую ("cellular architecture") "клеточную архитектуру", в которой для создания суперкомпьютеров используется множество однотипных процессоров (от 10 тыс. до 1 миллиона), каждый из которых оснащён собственным контроллером RAM и определённым объёмом самой оперативной памяти.

Cell может работать не только в качестве, собственно, процессора, но и в качестве элемента большой системы. Путём объединения различной техники, содержащей чипы Cell, можно построить единую сеть, которая будет функционировать, как одно "устройство". Единая глобальная сеть. Разве не об этом долгие годы мечтают писатели-фантасты? С приходом Cell мечты могут стать реальностью. Но мы отложим их на потом и перейдём к теме статьи - процессору Cell.

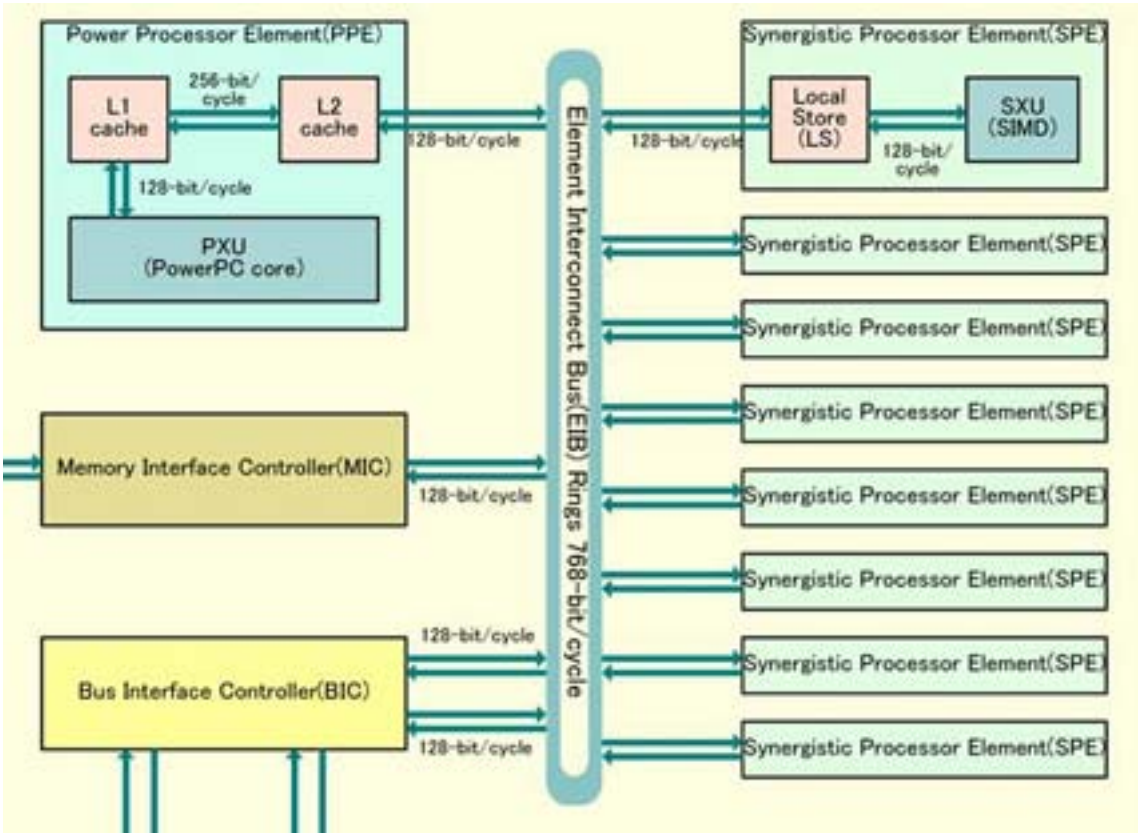


Рис.10.2. Структура процессора CELL

Как видно на рис.11.2, архитектура Cell состоит из следующих компонентов:

- Процессорного Элемента на основе POWER-архитектуры (PPE - POWER Processor Element);

- Восьми Синергических Процессорных Элементов (SPE - Synergistic Processor Element), ранее называвшихся Присоединяемыми Процессорными Устройствами (APU - Attached Processing Unit);
- Шины Взаимосвязываемых Элементов (EIB - Element Interconnect Bus);
- Контроллера Интерфейса Памяти (MIC - Memory Interface Controller);
- Контроллера Интерфейса Шины ввода/вывода (BIC - Bus Interface Controller).

Процессорный элемент на основе POWER-архитектуры. К сожалению, на ISSCC 2005 о PPE рассказывали меньше всего, поэтому такие важные характеристики процессорного ядра, как его архитектура и производительность остались "за кадром". Тем не менее, кое-какие данные нам уже известны.

PPE представляет собой два 64-разрядных процессорных ядра (на основе POWER-архитектуры) с поочерёдным выполнением команд (in-order execution), в отличие от внеочередного выполнения (out-of-order execution), присущего всем современным процессорам. PPE поддерживает технологию одновременной обработки двух потоков (SMT - Simultaneous Multi-Threading), примерным аналогом которой является Hyper-Threading в Pentium 4. В PPE присутствует блок VMX (VMX - Vector Multimedia eXtensions, более известный как AltiVec). Объём кэша L1 составляет 64 Kb (по 32 Kb на кэш инструкций и данных), L2-кэша - 512 Kb. Несмотря на сравнительно небольшой 11-стадийный конвейер PPE, представители STI заверяют, что запас по наращиванию частоты у Cell очень большой.

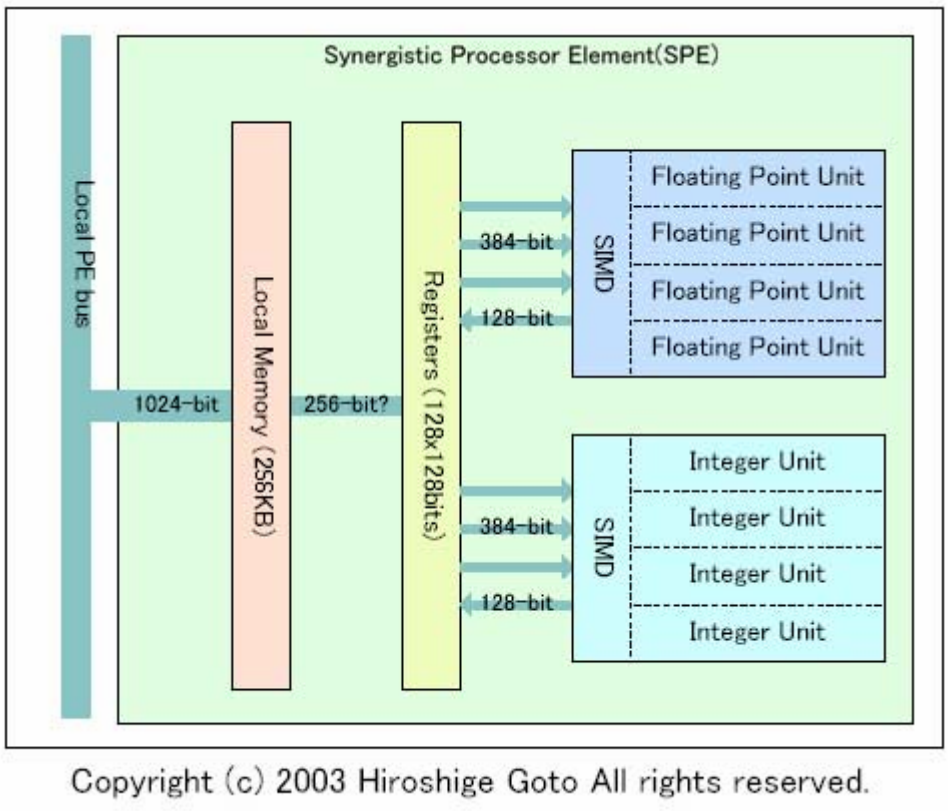


Рис.10.3. Синергический процессорный элемент

Синергический процессорный элемент (рис.10.3.) представляет собой специализированный векторный процессор, обрабатывающий SIMD-инструкции (аналог SSE в Pentium и 3Dnow! в Athlon), то есть технология MMX). Архитектура SPE довольно проста: четыре блока для работы с целочисленными векторными операциями и четыре блока для работы с числами с плавающей запятой. Правда, при этом большинство арифметических инструкций представляют собой 128-разрядные векторы, разделённые на четыре 32-битных элемента. Каждый SPE оснащён 128 реги-

страми, разрядность которых - 128-бит. Вместо кэша L1 процессор содержит 256 Kb собственной "локальной памяти" (local memory, также называемой local store) разделённой на четыре отдельных сегмента по 64 Кбайт каждый, а также DMA-контроллер. Последний предназначен для обмена данными между основной памятью (RAM) и локальной (LM/LS), минуя PPE.

Локальная память, по сути, выполняет роль кэша L1, но при этом полностью контролируется программистом, вследствие чего организована значительно проще. Достигается это путём переноса логики контроля за переполнением кэша из самого чипа в программный код, что способствует облегчению архитектуры локальной памяти (отсутствует поиск в RAM при каждом обращении к LM, упреждающая выборка и т.д.).

Благодаря наличию динамического механизма защиты памяти доступ к каждому SPE в Cell может быть закрыт, вследствие чего данные, обрабатывающиеся этим процессором, будут недоступны другим (например, в Cell другого устройства).

Несмотря на то, что SPE представляет собой векторный процессор, он не является аналогом VMX/AltiVec. Как уже говорилось, SPE - отдельный микропроцессор, выполняющий собственные команды, а VMX являются блоком (подмножеством команд), выполняемых PowerPC (G4/G5).

Для каких целей могут быть использованы SPE? Ответ на этот вопрос тянет на отдельную статью, так что ограничимся самым важным.

Одной из областей применения Cell аналитики считают рынок цифровых сигнальных процессоров (DSP - Digital Signal Processor), высокая скорость которых обусловлена многопоточным просчётом векторных инструкций. Восемь параллельно работающих SPE позволяют составить достойную конкуренцию специализированным DSP-процессорам. Установка PCI-E-платы с Cell в простой компьютер будет невероятной находкой для музыкантов и учёных.

Чтобы не быть многословными, перечислим возможности использования SPE через запятую: просчёт физических моделей, тесселяция поверхностей высшего порядка в полигональные модели, инверсная кинематика, скелетная анимация, обработка воксельных данных, анимация частиц, компрессия/декомпрессия видеоданных в формате MPEG4 "на лету", просчёт алгоритма ray-tracing "на лету", преобразование аудиоданных из одного формата в другой "на лету", обработка алгоритмов шифрования, искусственного интеллекта… это предложение можно продолжать очень долго, но мы остановимся и перейдём к следующему элементу архитектуры Cell.

Шина Взаимосвязываемых Элементов. Данная шина связывает в единую систему PPE, SPE, а также контроллеры MIC и BIC. Она представляет собой четыре концентрических кольца (шириной 128 бит на кольцо), проходящих через все элементы Cell. Для уменьшения возникающих шумов одна пара "колец" передаёт данные в одном направлении, а вторая - в обратном. Данные, проходящие из одного SPE в другой, используют установленные в них специальные буферы/повторители, обеспечивающие беспрерывное движение данных по шине. Передавая по 96 байт за цикл, EIB способна обрабатывать более 100 уникальных запросов.

Дизайн EIB был создан специально для возможности дальнейшего масштабирования Cell. Длина маршрута сигнала не изменяется в зависимости от количества SPE. Так как данные путешествуют от одного SPE к другому, увеличение их количества приводит лишь к увеличению латентности транспортировки данных от одного элемента Cell к другому.

В текущей версии архитектуры Cell шина EIB организована таким образом, что SPE устанавливаются лишь по горизонтальной оси. Благодаря этому существует возможность легко добавлять/убирать дополнительные SPE, доводя их общее количество до максимально возможного на том или ином производственном тех. процессе (с добавлением SPE увеличивается длина кристалла процессора).

PPE, SPE и EIB представляют собой основу архитектуры Cell, единое трио, работающее вместе. Кратко его и рассмотрим. Сразу напомним, что SPE - отдельный процессор с собственным ОЗУ, пусть и не такой продвинутый, как современные Pentium или PowerPC. Поэтому и выполняет SPE свои собственные инструкции. Синергический процессорный элемент является "подчинённым" POWER-процессора, выполняя те задачи, которые на него возложит PPE. Тем не менее, работают они параллельно.

Вместе с Cell на свет появляется термин "апулет" (APUlet), произошедший от старого названия SPE. Он представляет собой объединённый в единое целое код векторной программы и обрабатываемые ею данные.

При необходимости использования "подчинённого" процессора PPE направляет апулет по EIB в предназначенный для этого SPE, где тот выполняется. Затем обработанный SPE апулет отправляется либо в RAM (если они необходимы PPE), либо в локальную память векторного процессора, из которой они могут быть перенаправлены через EIB в следующий SPE одного и того же Cell-чипа (архитектура Cell позволяет SPE работать конвейерно - один за другим).

Контроллер Интерфейса Памяти. Для непрерывного обеспечения PPE и SPE данными необходима память с очень высокой пропускной способностью. Sony и Toshiba учли способность компании Rambus создавать высокоскоростные интерфейсы и память, приняв решение использовать Direct RDRAM в консоли.

"Эксперимент" удался. И теперь, во время создания Cell, сотрудничество с компанией Rambus продолжается. Во время открытия ISSCC 2005 7 февраля президент компании Дэйв Муринг (Dave Mooring) объявил о лицензировании STI технологий памяти XDR RAM и высокоскоростной шины Flex I/O.

Двухканальная память XDR RAM (Ранее носившая кодовое название Yellowstone), используемая с Cell, обеспечивает суммарную пропускную способность в 25,2 Gb/sec. В подсистеме памяти каждый канал способен обслуживать до 36 чипов памяти, соединённых одними шинами команд (ШК) и данных (ШД). ШД каждого чипа подключается к контроллеру памяти через набор двунаправленных соединений типа "точка-точка". Сигналы передаются по ШК и ШД со скоростью 800 Мбит/с. Скорость передачи интерфейса "точка-точка" составляет 3,2 Гбит/с. При использовании устройств с разрядностью ШД 16 бит каждый канал XDR RAM может обеспечивать максимальную пропускную способность в 102,4 Гбит/с (умножаем 2 канала на 16-битную ШД и на проп. способность интерфейса "точка-точка" 3,2 Гбит/с) или 12,6 Gb/sec. Поэтому контроллер памяти текущей версии процессора Cell, использующего двухканальную память XDR RAM и 4 чипа памяти, обеспечивает пропускную способность в 25,2 Gb/sec.

К сожалению, чипы XDR RAM, в данный момент доступные на рынке, обладают ёмкостью лишь 512 Мбит, так что максимальный объём RAM в системе на базе Cell не может превышать 256 Мбайт. К счастью микросхемы памяти XDR RAM могут быть сконфигурированы таким образом, что 36 чипов будут подключены к одному 36-битному каналу. В такой конфигурации двухканальная XDR RAM может поддерживать до 32 Гбайт памяти (512 Мбит чипы) с коррекцией ошибок (ECC). Да-да, Дэйв Барски (Dave Bursky) из журнала Electronic Design Magazine заявляет, что подсистема XDR RAM использует 72 пары сигналов для ШД, что говорит о поддержке памятью ECC. Конечно, игровой консоли она не пригодится, но вот для рабочих станций и серверов будет, как говорится - "в самый раз".

Контроллер Интерфейса Шины ввода/вывода. Второй технологией, лицензированной STI, является интерфейс высокоскоростной шины Flex I/O (ранее носившей кодовое название Redwood), применённый в BIC. Flex I/O состоит из 12-байтных каналов (разрядностью 8 бит каждый). Шина использует 96 дифференцированных сигнальных пар для достижения пропускной способности в 6,4 Гбайт/с по одному каналу. Конфигурация канала асимметрична: 7 байт отводится на передачу, а 5 байт - на приём. Благодаря этому пропускная способность 12-битной пары составляет 44,8 Гбайт/с на передачу и 32 Гбайт/с на приём. В результате суммарная пропускная способность шины ввода/вывода составляет 76,8 Гбайт/с. Во Flex I/O реализованы две интересные технологии - FlexPhase и DRSL (Differential Rambus Signaling Level, дословно - дифференциальный уровень сигналов Rambus).

Общие сведения о Cell. Разобравшись с архитектурой Cell, поговорим о его характеристиках. Продемонстрированный на ISSCC 2005 прототип процессора был изготовлен по 0,09 мкм тех. процессу (8-слоёв медных соединений) с использованием "кремния-на-изоляторе" (SOI, Silicon-on-Insulator). Количество транзисторов в чипе составляет 234 миллиона, площадь кристалла - 221 мм2 (приблизительно таким же был прототип Emotion Engine).

Каждый SPE в Cell состоит из 21 млн. транзисторов, из которых 14 млн. отводится на локальную память, а 7 млн. - на логику. При этом его размеры составляют 5,81 мм * 2,8 мм (0,09 мкм тех. процесс с применением SOI).

Кристалл Cell размещён внутри BGA-корпуса с размерами 42,5 * 42,5 мм. Из 1236 контактов процессора 506 являются сигнальными, а остальные предназначены для питания и "земли".

На ISSCC 2005 был продемонстрирован прототип, работающий на частоте 4 ГГц. При этом пиковая производительность восьми SPE, по заявлениям инженеров STI, составляет 250 Гигафлоп/с, или 250 миллиардов операций с числами с плавающей запятой (сравните это число с 851 Гигафлоп/с наименее быстрого суперкомпьютера из списка Top500). В лабораториях же STI "трудятся" Cell'ы с частотами от 3,2 до 5,3 ГГц (напряжение питания от 0,09В до 1,3В). Энергопотребление 5,3 ГГц чипа составляет около 180 Вт, 4 ГГц чипа - 80 Вт. При этом стоит учитывать, что эти процессоры - тестовые экземпляры, и изготовлены по "старой" технологии, которая при массовом производстве использоваться не будет.

Так, SCEI заявила, что процессор Cell, изготовленный по тех. процессу 0,065 мкм с использованием технологий SOI и "напряжённого кремния" (strained silicon) будет потреблять 30-40 Вт при частоте 4,6 ГГц.

Начало массового производства Cell должно начаться в третьем квартале этого года. IBM и Sony планируют выпускать вначале по 15 тыс. 30-мм пластин ежемесячно, постепенно увеличивая цифру.

Благодаря большим объёмам производства (десятки и сотни миллионов чипов в год) цена на процессоры Cell должна быть на приемлемом уровне. Желание STI конкурировать с Intel/AMD означает, прежде всего, ценовую войну. Даже после запуска в производство первых чипов Cell разработка архитектуры будет продолжаться. К 2010 году частоту процессоров, благодаря использованию всё более совершенного тех. процесса, а также оптических соединений между транзисторами, планируется поднять до 10 ГГц.

При всей сложности конструкции Cell главным компонентом, определяющим его коммерческий успех, будет программное обеспечение. "Предстоит гигантский труд по созданию ПО, - говорит Доэрти. - Мы уверены, что архитектура чипа появится своевременно, а вот искусство программирования, чтобы заставить этот процессор заработать в полную силу, придется совершенствовать".

Более того, решающим в судьбе Cell станет создание операционной системы и набора приложений, способных использовать мультипроцессорные возможности Cell и средства peer-to-peer вычислений. Понимая это, трио партнеров поставили перед собой задачу создания Cell как системы, параллельно с аппаратурой разрабатывая операционную систему и прикладное ПО. Конструкторы Cell проектируют чип таким образом, чтобы он мог функционировать с самыми разными ОС, включая Linux.

Лекция 11. Кластеры Beowulf

Оглавление

11.1. Реализация МКМД ЭВМ.

Существует несколько вариантов технической реализации МКМД ЭВМ. Эти варианты существенно отличаются по стоимости разработки, производства и обслуживания.

Исторически первыми были многопроцессорные ЭВМ, построенные на базе специализированной аппаратуры и системного математического обеспечения. Их называют MPP ЭВМ (Massively Parallel Processors – процессоры с массовым параллелизмом). Сейчас их выпуск практически прекратился из-за высокой стоимости разработки и производства.

Позже широкое распространение получили кластеры – многопроцессорные ЭВМ, построенные из элементов серийного или мелкосерийного производства. Это относится не только к аппаратуре, но и к программному обеспечению. Основной операционной системой для кластеров является бесплатная ОС Linux.

В свою очередь кластеры можно разделить на две заметно отличающиеся по производительности ветви:

- кластеры типа Beowulf [11], которые строятся на базе обычной локальной сети ПЭВМ. Чтобы получить такой кластер, к сети ПЭВМ нужно добавить только системное программное обеспечение, которое бесплатно распространяется в интернет;
- монолитные кластеры, все оборудование которых компактно размещено в специализированных стойках. Это очень быстрые машины, число процессоров в которых может достигать сотен и тысяч. Процессоры в монолитных кластерах не могут использоваться в персональном режиме.

11.2. Модель OSI

Основой любой пространственно распределенной вычислительной системы является принцип адресации ее узлов. Множество адресов, которые являются допустимыми в рамках некоторой схемы адресации, называется адресным пространством. Адресное пространство может иметь плоскую (линейную) или иерархическую организацию.

В первом случае множество адресов не структурировано, все адреса равноправны. Во втором случае оно организовано в виде вложенных друг в друга подгрупп, которые, последовательно сужая адресную область подобно почтовому адресу (страна, город, улица, дом, квартира), в конце концов, определяют отдельный сетевой интерфейс.

Примером плоского числового адреса является **MAC** адрес (**M**edia **A**ccess **C**ontrol), предназначенный для однозначной идентификации сетевых интерфейсов в локальных сетях. Такой адрес обычно используется только аппаратурой, поэтому его стараются сделать компактным и записывают в виде 6-байтового числа, например, 0081005e24a8. При задании адресов не требуется выполнения ручной работы, так как *они встраиваются в аппаратуру компанией-изготовителем*, поэтому их называют аппаратными. Распределение MAC адресов между изготовителями производится централизованно комитетом IEEE.

Типичным представителем иерархических числовых адресов являются 4-байтовые сетевые **IP** (**I**nternet **P**rotocol) адреса, используемые для адресации узлов в глобальных сетях. В них поддерживается двухуровневая иерархия, адрес делится на старшую часть (номер сети) и младшую часть

(номер узла в сети). Такое деление позволяет передавать сообщение между сетями только на основании номера сети, а номер узла используется после доставки сообщения в нужную сеть.

В современных узлах для адресации применяют обе системы: передача между сетями осуществляется с помощью IP адресов, а внутри сети – с помощью MAC адресов. Кроме того, для удобства пользователя используются и символические имена. Эти системы адресации заложены во все стандарты передачи данных.

Модель OSI (**O**pen **S**ystem **I**nterconnection). Формализованные правила, определяющие последовательность и формат сообщений, которыми обмениваются сетевые компоненты одного уровня различных узлов, называют протоколом. Для обмена используется набор протоколов различного уровня. Эти протоколы должны согласовывать величину, форму электрических сигналов, способ определения размера сообщения, методы контроля достоверности и др.

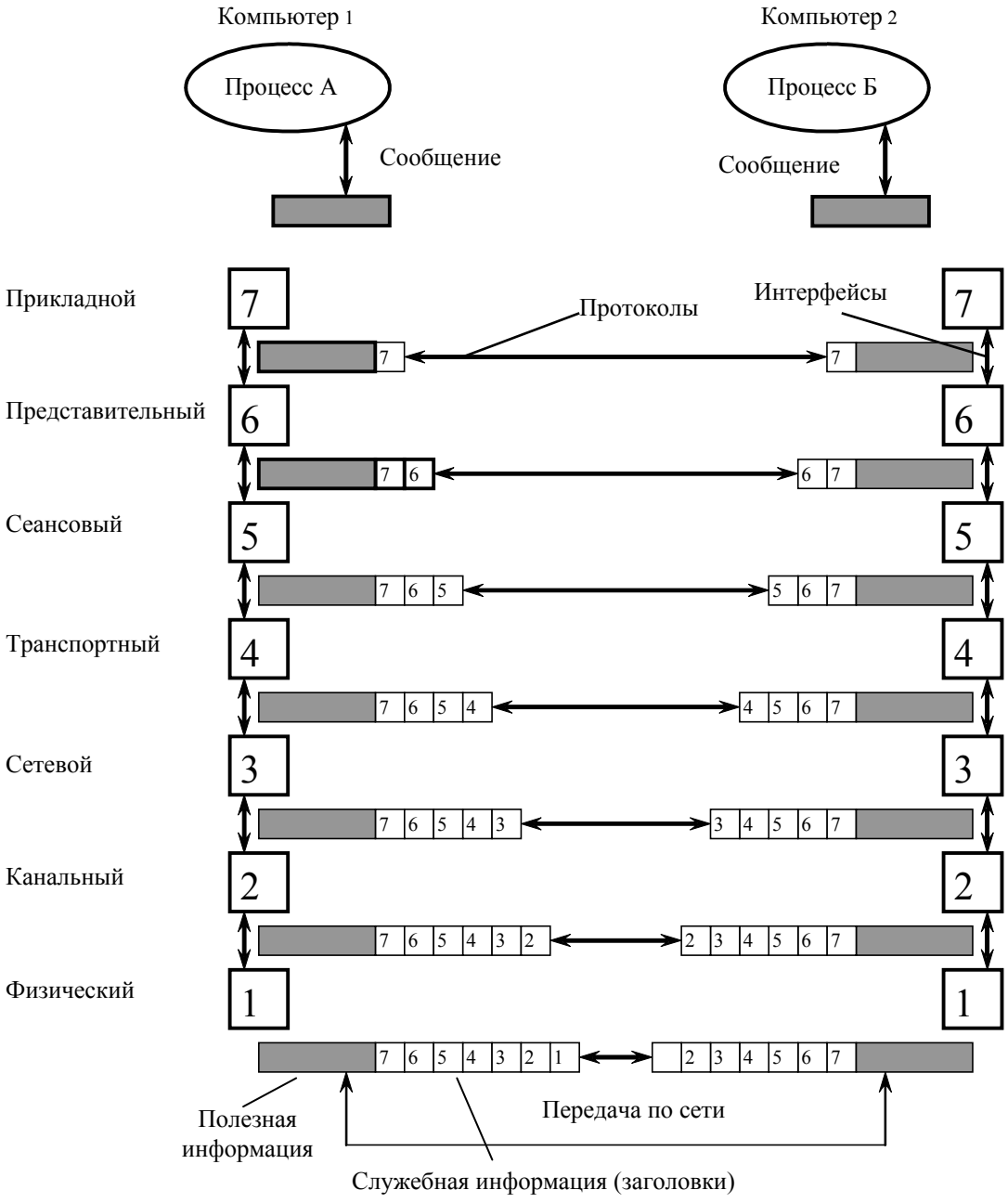


Рис.11.1 Модель взаимодействия открытых систем OSI.

сновным стандартом набора коммуникационных протоколов является модель OSI, разработанная в начале 80-х годов. В модели OSI (рис.11.1) средства взаимодействия делятся на семь уровней.

Рассмотрим работу модели OSI при выполнении обмена между двумя узлами. Пусть приложение обращается с запросом к прикладному уровню, например, к файловой службе. На основании этого запроса программы прикладного уровня формируют сообщение стандартного формата. Обычное сообщение состоит из заголовка и поля данных. Заголовок содержит служебную информацию, которую необходимо передать через сеть прикладному уровню машины-адресата, чтобы сообщить ему, какую работу надо выполнить. В нашем случае заголовок должен содержать информацию о месте нахождения файла и типе операции, которую необходимо над ним выполнить.

После формирования сообщения прикладной уровень направляет его вниз по стеку представительному уровню. Протокол представительного уровня на основании информации, полученной из заголовка прикладного уровня, выполняет требуемые действия и добавляет к сообщению собственную служебную информацию – заголовок представительного уровня, в котором содержатся указания для протокола представительного уровня машины-адресата.

Полученное в результате сообщение передается вниз сеансовому уровню, который в свою очередь добавляет свой заголовок и так далее. Наконец, сообщение достигает нижнего, физического уровня, который собственно и передает его по линии связи машине-адресату.

К этому моменту сообщение обрастает заголовками всех уровней (рис.11.2). Когда сообщение по сети поступает на машину адресат, оно принимается ее физическим уровнем и последовательно перемещается вверх с уровня на уровень. Каждый уровень анализирует и обрабатывает заголовок своего уровня, выполняя соответствующие данному уровню функции, а затем удаляет этот заголовок и передает сообщение вышележащему уровню.

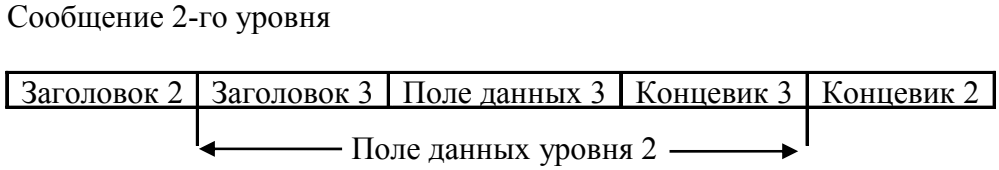


Рис.11.2. Взаимосвязь по информации между смежными уровнями

В модели OSI уровни выполняют различные функции:

1. Физический уровень является самым нижним уровнем. На этом уровне определяются характеристики электрических сигналов, передающих дискретную информацию, например, крутизна фронтов импульсов, уровни напряжения или тока передаваемых сигналов. Кроме того, здесь стандартизуются типы разъемов и назначение каждого контакта.
2. Канальный уровень отвечает за формирование кадров, физическую адресацию, разделение передающей среды, контроль ошибок.
3. Сетевой уровень служит для образования единой транспортной системы, объединяющей несколько сетей.
4. Транспортный уровень обеспечивает прикладному и сеансовому уровням передачу данных с той степенью надежности, которая им требуется.
5. Сеансовый уровень обеспечивает управление взаимодействием: фиксирует, какая из сторон является активной в настоящий момент, предоставляет средства синхронизации.
6. Представительный уровень имеет дело с формой представления передаваемой информации, не меняя при этом ее содержания.

7. Прикладной уровень – это набор разнообразных протоколов, с помощью которых пользователи сети получают доступ к разделяемым ресурсам.

Для построения кластеров обычно используются два нижних уровня – физический и канальный. Остальные уровни для локальной сети часто имеют вырожденный характер.

11.3. Коммуникационные технологии

Технология Ethernet. В локальной сети коммуникационное оборудование появляется только на физическом и канальном уровнях. Это оборудование и определяет локальную сеть. Наиболее распространенной коммуникационной технологией для локальных сетей является технология Ethernet, которая имеет несколько технических реализаций. Технология Ethernet, Fast Ethernet, Gigabit Ethernet обеспечивают скорость передачи данных соответственно 10, 100 и 1000 Мбит/с. Все эти технологии используют один метод доступа – CSMA/CD (carrier-sense-multiply-access with collision detection), одинаковые форматы кадров, работают в полу- и полнодуплексном режимах. Метод предназначен для среды, разделяемой всеми абонентами сети

На рис.11.3 представлен метод доступа CSMA/CD. Чтобы получить возможность передавать кадр, абонент должен убедиться, что среда свободна. Это достигается прослушиванием несущей частоты сигнала. Отсутствие несущей частоты является признаком свободности среды.

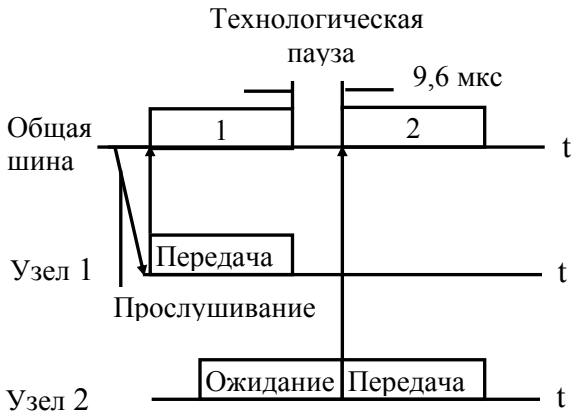


Рис.11.3. Метод случайного доступа CSMA/CD

На рис.11.3 узел 1 обнаружил, что среда сети свободна, и начал передавать свой кадр. В классической сети Ethernet на коаксиальном кабеле сигналы передатчика узла 1 распространяются в обе стороны, так что все узлы сети их получают. Все станции, подключенные к кабелю, могут распознать факт передачи кадра, и та станция, которая узнает свой адрес в заголовке передаваемого кадра, записывает его содержимое в свой внутренний буфер, обрабатывает полученные данные, передает их вверх по своему стеку, а затем посылает по кабелю кадр-ответ. Адрес станции-источника содержится в исходном кадре, поэтому станция-получатель знает, кому послать ответ.

Узел 2 во время передачи кадра узлом 1 также пытался начать передачу своего кадра, однако обнаружил, что среда занята – на ней присутствует несущая частота, - поэтому узел 2 вынужден ждать, пока узел 1 не прекратит передачу кадра.

После передачи кадра все узлы обязаны выдержать технологическую паузу в 9,6 мкс. Этот межкадровый интервал нужен для приведения сетевых адаптеров в исходное состояние, а также для предотвращения монопольного захвата среды одной станцией. После окончания технологической паузы узлы имеют право начать передачу своего кадра, так как среда свободна. В приведенном примере узел 2 дождался передачи кадра узлом 1, сделал паузу в 9,6 мкс и начал передачу своего кадра.

В разделяемой среде возможны ситуации, когда две станции пытаются одновременно передать кадр данных, при этом происходит искажение информации. Такая ситуация называется коллизией.

В Ethernet предусмотрен специальный механизм для разрешения коллизий. В большинстве случаев компьютеры сети объединяются с помощью концентраторов (concentrator, hub) или коммутаторов (commutator, switch). На рис.11.4 показано соединение с помощью концентратора.

Концентратор осуществляет функцию общей среды, то есть является логической общей шиной. Каждый конечный узел соединяется с концентратором с помощью двух витых пар. Одна витая пара требуется для передачи данных от станции к порту концентратора (выход T_x сетевого адаптера), а другая – для передачи данных от порта концентратора (вход R_x сетевого адаптера). Концентратор принимает сигналы от одного из конечных узлов и синхронно передает их на все свои остальные порты, кроме того, с которого поступили сигналы.

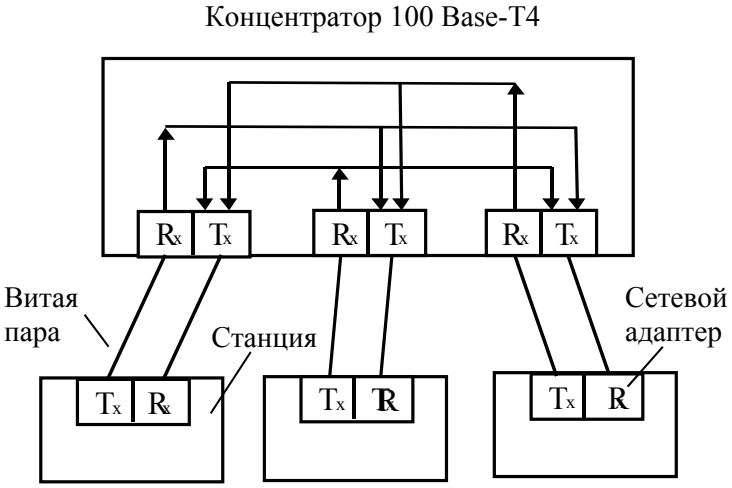


Рис.11.4. Сеть стандарта 100 Base-T4: T_x – передатчик; R_x – приемник

Основной недостаток сетей Ethernet на концентраторе состоит в том, что в них в единицу времени может передаваться только один пакет данных. Обстановка усугубляется задержками доступа из-за коллизий.

Пропускная способность Ethernet сетей значительно повышается при использовании коммутатора. Существует много типов коммутаторов. На рис.11.5 представлена схема коммутатора типа координатный переключатель. В таком коммутаторе каждый порт связан электрически со всеми другими портами. Очевидно, что в таком коммутаторе могут одновременно выполняться n пар обменов, где n – число портов. На рис.11.5 представлен частный случай логического замыкания портов.

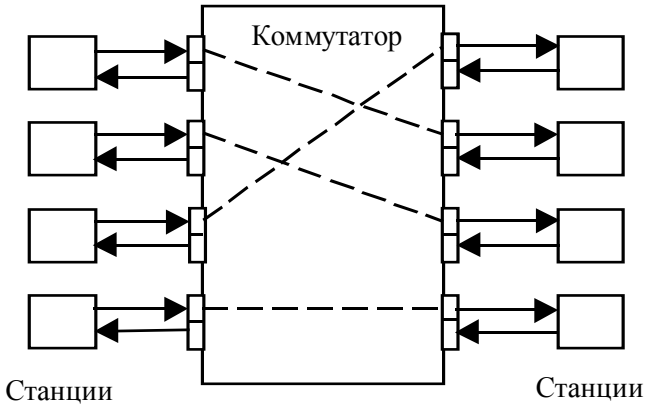


Рис.11.5. Параллельная передача кадров коммутатором

Для кластеров на локальных сетях как правило используется технология Fast Ethernet + Switch. Для монолитных кластеров, учитывая их небольшие размеры, используют технологии со значительно большей пропускной способностью.

Время передачи сообщения от узла *A* к узлу *B* в кластерной системе определяется выражением

$$T = S + L/R,$$

где S - латентность, *L* – длина сообщения, а *R* - пропускная способность канала связи. Латентность (задержка) – это промежуток времени между запуском операции обмена в программе пользователя и началом реальной передачи данных в коммуникационной сети. Другими словами – это время передачи пакета с нулевым объемом данных. В таблице 11.1 приведены характеристики некоторых коммуникационных сетей.

Таб.11.1 Характеристики некоторых коммуникационных технологий

Характеристика	SCI	Myrinet	Fast Ethernet
Латентность	5,6 мкс	17 мкс	170 мкс
Пропускная способность (MPI)	80 Мбайт/с	40 Мбайт/с	10 Мбайт/с
Пропускная способность (аппаратная)	400 Мбайт/с	160 Мбайт/с	12,5 Мбайт/с

11.4. Локальная сеть

Локальная сеть – это коммуникационная среда для обмена сообщениями между множеством компьютеров. Размеры локальных сетей колеблются от нескольких меторов в случае монолитных кластеров до десятков и сотен метров в случае сетей общего назначения на ПЭВМ. Элементами сети являются:

- компьютеры;
- коммуникационное оборудование;
- операционные системы.

Основой локальной сети является коммуникационное оборудование, которое включает:

- Сетевые адаптеры, модемы и драйверы. Эти элементы размещены в компьютерах.
- Кабельное хозяйство (витая пара, коаксиальный кабель, волоконно-оптический кабель).
- Коммутационное оборудование (хабы, коммутаторы).

На каждом компьютре сети расположена **сетевая** операционная система, которое управляет работой компьютеров в сети.

Следует четко различать модель OSI, описанную ранее, и стек OSI. Модель OSI является концептуальной схемой взаимодействия уровней, в то время как стек OSI – это конкретная ралиация модели OSI. Примеры некоторых реализаций приведены в таб.11.2.

Таб.11.2 Стеки OSI

<table>
<tr><th>Уровни модели OSI</th><th>Стек IBM/Microsoft</th><th>СтекTCP/IP</th></tr>
<tr><td>Прикладной</td><td rowspan="2">SMB</td><td rowspan="2">HTTP,Telnet, FTP,MPICH и др.</td></tr>
<tr><td>Представительный</td></tr>
<tr><td>Сеансовый</td><td rowspan="3">NetBIOS</td><td rowspan="2">TCP</td></tr>
<tr><td>Транспортный</td></tr>
<tr><td>Сетевой</td><td>IP, RIP, OSPF</td></tr>
<tr><td>Канальный</td><td colspan="2">Протоколы Ethernet, Fast Ethernet, Token Ring, и др.</td></tr>
<tr><td>Физический</td><td colspan="2">Коаксиал, витая пара, оптово-локно</td></tr>
</table>

11.5. Организация и функционирование кластера

Чтобы организовать кластер на базе локальной сети, принципиально дополнительного оборудования не требуется. Однако, для эффективного функционирования кластера следует учитывать, что:

- для кластера нужно использовать компьютеры с достаточно высоким быстродействием, иначе один компьютер с большим быстродействием будет со всех точек зрения эффективнее, чем сеть на низкоскоростных компьютерах;
- кластер на хабах имеет низкую эффективность, поэтому для обмена следует использовать коммутатор (switch).

Что касается аппаратуры для монолитных кластеров, то она заранее проектируется с учетом вышесказанного.

Основное отличие кластеров от локальных сетей заключается в системном программном обеспечении (СПО). Опишем основные требования к СПО на примере широко распространенной системы программирования для кластеров MPI (Message Passsing Interfase). Система MPI включает библиотеку функций, предназначенных для программирования межпроцессорного обмена данными. В библиотеку входят:

- функции для обмена между двумя процессами (функции точка-точка);
- функции для коллективного обмена: один-всем, все-одному;
- топологические функции организации данных (сетка, дерево);
- различные вспомогательные функции.

Этот набор функций (всего около 300 функций) включен в состав языков высокого уровня: C, C++, Fortran, то есть является расширением указанных ЯВУ. На основе этих функций создается параллельная программа, ветви которой распределяются по процессорам, запускаются и исполняются. В процессе исполнения осуществляется необходимый обмен данными.

Следовательно, сетевая операционная система, используемая в кластере, должна выполнять как минимум следующие операции:

- обеспечивать запуск процессов;
- осуществлять передачи данных между процессами;
- обеспечивать доступ к удаленной файловой системе и устройствам вывода.

В любой сетевой операционной системе имеются средства для удаленного запуска процессов, например, команда *rsh* (*remote start host*). Следовательно, проблема запуска ветвей параллельной программы, размещенных на разных процессорах, может быть решена операционной системой.

Но никакая операционная система не содержит реализацию 300 функций обмена, используемых в MPI. Для реализации этих функций разработчики MPI на основе возможностей ОС Linux создали специальный пакет MPICH, решающий эту и многие другие задачи. Такой же пакет сделан и для ОС Windows NT.

Таким образом, принципиальное отличие кластера от локальнойц сети – наличие специального программного пакета для реализации функций обмена MPICH.

11.6. Эффективность вычислений

Эффективность параллельных вычислений сильно зависит от объема обмена в выполняемом приложении и от свойст коммуникатора. Чтобы оценить это влияние, постороим некоторую формульную модель выполнения приложения на кластере.В качестве модели вычислений примем сетевой закон Амдала. Сетевой закон Амдала обладает большими возможностями расширения и позволяет учесть затраты времени на межпроцессорный обмен данными. В частном случае он может выглядеть так:

$$R = \frac{T_1}{T_n} = \frac{w \cdot t}{w_c \cdot t + \cdot \frac{w - w_c}{n} \cdot t + w_o \cdot t_o} = \frac{1}{a + \cdot \frac{1-a}{n} + \frac{w_o \cdot t_o}{w \cdot t}}$$

Здесь: T_1 - время решения приложения на однопроцессорной системе; T_n - время решения того же приложения на n - процессорной системе; w -число вычислительных операций в приложении; w_c - число скалярных операций в приложенийи; w_o - число операций обмена; t – время выполнения одной вычислительной операции; t_o - время выполнения одной операции обмена; a – удельный вес операций обмена.

Кроме ускорения для характеристики эффективности вычислений часто используют коэффициент использования процессорного времени h, который по существу является коэффициентом полезного действия процессоров (КПД):

$$h = \frac{R}{n}$$

Этот параметр позволяет оценить масштабируемость кластера. Под масштабируемостью понимают степень сохранения эффективности при изменении размеров системы (числа процессоров или размера данных).

Для простоты положим a=0, тогда

$$h = \frac{R}{n} = \frac{1}{1 + \frac{1}{n} \cdot \frac{w_0 \cdot t_0}{w \cdot t}}$$

Исследуем на этом упрощенном варианте сетевого закона Амдала, как меняется эффективность вычислений при увеличении числа процессоров. Для многих задач отношение времени обмена к времени счета

$$z = \frac{w_o \cdot t_o}{w \cdot t} \approx 0.005 \cdots 0.1$$

Соответствующий график для этого диапазона изменения z приведен ниже. Обычно считают, что КПД не должен быть ниже 0.5. Из графика видно, что этому условию в лучшем случае удовлетворяют 256 процессоров.

Значение z в сильной степени определяется характеристиками коммуникатора, то есть величиной $w_o \cdot t_o$. Поэтому можно допустить, что кривая для $w_o \cdot t_o = 0.10$ соответствует коммуникатору Fast Ethernet, а кривая $w_o \cdot t_o = 0.0.005$ - более быстрому коммуникатору, например, SCI.

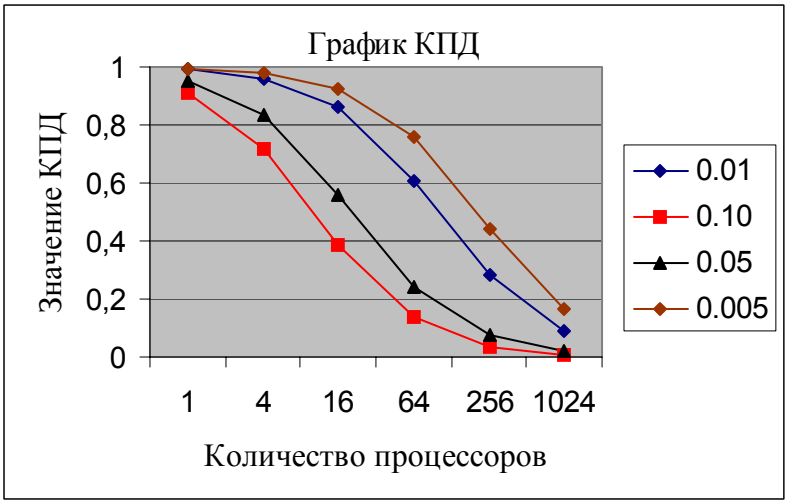


График наводит на простую мысль: параллелизм – это прежде всего борьба с обменом.

Лекция 12. Основы программирования в стандарте MPI

12.1. Стандарт MPI

Наиболее распространенной библиотекой параллельного программирования в модели передачи сообщений является **MPI (Message Passing Interface).** Рекомендуемой бесплатной *реализацией* MPI является пакет MPICH, разработанный в Аргоннской национальной лаборатории.

MPI [11, 12] является библиотекой функций межпроцессорного обмена сообщениями и содержит около 300 функций, которые делятся на следующие классы: операции точка-точка, операции коллективного обмена, топологические операции, системные и вспомогательные операции. Поскольку MPI является стандартизованной библиотекой функций, то написанная с применением MPI программа без переделок выполняется на различных параллельных ЭВМ. MPI содержит много функций, однако с принципиальной точки зрения для написания подавляющего большинства программ достаточно нескольких функций, которые приведены ниже.

Функция **MPI_Send** является операцией точка-точка и используется для посылки данных в конкретный процесс.

Функция **MPI_Recv** также является точечной операцией и используется для приема данных от конкретного процесса. Для рассылки одинаковых данных всем другим процессам используется коллективная операция

MPI_BCAST, которую выполняют все процессы, как посылающий, так и принимающие. Функция коллективного обмена

MPI_REDUCE объединяет элементы входного буфера каждого процесса в группе, используя операцию **op**, и возвращает объединенное значение в выходной буфер процесса с номером **root.**

MPI_Send(address, count, datatype, destination, tag, comm),
address – адрес посылаемых данных в буфере отправителя
count – длина сообщения
datatype – тип посылаемых данных
destination – имя процесса-получателя
tag – для вспомогательной информации
comm – имя коммуникатора

MPI_Recv(address, count, datatype, source, tag, comm, status)
address – адрес получаемых данных в буфере получателя
count– длина сообщения
datatype– тип получаемых данных
source – имя посылающего процесса процесса
tag - для вспомогательной информации
comm– имя коммуникатора
status - для вспомогательной информации

MPI_BCAST (address, count, datatype, root, comm)
root – номер рассылающего (корневого)процесса

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
sendbuf - адрес посылающего буфера
recvbuf - адрес принимающего буфера
count - количество элементов в посылающем буфере
datatype – тип данных
op - операция редукции
root - номер главного процесса

Кроме этого, используется несколько организующих функций.

MPI_INIT ()

MPI_COMM_SIZE (MPI_COMM_WORLD, numprocs)

MPI_COMM_RANK (MPI_COMM_WORLD, myid)

MPI_FINALIZE ()

Обращение к MPI_INIT присутствует в каждой MPI программе и должно быть первым MPI – обращением. При этом в каждом выполнении программы может выполняться только один вызов После выполнения этого оператора все процессы параллельной программы выполняются параллельно. MPI_INIT. MPI_COMM_WORLD является начальным (и в большинстве случаев единственным) коммуникатором и определяет коммуникационный контекст и связанную группу процессов.

Обращение MPI_COMM_SIZE возвращает число процессов numprocs, запущенных в этой программе пользователем.

Вызывая MPI_COMM_RANK, каждый процесс определяет свой номер в группе процессов с некоторым именем.

Строка **MPI_FINALIZE ()** должно быть выполнена каждым процессом программы. Вследствие этого заканчивается действие описанного в начале программы коммуникатора и никакие MPI – операторы больше выполняться не будут. Перемсенная COM_WORLD определяет перечень процессов, назначенных для выполнения пролграммы.

12.2. MPI программа для вычисления числа π на языке С.

Для первой параллельной программы удобна программа вычисления числа π, поскольку в ней нет загрузки данных и легко проверить ответ. Вычисления сводятся к вычислению интеграла приближенными методами по нижеследующей формуле:

$$Pi = \int_0^1 \frac{4}{1+x^2} dx \approx \frac{1}{n} \sum_1^n \frac{4}{1+x_i^2}$$

где $x_i = (i-1/2)/n$. Программа представлена рис.12.1.

При выполнении программы можно назначать ординаты процессам двумя способами:

- Для простоты объяснения примем: число процессов равно 10, а число ординат равно1000. Тогда первый метод состоит в том, что 1000 процессов разбивается на 10 участков, по 100 ординат в каждом. Тогда 0-му процессу назначается первый участок, 1-му процессу – второй участок, и так далее.
- Второй метод состоит в том, что длина участков делается равной числу процессов (10 – в нашем случае). Тогда к 0-му процессу относятся 1-ые ординаты всех участков (их попрежнему будет 100), к 1-му процессу относятся вторые ординаты всех участков, и так далее. Этот метод и использован в программе для организации цикла for (i = myid +1; i <= n; i+= numprocs).

```
#include "mpi.h"
#include <math.h>
int main ( int argc, char *argv[] )
{
int n, myid, numprocs, i;                /* число ординат, имя и число процессов*/
double PI25DT = 3.14159265358979323846264643;  /* используется для оценки
                                                    точности вычислений */
double mypi, pi, h, sum, x; /* mypi – частное значение π отдельного процесса, pi –
                                                        полное значение π */
MPI_Init(&argc, &argv);                                  /* задаются системой*/
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
while (1)
{
     if (myid == 0) {
     printf ("Enter the number of intervals: (0 quits) ");      /*ввод числа ординат*/
     scanf ("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0)                                  /*  задание условия выхода из программы */
break;
else {
h  = 1.0/ (double) n;  /* вычисление частного значения π некоторого процесса */
sum = 0.0;
for (i = myid +1; i <= n; i+= numprocs) {
x = h * ( (double)i - 0.5);
sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;       /* вычисление частного значения π некоторого процесса */
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                          MPI_COMM_WORLD);  /* сборка полного значения π */
    if (myid == 0)                                /* оценка погрешности вычислений */
     printf ("pi is approximately %.16f. Error is
                %.16f\n", pi, fabs(pi - PI25DT));
}
}
 MPI_Finalize();                                                    /* выход из MPI */
 return 0;
}
```

Рис.12.1 Программа вычисления числа π на языке С

12.3. Программа умножения матрицы на вектор

Задача умножения матриц является базовой операцией для многих приложений. Рассмотрим сначала задачу вычисления произведения матрицы на вектор, которая естественным образом обобщается на задачу умножения матриц. Результатом умножения матрицы на вектор является вектор результата, *i* – ый элемент которого является суммой парных произведений *i* – ой строки матрицы и заданного вектора. Для решения задачи используется алгоритм, в котором один процесс (главный) координирует работу других процессов (подчиненных). Для наглядности единая программа матрично-векторного умножения разбита на три части: общую часть (рис.12.2), код главного процесса (рис.12.3) и код подчиненного процесса (рис.12.4).

В общей части программы описываются основные объекты задачи: матрица *A*, вектор *b*, результирующий вектор *c*, определяется число процессов (не меньше двух). Задача разбивается на две части: главный процесс (master) и подчиненные процессы. В задаче умножения матрицы на вектор единица работы, которую нужно раздать процессам, состоит из скалярного произведения строки матрицы *A* на вектор *b*. Знаком ! отмечены комментарии.

```
   program main
   use mpi
   integer MAX_ROWS, MAX_COLS, rows, cols
   parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
!  матрица A, вектор b, результирующий вектор c
   double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_ROWS)
   double precision buffer (MAX_COLS), ans           /* ans – имя результата*/
   integer myid, master, numprocs, ierr, status (MPI_STATUS_SIZE)
   integer i, j, numsent, sender, anstype, row /* numsent – число посланных строк,
         sender – имя процесса-отправителя, anstype – номер посланной строки*/
   call MPI_INIT( ierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!  главный процесс –  master
   master = 0
!  количество строк и столбцов матрицы A
   rows   = 100
   cols   = 100
   if ( myid .eq. master ) then
!  код главного процесса
   else
!  код подчиненного процесса
   endif
   call MPI_FINALIZE(ierr)
   stop
   end
```

Рис.12.2. Программа умножения матрицы на вектор: общая часть

Код главного процесса представлен на рис.12.3. Единицей работы подчиненного процесса является умножение строки матрицы на вектор.

```
!    инициализация  A и b
     do 20 j = 1, cols
         b(j) = j
        do 10 i = 1, rows
           a(i,j) = i
10      continue
20    continue
      numsent = 0
!     посылка b каждому подчиненному процессу
      call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                           MPI_COMM_WORLD, ierr)
!     посылка строки каждому подчиненному процессу;  в  TAG номер строки = i
     do 40 i = 1,min(numprocs-l, rows)
        do 30 j = I, cols
            buffer(j) = a(i,j)
30      continue
        call MPI_SEND(buffer, cols,  MPI_DOUBLE_PRECISION, i, i,
                           MPI_COMM_WORLD, ierr)
        numsent = numsent + l
40   continue
!    прием результата от подчиненного процесса
     do 70 i = 1, rows
!        MPI_ANY_TAG – указывает, что принимается любая строка
        call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
                           MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
        sender     = status (MPI_SOURCE)
        anstype   = status (MPI_TAG)
!       определяем номер строки
        c(anstype) = ans
        if (numsent .lt. rows) then
!       посылка следующей строки
           do 50 j = 1, cols
              buffer(j) = a(numsent+l, j)
50         continue
           call MPI_SEND (buffer, cols, MPI_DOUBLE_PRECISION,  sender,
                    numsent+l, MPI_COMM_WORLD, ierr)
           numsent = numsent+l
        else
!       посылка признака конца работы
           call MPI_SEND(MPI_BOTTQM, 0, MPI_DOUBLE_PRECISION,sender,
                      0, MPI_COMM_WORLD, ierr)
        endif
70   continue
```

Рис.12.3. Программа для умножения матрицы на вектор: код главного процесса

Сначала главный процесс передает вектор *b* в каждый подчиненный процесс, затем пересылает одну строку матрицы *A* в каждый подчиненный процесс. Главный процесс, получая результат от очередного подчиненного процесса, передает ему новую работу. Цикл заканчивается, когда все строки будут розданы и получены результаты.

При передаче данных из главного процесса в параметре **tag** указывается номер передаваемой строки. Этот номер после вычисления произведения вместе с результатом будет отправлен в главный процесс, чтобы главный процесс знал, где размещать результат.

Подчиненные процессы посылают результаты в главный процесс и параметр **MPI_ANY_TAG** в операции приема главного процесса указывает, что главный процесс принимает строки в любой последовательности. Параметр **status** обеспечивает информацию, относящуюся к полученному сообщению. В языке Fortran это – массив целых чисел размера **MPI_STATUS_SIZE**. Аргумент **SOURCE** содержит номер процесса, который послал сообщение, по этому адресу главный процесс будет пересылать новую работу. Аргумент **TAG** хранит номер обработанной строки, что обеспечивает размещение полученного результата. После того как главной процесс разослал все строки матрицы A, на запросы подчиненных процессов он отвечает сообщением с отметкой 0.

Код подчиненного процесса представлен на рис.12.4.

```
!    прием вектора b всеми подчиненными процессами
     call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                          MPI_COMM_WORLD, ierr)
!    выход, если процессов больше количества строк матрицы
     if (numprocs  .gt. rows)    goto 200
!    прием строки матрицы
90   call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
                          MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
     if (status (MPI_TAG) .eq. 0) then  go to 200
!    конец работы
     else
          row = status (MPI_TAG)
!         номер полученной строки
          ans = 0.0
          do 100 i = 1, cols
!              скалярное произведение векторов
               ans = ans+buffer(i)*b(i)
100       continue
!         передача результата головному процессу
          call MPI_SEND(ans,1,MPI_DOUBLE_PRECISION,master,row,
                               MPI_COMM_WORLD, ierr)
          go to 90
!         цикл для приема следующей строки матрицы
     endif
200 continue
```

Рис.12.4. Программа для матрично-векторного умножения: подчиненный процесс

Каждый подчиненный процесс получает вектор b. Затем организуется цикл, состоящий в том, что подчиненный процесс получает очередную строку матрицы *A*, формирует скалярное произведение строки и вектора b, посылает результат главному процессу, получает новую строку и так далее.

12.4. Пакет MPICH

MPI – это описание библиотеки функций, которые обеспечивают обмен данными между процессами. Исполнительная среда – это то, что принадлежит локальной сети компьютеров: вычисли-

тельная и сетевая аппаратура, коммуникационные протоколы, операционная система. Следовательно, чтобы библиотека MPI работала в некоторой исполнительной среде, необходимо между описанием библиотеки и исполнительной средой иметь промежуточный слой, который называется *реализацией* MPI для данной исполнительной среды. Таким слоем является пакет MPICH (MPI Chameleon). Chameleon это пакет – предшественник MPICH. Основными функциями MPICH являются:

- Запуск всех ветвей программы, расположенных на разных процессорах
- Выполнение любой функции обмена, которая встречается в программе.
- Общение с исполнительной средой, профилирование, оценка эффективности параллельных вычислений и многое другое.

Для MPI разработано много специализированных реализаций, но MPICH является основной реализацией. MPICH, как и библиотека MPI, бесплатно доступны в Интернет.

Возможны два способа построения реализаций: прямая реализация для конкретной ЭВМ и реализация через **ADI** (Abstract Device Interface – интерфейс для абстрактного устройства).

Поскольку имеется большое количество типов реальных параллельных систем, то количество реализаций в первом случае будет слишком велико. Во втором случае строится реализация только для одного ADI, а затем архитектура ADI поставщиками параллельного оборудования реализуется в конкретной исполнительной среде (как правило, программно). Такая двухступенчатая реализация MPI уменьшает число вариантов реализаций и обеспечивает переносимость реализации. Поскольку аппаратно зависимая часть этого описания невелика, то использование метода ADI позволяет создать пакет программ, который разработчики реальных систем могут использовать практически без переделок. MPICH рализован через ADI.

Пакет MPICH содержит папки, содержимое которых содержит служебные программы и вспомогательную информацию:

- **doc** – содержит документацию по установке и использованию MPICH;
- **examples/nt** – содержит поддиректории с примерами простых программ и тесты для проверки правильности установки (TEST) и для проверки производительности кластера (PERFTEST);
- **include** – содержит подключаемые компилятором h-файлы для MPI-программ;
- **lib** – содержит lib-файлы для компоновки программ;
- **bin** – хранит файлы для запуска программ, например, файл mpirun;
- **mpe** – содержит исходные коды библиотеки MPE, используемые для профилирования, выполнения графических операций и визуализации логфайлов производительности;
- **mpid** – в папке хранятся исходные коды всех функций MPI и вспомогательные процедуры, написанные для различных ADI (p4, ch_p4, ch_p4dem и др.), выбираемых для конкретной исполнительной среды;
- **src** – содержит исходные коды всех функций MPI и вспомогательные процедуры MPICH; в этих кодах содержатся обращения к функциям соответствующих приборов из папки MPID, которые и обеспечивают выход на коммуникационные протоколы NetBIOS или TCP/IP;
- **www** – хранит HTML-версию описания функций. Файл index.html содержит ссылки на список функций MPI, а также на документацию по установке и настройке MPICH.

В директории **bin** содержатся следующие важные программы:

- **mpd** (Multipurpose Daemon) – сервис NT, позволяющий производить запуск MPI-программ; в данной реализации к нему обращается mpirun при запуске программ на кластере;
- **mpiconfig** – программа для конфигурации вычислительной сети с графическим интерфейсом, позволяет получить имена машин для использования в качестве узлов вычислительной сети, задать таймаут, сделать другие настройки;
- **mpirun** – утилита для запуска MPI-программ из командной строки. В качестве параметров принимает: число процессов, следующее за ключом -np или -localonly; имя запускаемой программы; другие параметры либо имя файла конфигурации, написанного в специальном формате. Сведения о форматах командной строки и файла конфигурации можно получить, запустив mpirun без параметров либо с параметром –help.

Лекция 13. Алгоритм решения СЛАУ в тесте Linpack

Оглавление

13.1 Метод Гаусса решения СЛАУ с постолбцовым выбором главного элемента

Рассматриваются системы вида:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1, \\ &a_{21}x_{11} + a_{22}x_2 + ... + a_{2n}x_n = b_2, \\ &..... \\ &a_{n1}x_1 + a_{n2}x_2 + ... + a_{mn}x_n = b_n, \end{aligned} \qquad (13.1)$$

или иначе, системы векторно-матричных уравнений

$$Ax=b.$$

Наиболее известным методом решения систем вида (13.1) является метод Гаусса [13], состоящий в последовательном исключении неизвестных. Будем поэтапно приводить систему (13.1) к треугольному виду, исключая последовательно сначала x_1 из второго, третьего,…, n-го уравнений, затем x_2 из третьего, четвертого,…, n-го уравнения преобразованной системы и так далее. На первом этапе заменим второе, третье,…, n-ое уравнения на уравнение, получающееся сложением этих уравнений с первым, умноженным соответственно на $-\frac{a_{21}}{a_{11}}, -\frac{a_{31}}{a_{11}}, ..., -\frac{a_{n1}}{a_{11}}$. Результатом этого этапа преобразования будет эквивалентная (13.1) система:

$$\begin{aligned} a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + ... + a_{1n}x_{1n} &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + ... + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 + ... + a_{3n}^{(1)}x_n &= b_3^{(1)} \\ ..... & \\ a_{n2}^{(1)}x_2 + a_{n3}^{(1)}x_3 + ... + a_{nn}^{(1)}x_n &= b_n^{(1)} \end{aligned} \qquad (13.2)$$

Коэффициенты в системе (с верхним индексом) подсчитываются по формулам:

$$a_{ij}^{(1)} = a_{ij} - \frac{a_{i1}}{a_{11}}a_{1j}, \quad b_i^{(1)} = b_i - \frac{a_{i1}}{a_{11}}b_1, \quad где \quad i, j = 2,3,...,n.$$

На втором этапе проделываем такие же операции, как и на первом, с подсистемой системы (13.2), получающейся исключением первого уравнения. Эквивалентный (13.2) результат второго этапа будет иметь вид:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\
a_{33}^{(2)}x_3 + \ldots + a_{3n}^{(2)}x_n &= b_3^{(2)} \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\
a_{n3}^{(2)}x_3 + \ldots + a_{nn}^{(2)}x_n &= b_n^{(2)}
\end{aligned}
\qquad (13.2a)
$$

где

$$
a_{ij}^{(2)} = a_{ij}^{(1)} - \frac{a_{i2}^{(1)}}{a_{22}^{(1)}} a_{2j}^{(1)}, \quad b_i^{(2)} = b_i^{(2)} - \frac{a_{i2}^{(1)}}{a_{22}^{(1)}} b_2^{(1)}, \quad \text{где} \quad i, j = 3,\ldots,n.
$$

Продолжая этот процесс, на (n -1)–ом этапе так называемого прямого хода метода Гаусса данную систему (13.1) приведем к треугольному виду:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\
a_{nn}^{(n-1)}x_n &= b_n^{(n-1)}
\end{aligned}
\qquad (13.3)
$$

Коэффициенты этой системы могут быть получены из коэффициентов данной системы последовательным пересчетом по формулам

$$
a_{ij}^{(k)} = a_{ij_{ij}}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}, \quad b_i^{(k)} = b_i^{(k)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} b_k^{(k-1)}, \quad (13.4)
$$

где верхний индекс k (номер этапа) изменяется от 1 до (n -1), нижние индексы i и j (в любой очередности) – от k+1 до n; по определению полагаем $a_{ij}^{(0)} := a_{ij}$, $b_i^{(0)} := b_i$.

Треугольная структура системы (13.3) позволяет последовательно одно за другим вычислять значения неизвестных, начиная с последнего:

$$
\begin{aligned}
x_n &= \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}; \\
&\ldots\ldots\ldots\ldots \\
x_2 &= \frac{b_2^{(1)} - a_{23}^{(1)}x_3 - \ldots - a_{2n}^{(1)}x_n}{a_{22}^{(1)}}; \\
x_1 &= \frac{b_1 - a_{12}x_2 - \ldots - a_{1n}x_n}{a_{11}}.
\end{aligned}
$$

Этот процесс последовательного вычисления неизвестных называют обратным ходом метода Гаусса. Он определяется одной формулой

$$
x_k = \frac{1}{a_{kk}^{(k-1)}} \left(b_k^{(k-1)} - \sum_{j=k+1}^{n} a_{kj}^{(k-1)} x_j \right),
$$

где k полагают равным n, n-1, …,2,1 и сумма по определению считается равной нулю, если нижний прел=дел суммирования у знака суммы имеет значение больше верхнего.

Таким образом, алгоритм Гаусса выглядит так:

1. *Для* $k = 1,2,...,n-1,2$

2. *Найти* $m \geq k$, *что* $|a_{mk} = \max\{|a_{ik}|\}$, *обменять строки* m *и* k

3. *Для* $i = k+1,...,n$:

4. $t_{ik} := a_{ik} / a_{kk}$,

5. $b_i := b_i - t_{ik} b_{k,}$;

6. *Для* $j = k+1,...,n$: (13.4)

7. $a_{ij} := a_{ij} - t_{ik} a_{kj}$.

8. $x_n := b_n / a_{nn}$;

9. *Для* $k = n-1,...,2,1$:

10. $x_k := \left(b_k - \sum_{j=k+1}^{n} a_{kj} x_j \right) \Big/ a_{kk.}$

Подав на его вход квадратную матрицу коэффициентов при неизвестныхсистемы (13.1) и вектор свободных членови выполнив три вложенных цикла прямого хода (строки 1 – 6) и один цикл вычислений обратного хода (строки 7 9), на выходе получим вектор –решение.

Чтобы уменьшить влияние ошибок округления на каждом этапе прямого хода уравнения системы обычно переставляют так, чтобы деление производилось на наибольший по модулю в данном столбце (обрабатываемом подстолбце) элемнт.Числа, на которые производится деление вметоде Гаусса, называются ведущими или главными элементами. Отсюда название – *метод Гаусса с постолбцовым выбором главного элемента (*или *с частичным упорядочиванием по столбцам).*

Частичное упорядочивание по столбцам требует внесения в алгоритм следуюших изменений : между строками 1 и 2 нужно сделать вставку:

- Найти такое $m \geq k$, что $|a_{mk}| = \max\{|a_{ik}|\}$ при $i \geq k$,
- иначе поменять местами b_k и b_m, a_{kj} и a_{mj} при всех $j=k,...,n$..

Сравнение метода единственного исключения с компактной схемой Гаусса.

Кроме изложенного выше метода Гаусса единственного исключения существуют и другие методы решения СЛАУ, например, метод LU- факторизации матриц, называемый компактной схемой Гаусса. Покажем, в чем сходство этих методов. В случае компактной схемы матрица представляется в виде произведения

$$A=LU,$$

где L – нижняя треугольная матрица, U – верхняя треугольная матрица. После нахождения матриц система Ax=b заменяется системой LUx=b и решение СЛАУ выполняется в два этапа:

$$Ly=b$$
$$Ux=y$$

Таким образом, решение данной системы с квадратной матрицей коэффициентов свелось к последовательному решению двух систем с треугольными матрицами коэффициентов.

Получим сначала формулы для вычисления элементов y_i вспомогательного вектора y. Для этого запишем уравнение Ly=b в развернутом виде:

$$
\begin{aligned}
&y_1 && = b_1 \\
&l_{21} y_1 + \quad y_2 && = b_2 \\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots && \\
&l_{n1} y_1 + l_{n2} y_2 + \ldots + l_{n,n-1} y_{n-1} + y_n && = b_n
\end{aligned}
$$

Очевидно, что все y_i могут быть последовательно найдены при i=1, 2, …., n по формуле

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k \qquad (13.5)$$

Развернем теперь векторно-матричное уравнение $Ux = y$:

$$
\begin{aligned}
u_{11} x_1 + u_{12} x_2 + \ldots\ldots + u_{1n} x_n &= y_1 \\
+ u_{22} x_2 + \ldots\ldots + u_{2n} x_n &= y_2 \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\
u_{nn} x_n &= y_n
\end{aligned}
\qquad (13.6)
$$

Отсюда значения неизвестных x_i находятся в обратном порядке, то есть при $i=n,\ n\text{-}1,\ \ldots 2,\ 1,$ по формуле

$$x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{k=i+1}^{n} u_{ik} x_k \right) \qquad (13.7)$$

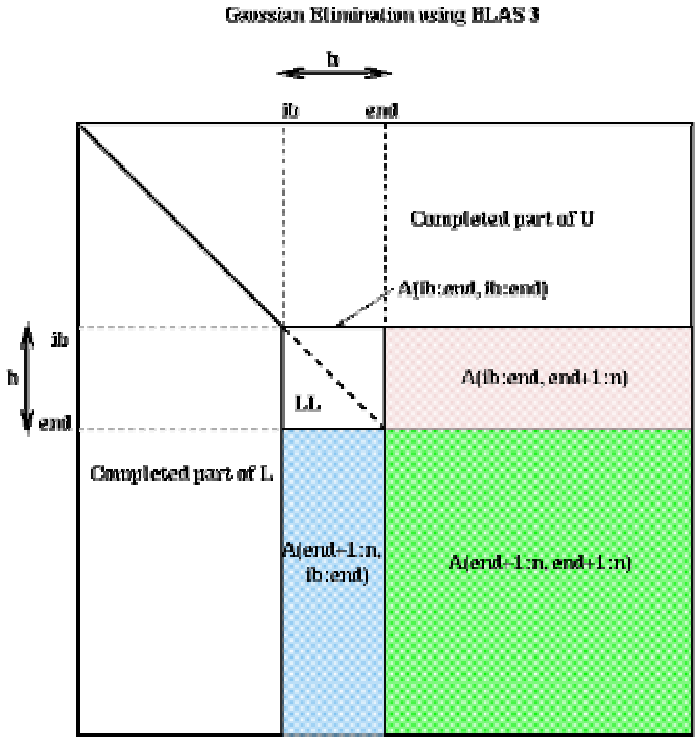
Сходство метода Гаусса (МГ) с компактной схемой Гаусса (КСГ) состоит в том, что элементы матриц L и U в КСГ соответствуют по величине коэффициентам, получаемым при разложении в МГ. Например, матрица U в КСГ строго соответствует коэффициентам при неизвестных в системе (13.3) для МГ. Переход от МГ к КСГ рассмотрим на примере разложения в МГ (13.2): после исключения x_1, начиная со второй строки и ниже получается подматрица с верхним левым элементом $a_{21}^{(1)}$. Все элементы левого столбца этой подматрицы составляют очередной столбец матрицы L в КСГ, а верхняя строка – строку в матрице U (за исключением $a_{21}^{(1)}$ в методе Краута). При исключении следующего элемента в МГ (рис. 13.2а) ситуация повторяется.

Сходство метода Гаусса (МГ) с компактной схемой Гаусса (КСГ) состоит в том, что элементы матриц L и U в КСГ соответствуют по величине коэффициентам, получаемым при разложении в МГ. Например, матрица U в КСГ строго соответствует коэффициентам при неизвестных в системе (13..3) для МГ. Переход от МГ к КСГ рассмотрим на примере разложения в МГ (13.2): после исключения x_1, начиная со второй строки и ниже получается подматрица с верхним левым элементом $a_{21}^{(1)}$. Все элементы левого столбца этой подматрицы составляют очередной столбец матрицы L в КСГ, а верхняя строка – строку в матрице U (за исключением $a_{21}^{(1)}$ в методе Краута). При исключении следующего элемента в МГ (13.2а) ситуация повторяется.

13.2. Методы блочного размещения данных в кластере

Теперь мы рассмотрим размещение матрицы по слоям в машинах с распределенной памятью, с целью наиболее эффективного выполнения гауссового исключения [14, 15]. Мы обсудим порядок

слоев данных, начиная с самого простого, но не эффективного варианта и продвигаться к более эффективным вариантам. Система обозначений показана на следующем рисунке.

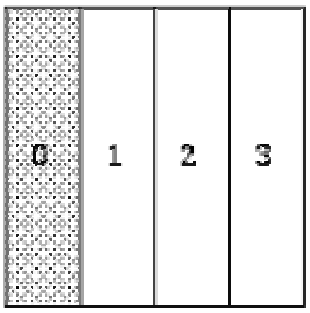


Двумя главными затруднениями в выборе размещения данных для гауссова исключения являются:

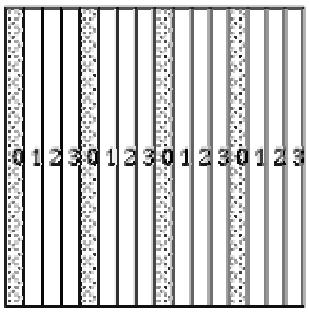
- Баланс нагрузки, то есть обеспечение загрузки всех процесоров на протяжении всего времени вычислений
- Возможность использования BLAS3 на одном процессоре, чтобы подсчитать иерархию памяти на каждом процессоре

Примечание. Уровенень BLAS1 библиотеки BLAS используется для выполнения операций вектор-вектор, уровенень BLAS2 – для выполнения матрично-векторных операций, уровенень BLAS3 – для выполнения матрично-матричных операций.

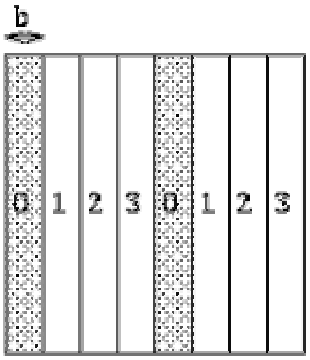
Понять эти проблемы помогает понять это рисунок ниже. Для удобства мы будем нумеровать процессоры от 0 до p-1, и матричные столбцы (или строки) от 0 до n-1. Во всех случаях каждая подматрица обозначается номером процессора (от 0 до 3), который содержит ее. Процессор 0 представлен затененными подматрицами.



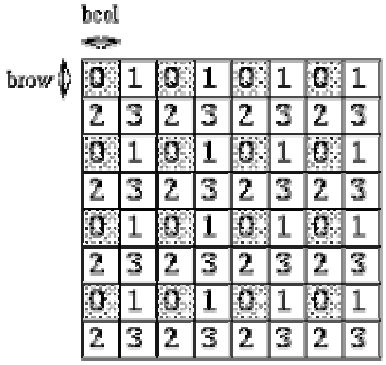
1) Column Blocked Layout



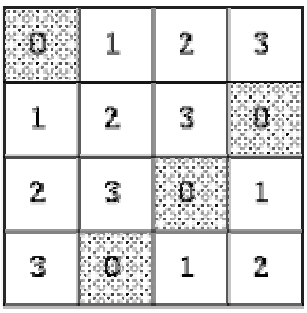
2) Column Cyclic Layout



3) Column Block Cyclic Layout



4) Row and Column Block Cyclic Layout



5) Block Skewed Layout

Рассмотрим перый вариант – размещение по столбцам матрицы A (Column Blocked Layout). При этом разбиении столбец *i* хранится в последнем незаполненном процессоре, если считать, что c=ceiling(n/p) есть максимальное число столбцов, приходящееся на один процессор и вести счет столбцов слева направо. На рисунке n = =16, p = 4. Это разбиение не позволяет сделать хорошую балансировку нагрузки, поскольку как только первые *c* столбцов завершены, процесор 0 становтся свободным до конца вычислений. Размещение по строкам (Row Blocked Layout) создает такую же проблему.

Другой вариант – циклическое размещение по столбцам (Column Cyclic Layout) использует для решения проблемы простоев назначение столбца *i* процессору с номером *i* mod *p*. Однако, тот факт, что хранятся одиночные столбцы, а не их блоки, означает, что мы не можем использовать BLAS2 для факторизации A(ib:n,ib:end) и возможно не сможем использовать BLAS3 для обновле-

ния A(end+1:n,end+1:n). Циклическое размещение по строкам (Row Cyclic Layout) создает такую же проблему.

Третье размещение - столбцовый блочно- циклический вариант (*Column Block Cyclic Layout)* есть компромисс между двумя предыдущими. Мы выбираем размер блока b, делим столбцы на группы размера b, и распределяем эти группы циклическим образом. Это означает, столбец i хранится в процессоре (последний (i/b)) mod p. В действительности это распределение включает первые два как частный случай b=c=ceiling(n/p) и b=1, соответственно. На рисунке n=16, p=4 and b=2. Для b > 1 это имеет слегка худший баланс, чем *Column Cyclic Layout*, но можно использовать BLAS2 и BLAS3. Для b < *c*, получается лучшая балансировка нагрузки *Columns Blocked Layout*, но можно использовать BLAS только на меньших подпроблеммах. Однако, это размещение имеет недостаток в том, что факторизация A(ib:n,ib:end) будет иметь место возможно только на процессоре, где столбцовые блоки в слоях соответствуют столбцовым блокам в гауссовом исключении. Это будет последовательный bottleneck (узкое место).

Последовательный bottleneck облегчается четвертым размещением – двумерное блочно-циклическое размещение (*2D Block Cyclic Layout*). Здесь мы полагаем, что наши *p* процессоров аранжированы в prow × pcol прямоугольный массив поцессоров, индексируемый 2D образом (pi,pj), 0 <= pi < prow и 0 <= pj < pcol. Все процессоры (i,j) с фиксированным j обращаются к процессорному столбцу *j*. Все процессоры (i,j) с фиксированным i обращаются к процессорной строке *i*. Вход матрицы (i,j) маркируется к процессору (pi,pj) путем назначения i до pi и j до pj независимо, используя формулу для блочно-циклического размещения:

pi = последний (i/brow) mod prow, где
brow = размер блока в направлении строк (block size in the row direction)

pj = последний (j/bcol) mod pcol, где
bcol = размер блока в направлении столбцов (block size in the column direction)

Поэтому, это размещение включает все предыдущие и их транспозиции как специальные случаи. На рисунке n=16, p=4, prow=pcol=2, and brow=bcol=2. Это размещение позволяет pcol-fold параллелизм в любом столбце и использует BLAS2 и BLAS3 на матрице размера brow × bcol. Это размещение мы будем использовать для гауссова исключения.

Поскольку гауссово исключение не целиком «симметричное» (имеется много коммуникаций, требуемых во время BLAS2 факторизации A(ib:n,ib:end), чем во время BLAS3 обновления A(ib:end,end+1:n)), неясно, какой выбор полностью симметричного размещения наиболее эффективен. Впрочем, позже мы посмотрим prow.

Есть еще одно размещение – блочное смещенное размещение (*Block Skewed Layout*), которое не является частным случаем 2D Block Cyclic Layout. В нем имеется особенность, что каждая строка и каждый столбец распределяются среди всех p процессоров. Так называемый винтовой (p-fold) параллелизм пригоден для любых строчных и столбцовых операций.

13.3. Варианты LU Decomposition

Возможны три естественных варианта для LU decomposition: левосторонний поиск, правосторонний поиск и метод Краута (right-looking, left-looking and Crout).

- left-looking вариант вычисляет блочный столбец зараз, используя ранее вычисленные столбцы.
- right-looking вариант вычисляет на каждом шаге строчно-столбцовый блок (block row column) и использует их затем для обновления заключительной подматрицы. Этот метод называется также рекурсивным алгоритмом. Термины right and left относятся к области доступа к данным.
- Crout вариант представляет гибрид left- and right версий.

Графическое представление алгоритмов дано на рис. 3.6, 3.7, 3.8 ниже.

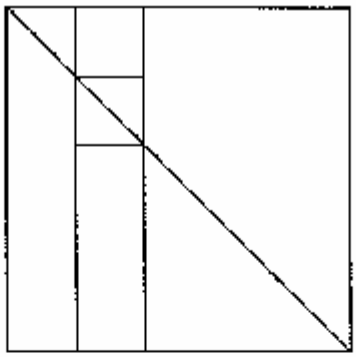


Figure 3.6 Left-Looking LU Algorithm

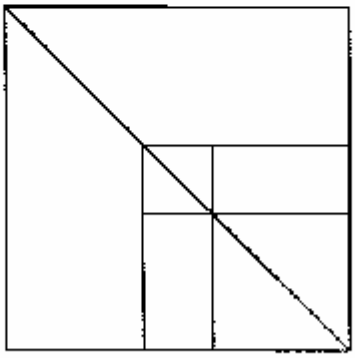


Figure 3.7 Right-Looking LU Algorithm

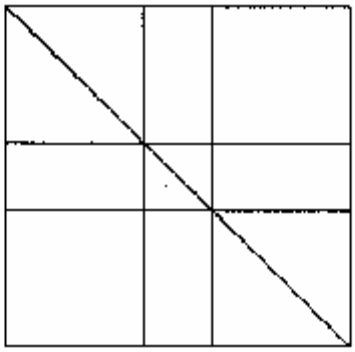
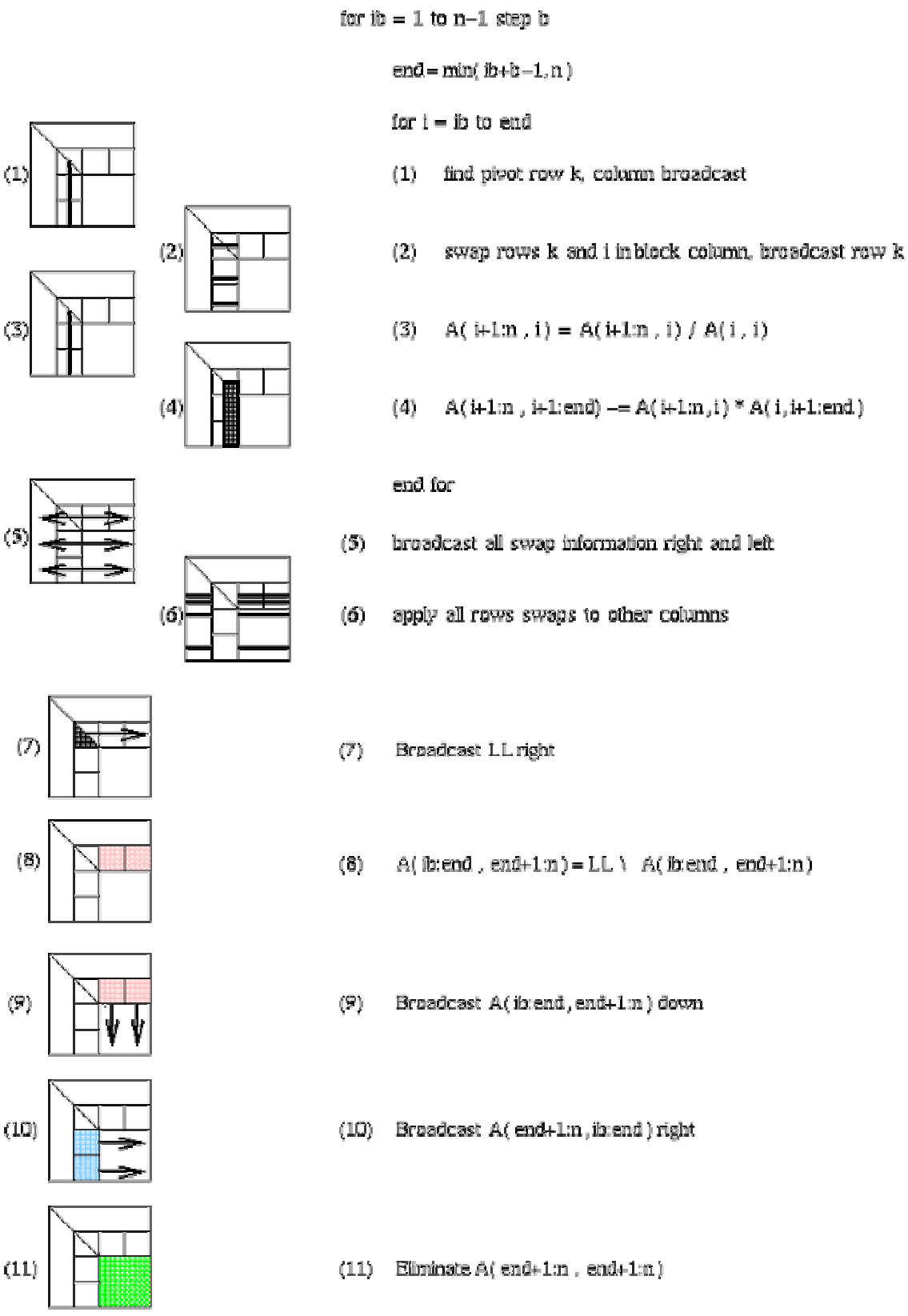


Figure 3.8 Crout LU Algorithm

Distributed Gaussian Elimination with a 2D Block Cyclic Layout



Вышеприведенный рисунок (и программа) показывает, как алгоритм BLAS3 выполняется на двумерном блочно-циклическом размещении исходной матрицы (2D block cyclic layout). Блок размера *b* и блочные размеры *brow* и *bcol* в размещении удовлетворяют *b=brow=bcol*. Затененные области отмечают занятые процессоры или выполненные коммуникации. Программа рисунка повторяется ниже, она во многом соответствует алгоритму (13.4).

```
for ib = 1 to n–1 step  b
end  = min(ib+b-1, n)
        for i =ib to end
        (1)   find pivot row k, column broadcast
        (2)   swap rows k and i in block column,   broadcast row k
        (3)   A(i+1:n, i) = A(i+1:n,i) / A(i , i)
        (4)   A(i+1:n, i+1: end) - = A(i+1:n, i) * A(I ,i+1:end)                    (13.8)
        end for
  (5) broadcast all swap information right and left
  (6) apply all rows swaps to other columns
  (7) broadcast LL right
  (8) broadcast A(ib:end, end+1:n) = LL\ A(ib:end +1:n)
  (9) A(ib:end, end +1:n) down
  (10)  broadcast A(end+1:n, ib:end) right
  (11)  Eliminate A(end+1:n, end+1:n)
```

Рассмотрим для примера выполнение алгоритма по шагам. Программа содержит двойной цикл. Внешний цикл соответствует перебору диагональных блоков, а внутренний обеспечивает перебор по отдельным столбцам внутри блока для выполнения всех операций для выбранного столбца. блока.

Шаг (1) требует операции редукции (редукция - коллективная операция MPI поиска максимального элемента в массиве процессоров) среди prow процессоров, владеющих текущим столбцом, чтобы найти найти наибольший абсолютный вход (pivot) и широковещать его индекс. Выполняется во внешнем цикле.

Шаг (2) требует обмена среди процессоров строк *k* и *i* в столбце и широковещает ведущую строку *k* всем процессорам в столбце.

Шаги (3) and (4) выполняют локальные вычисления BLAS1 and BLAS2.
(3) – это вычисление столбца L_k, соответствует циклу 3 в (10).
(4) – это вычисления строки U_k и коррекция массива остальных усеченных строк. Это третий вложенный цикл, записанный на языке Matlab и соответствует циклу 6 в (10) или (2.4).
Шаги (1) - (4) заканчивают обработку одного столбца в выбранном во внешнем цикле блоке, а внутренний цикл обеспечивает обработку всех столбцов блока.

Шаги 5) and (6) используют коммуникации, широковещая ведущую информацию и обменивая строки среди всех других p-prow processors. Эти шаги выполняются уже во внешнем цикле.
Шаг (5) нужен для передачи в обе стороны индексов строк *k* и *i*, для которых выполнен обмен.
Шаг (6) используется для перемещения строк *k* и *i* слева и справа по вертикали.

Шаг (7) выполняет много обмена , посылая LL всем pcol processors в его строке. Эта информация необходима для формирования всех сторок, соответствующих диагональному блоку.

Шаг (8) есть локальные вычисления всех строк, соответствующих диагональному блоку. Неясен слэш.

Шаги (9) and (10) обеспечивают коммуникацию: столбец и строка матричных блоков, соответствующих данному диагональному матричному блоку, посылаются вправо и вниз для выполнения матричных умножений.

Шаг (11). Вычисления, аналогичные шагам 3 и 4**,** которые обновляют остаточную юго-восточную остаточную площадь матрицы.

Нет необходимости иметь баръер между каждым шагом алгоритма. Другими словами, может быть параллелизм между разичными шагами алгоритма, формируя конвейер. Например, рассмотрим шаги steps 9, 10 and 11, где имеет место большая часть коммуникаций и счета. Как только процессор получил требуемый ему (blue) субблок A(end+1:n,ib:end) слева и (pink) субблок A(ib:end:end+1:n) сверху, он может его локальное умножение для лбновления его части (green) подматрицы A(end+1:n,end+1:n). Как только самый левые b столбцов A(end+1:n,end+1:n) обновлены, их LU factorization может начинаться, в то время как остающимиеся столбцы зеленой подматрицы будут обновляться другими процессорами.

13.5. Эффективность вычислений

Для дальнейшего расчета предположим, что:

- Одна плавающая операция на скорости матричного умножения требует одну единицу времени (для абсолютных расчетов $t=1/v$ *сек*, где v –быстродействие процессора).
- Посылка сообщения из n слов от *одного* процесора другому требует alpha + beta*n единиц времени, где alpha – начальная задержка передачи сообщения, beta – время передачи одного слов данных, n – количество переданных слов.
- Коллективных операций нет.
- Имеется p процессоров, объединенных в prow×-pcol решетку.
- b есть размер блока в 2D block cyclic размещении.
- n есть размер исходной матрицы

Пусть время исполнения алгоритма Time по методу Гаусса равно сумме:

Time = msgs * alpha + words * beta + flops, единиц времени

где **msgs** есть число посланных сообщений (без учета параллельно посланных), **words** есть число слов (без учета параллелизма) и **flops** есть число выполненных плавающих операций (без учета параллелизма). Чтобы упростить представление, **мы будем учитывать в деталях только шаги 9, 10 и 11.** Это практически верно для больших матриц.

Шаг (9). Более точно, в шаге 9 мы будем использовать древовидное широковещание в каждом процессорном столбце с первым процессором, посылающим двум другим, и так далее, так что общее число log_2 prow сообщений требуется для посылки всех сообщений в столбце (все процессоры по вертикали работают параллельно). Поскольку размер сообщения есть b*(n-end)/pcol (суммарное сообщение для всех процесоров), для шага 9 требуется единиц времени

Time for step 9 = (log_2 prow) * (alpha + (b*(n-end)/pcol)*beta), единиц времени

Здесь b*(n-end)/pcol распадается на части:

- b*(n-end) – длина полосы шириной b от end до самого низа матрицы (n).
- pcol – количество процессоров в матрице процессов.
- b*(n-end)/pcol – размер блока.

- Здесь надо делить на pcol, поскольку число процессоров (pcol) остается постоянным, размер остаточной матрицы уменьшается с каждой итерацией и, следовательно, размер передаваемого сообщения уменьшается.

Шаг (10). Мы будем использовать широковещание на основе кольца, когда каждый процессор посылает соседу справа, поэтому требуется pcol посылок. Поскольку размер сообщения есть b*(n-end)/prow, потребуется

(pcol) * (alpha + (b*(n-end)/prow)*beta), единиц времени

Однако, поскольку только *самый левый* процессорный столбец в зеленой подматрице нуждается получить его сообщение, pass it on, и обновить его подматрицу перед LU факторизацией следующего блочного столбца, мы только расходуем

Время шага 10 = 2 * (alpha + (b*(n-end)/prow)*beta), единиц времени

Обновление остальной части зеленой подматрицы можно делать во время шагов (1) – (10) внешнего цикла основного алгоритма для следующего диагонального блока.

Шаг (11). Каждый процессор делает умножение матриц $\left[\frac{n-end}{pcol}\times b\right]\times\left[b\times\frac{n-end}{prow}\right]$, которое требует

Время шага 11 = 2*b*(n-end)^2/p, единиц времени

Суммарный результат. Чтобы оценить общие затраты для распределенного МГ, нам надо просуммировать приведенные выше три составляющие для end = b, 2*b, 3*b, ... , n-b и получить

```
Время для шагов 9, 10 11 =
    ( ( n * (log_2 prow) + 2 ) / b ) * alpha
  + ( n^2 * ( (log_2 prow)/(2*pcol) + 1/prow ) ) * beta
  + ( (2/3)*n^3/p )
```

Принимая во внимание *другие шаги алгоритма* в подсчете повышений коэффициентов alpha и beta, конечные затраты будут таковы:

```
Полное время для метода Гаусса =
    ( n * (6+ log_2 prow) ) * alpha                                    (13.9)
  + ( n^2 * ( 2*(prow-1)/p + (pcol-1)/p + (log_2 prow)/(2*pcol))) * beta
  + ( (2/3)*n^3/p )
```

Вычислим эффективность путем делением последовательного времени (2/3)*n^3 на *p* раз времени, представленного выше, и получаем:

```
Efficiency = 1 / (  1
  + (1.5*p*(6 + log_2 prow)/n^2) * alpha
  + ( 1.5*( 2*prow + pcol                                              (13.10)
  + (prow * log_2 prow)/2 -3 )/n )* beta )(ib:end, end+1:n)
```

Исследуем формулу, чтобы выяснить,, когда мы можем ожидать хорошей эффективности. Первое, для фиксированных *p, prow, pcol, alpha* and *beta* эффективность растет пропорционально *n.* Это потому, что мы делаем O(n^3) плавающих операций, но коммуникаций только для O(n^2) слов, так что плавающая точка, которая балансирует нагрузку, превосходит коммуникацию, по-

этому эффективность хорошая. Эффективность также растет если *alpha* и *beta* уменьшаются, то есть коммуникации становятся дешевле.

Теперь мы рассмотрим выбор *prow* и *pcol*. Выражение

$$2*prow + pcol + (prow * log\_2\ prow)/2 = 2*prow + 1/prow + (prow * log\_2\ prow)$$

минимизируется, когда *prow* слегка меньше, чем sqrt(p), означая, что нам хотелось бы иметь процессорную решетку prow × pcol, которая слегка длинее, чем выше.

Обобщим, полагая prow ~ pcol ~ sqrt(p) и игнорируя log terms. Тогда эффективность есть:

$$Efficiency = 1 / (1 + O(p/n^2 * alpha) + O(sqrt(p)/n * beta)) = E(n^2/p)$$

то есть эффективность растет с увеличениме функции n^2/p. Однако, n^2/p есть сумма данных, хранимых на одном процесоре, поскольку n×n матрица требует n^2 слов. Другими словами, если мы позволяем общему ращ=змеру проблемы n^2 расти пропорционально числу процессоров, мы получим эффективность постоянной.

Экспериментально показано, что предсказанные оценки времени выполнения алгоритма решения СЛАУ и реально измеренные в эксперименте достаточно близки, ошибка составляет несколько процентов. Это же относится и к расчету эффективности.

Лекция 14. Распределенные вычисления

14.1. Что такое Грид

Грид — географически распределенная инфраструктура, объединяющая множество ресурсов разных типов (процессоры, долговременная и оперативная память, хранилища и базы данных, сети), доступ к которым пользователь может получить из любой точки, независимо от места их расположения. Грид предполагает коллективный разделяемый режим доступа к ресурсам и к связанным с ними услугам в рамках глобально распределенных виртуальных организаций, состоящих из предприятий и отдельных специалистов, совместно использующих общие ресурсы.

Слово Грид не аббревиатура, и в прямом переводе обозначает «решетка», но правильнее его употреблять в смысле «вычислительная сеть» по аналогии с энергосетью, из которой можно потреблять энергию, не заботясь о ттом, кем и где она произведена.

Грид отличается от Web Services в следующем:

•Грид предоставляет заказчику вычислительные мощности для выполнения его задач

•Web Services выполняет на своем оборудовании и программном обеспечении функцию, запрошенную заказчиком, напрмер, перевод текста с некоторонго языка. Механизмы Web Services являются основой Грид.

Основой Грид являются ресурсы – вычислительные мощности (компъютеры) и средства для хранения больших массивов данных (дисковые массивы или другие гоители информации).

Ресурсы могут быть представлены в виде кластеров ПЭВМ, суперЭВМ и различных вариантов их объединения. Особенно интересны блеэд-компъютеры, которые в едином конструктиве содержат десятки и сотни процессоров, каждый из них имеет интернет выход во внешнюю среду.
Существуют сотни поставщиков ресурсрв и специализированных Грид, работающих на скоростных сетях. Особенно интересны ресрсы IBM, Sun, Google и др. Непременным условием использования ресурсов является их доступность и надежность. За низкое качество обслуживания поставщие ресурсов несет материальную ответственность.

Примером наиболее развитой Грид является организация WLCG (Worldwide LHC Computing Grid, http://lcg.web.cern.ch/), которая размещается в международном центре ядерных исследований ЦЕРНе (Женева, Швейцария, http://www.cern.ch/). Она объединяет с помощью магистральная европейская сеть **GEANT** сотни компьютерных центров во всем мире, включающих несколько десятков тысяч современных мощных компьютеров.

Задача WLCG - обработка и анализ беспрецедентного объема (около 15-20 Петабайт в год, 1 Петабайт=10^{15} байт) экспериментальных данных, которые будут поступать с наибольшего в мире ускорителя элементарных частиц и ядер LHC (Large Hadron Collider). Ускоритель начнет работать в ЦЕРНе в 2008 году.

Основой организационной структуры Грид является виртуальная организация. Инфраструктура Грид основана на предоставлении ресурсов в общее пользование, с одной стороны, и на использовании публично доступных ресурсов, с другой. В этом плане, ключевым понятием инфраструктуры Грид является виртуальная организация (ВО), в которую кооперируются как потребители, так и владельцы ресурсов. В существующих Грид - системах виртуальная организация представляет собой объединение специалистов из некоторой прикладной области, которые объединяются для достижения общей цели.

Любая ВО располагает определенным количеством ресурсов, которые предоставлены зарегистрированными в ней владельцами (некоторые ресурсы могут одновременно принадлежать нескольким ВО). Каждая ВО самостоятельно устанавливает правила работы для своих участников, исходя из соблюдения баланса между потребностями пользователей и наличным объемом ресурсов, поэтому пользователь должен обосновать свое желание работать с Грид -системой и получить согласие управляющих органов ВО.

Для чего нужны Грид? Среди основных задач, решаемых с помощью Грид -технологий, на данный момент можно выделить:

- организация эффективного использования ресурсов для небольших задач, с утилизацией временно простаивающих компьютерных ресурсов. Это и есть работа по запросу.
- распределенные супервычисления, решение очень крупных задач, требующих огромных процессорных ресурсов, памяти и т.д. Это имеет название метакомпъютинг.
- вычисления с привлечением больших объемов географически распределенных данных, например, в метеорологии, астрономии, физике высоких энергий;
- коллективные вычисления, в которых одновременно принимают участие пользователи из различных организаций.

Грид – географически распределенная инфраструктура, объединяющая множество ресурсов разных типов (процессоры, долговременная и оперативная память, хранилища и базы данных, сети), доступ к которым пользователь может получить из любой точки, независимо от места их расположения. Грид предполагает коллективный разделяемый режим доступа к ресурсам и к связанным с ними услугам в рамках глобально распределенных виртуальных организаций, состоящих из предприятий и отдельных специалистов, совместно использующих общие ресурсы. В этом пространстве должны быть сосредоточены **универсальные** средства аутентификации, авторизации и доступа к ресурсам, средства поиска необходимых ресурсов и так далее.

14.2. Архитектура Грид.

Правильно разработанная и хорошо реализованная грид-среда характеризуется следующими основными функциональными возможностями [4]:

- доступ должен быть виртуальным (нужен доступ не к серверам, а к сервисам, поставляющим данные или вычислительные ресурсы — причем без необходимости знания аппаратной структуры, обеспечивающей эти сервисы);
- доступ должен осуществлять по требованию (с заданным качеством), а ресурсы должны предоставляться тогда, когда в них возникает нужда;
- доступ должен быть распределенным, обеспечивая возможность совместной коллективной работы виртуальных команд;
- доступ должен быть устойчив к сбоям, а при выходе из строя серверов приложения должны автоматически мигрировать на резервные серверы;
- доступ должен обеспечивать возможность работы в гетерогенной среде – с различными платформами.

Необходимо отметить, что не все из этих требований в должной мере реализованы в настоящее время.

В дальнейшем мы сосредоточимся на случае глобального грида (в соответствии с классификацией, приведенной в предыдущем разделе). Для такой системы важнейшим условием эффективной работы является обеспечение взаимодействия (интероперабельности) между различными платформами, языками и программными средами.

В сетевой среде интероперабельность подразумевает работу по общим протоколам. Протоколы регламентируют взаимодействие элементов распределенной системы, а также структуру передаваемой информации.

С другой стороны, как уже отмечалось ранее, и как мы более подробно расскажем ниже, функциональной базовой компонентой грид-системы является сервис (служба). Поэтому при формулировке общих принципов построения грида важно определить как структуру протоколов, на которых основана его работа, так и его архитектуру в терминах сервисов.Образно говоря, архитектура грид-систем имеет две «проекции» - протокольную и сервисную.

Общая структура глобального грида описывается в виде стека (набора уровней или слоев) протоколов. В такой модели каждый уровень предназначен для решения узкого круга задач и ис-

пользуется для предоставления услуг для более высоких уровней. Верхние уровни ближе к пользователю и работают с наиболее абстрактными объектами, тогда как нижние уровни сильно зависят от физической реализации грид-ресурсов. Полезно иметь в виду, что эта структура аналогична сетевой модели OSI (Open Systems Interconnection Reference Model; модель взаимодействия открытых систем), - абстрактной модели для сетевых коммуникаций и разработки сетевых протоколов (см., например, [38]). В левой части рис. 14.1 показаны уровни стека грид-протоколов, а справа - четыре аналогичных им уровней модели OSI (всего в стеке OSI семь уровней).

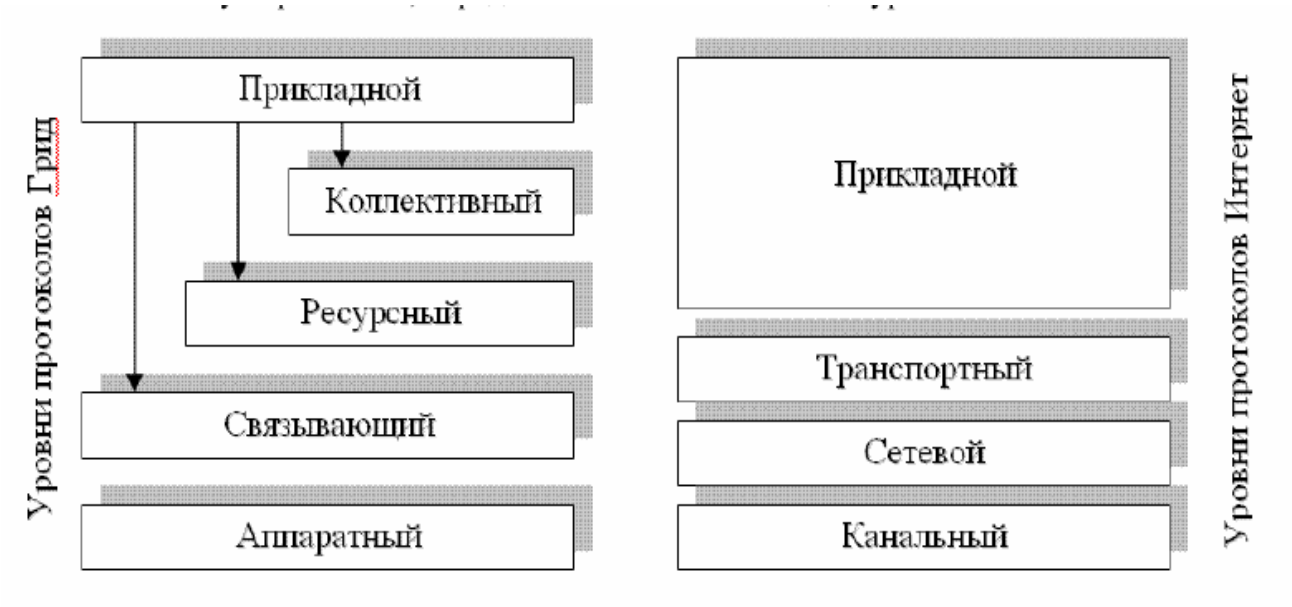


Рис.14.1 Стеки протоколов грид системы и сетевой модели

Итак, стек грид-протоколов включает:

1. аппаратный (базовый) уровень (Fabric Layer) составляют протоколы, по которым соответствующие службы непосредственно работают с ресурсами;
2. связывающий уровень (Connectivity Layer) составляют протоколы, которые обеспечивают обмен данными между компонентами базового уровня и протоколы аутентификации;
3. ресурсный уровень (Resource Layer) – это ядро многоуровневой системы, протоколы которого взаимодействуют с ресурсами, используя унифицированный интерфейс и не различая архитектурные особенности конкретного ресурса;
4. коллективный (Collective Layer) уровень отвечает за координацию использования имеющихся ресурсов;
5. прикладной уровень (Application Layer) описывает пользовательские приложения, работающие в среде виртуальной организации; приложения функционируют, используя протоколы, определенные на нижележащих уровнях.

Аппаратный уровень

Аппаратный уровень (Fabric Layer) описывает службы, непосредственно работающие с ресурсами. Ресурс является одним из основных понятий архитектуры Грид. Ресурсы могут быть весьма разнообразными, однако, как уже упоминалось, можно выделить несколько основных типов (рис. 14.2):

- вычислительные ресурсы;
- ресурсы хранения данных;

- информационные ресурсы, каталоги;
- сетевые ресурсы

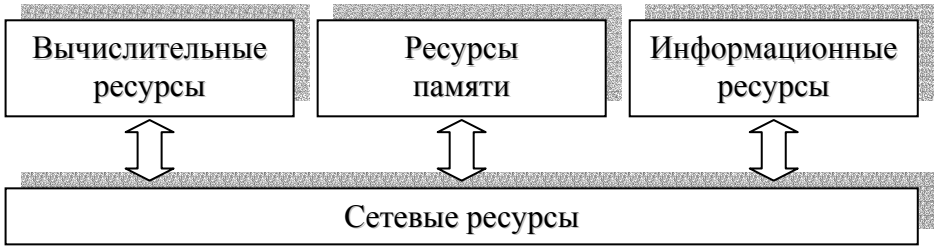


Рис. 14.2. Ресурсы Грид

Вычислительные ресурсы предоставляют пользователю Грид-системы (точнее говоря, задаче пользователя) процессорные мощности. Вычислительными ресурсами могут быть как кластеры, так и отдельные рабочие станции. При всем разнообразии архитектур любая вычислительная система может рассматриваться как потенциальный вычислительный ресурс Грид-системы. Необходимым условием для этого является наличие специального программного обеспечения, называемого ПО промежуточного уровня (middleware), реализующего стандартный внешний интерфейс с ресурсом и позволяющего сделать ресурс доступным для Грид-системы. Основной характеристикой вычислительного ресурса является производительность.

Ресурсы памяти представляют собой пространство для хранения данных. Для доступа к ресурсам памяти также используется программное обеспечение промежуточного уровня, реализующее унифицированный интерфейс управления и передачи данных. Как и в случае вычислительных ресурсов, физическая архитектура ресурса памяти не принципиальна для Грид-системы, будь то жесткий диск на рабочей станции или система массового хранения данных на сотни терабайт. Основной характеристикой ресурса памяти является его объем.

Информационные ресурсы и каталоги являются особым видом ресурсов памяти. Они служат для хранения и предоставления метаданных и информации о других ресурсах Грид-системы. Информационные ресурсы позволяют структурированно хранить огромный объем информации о текущем состоянии Грид-системы и эффективно выполнять задачи поиска.

Сетевой ресурс является связующим звеном между распределенными ресурсами Грид-системы. Основной характеристикой сетевого ресурса является скорость передачи данных. Географически распределенные системы на основе рассматриваемой технологии способны объединять тысячи ресурсов разного типа, независимо от их географического положения.

Связывающий уровень

Уровень связи (Connectivity Layer) определяет коммуникационные протоколы и протоколы аутентификации. Коммуникационные протоколы обеспечивают обмен данными между компонентами базового уровня. Протоколы аутентификации, основываясь на коммуникационных протоколах, предоставляют криптографические механизмы для идентификации и проверки подлинности пользователей и ресурсов.

Протоколы уровня связи должны обеспечивать надежный транспорт и маршрутизацию сообщений, а также присвоение имен объектам сети. Несмотря на существующие альтернативы, сейчас протоколы уровня связи в Грид-системах предполагают использование только стека протоколов TCP/IP, в частности: на сетевом уровне – IP и ICMP, транспортном уровне – TCP, UDP, на при-

кладном уровне – HTTP, FTP, DNS, RSVP. Учитывая бурное развитие сетевых технологий, в будущем уровень связи, возможно, будет зависеть и от других протоколов.

Для обеспечения надежного транспорта сообщений в Грид-системе должны использоваться решения, предусматривающие гибкий подход к безопасности коммуникаций (возможность контроля над уровнем защиты, ограничение делегирования прав, поддержка надежных транспортных протоколов). В настоящее время эти решения основываются как на существующих стандартах безопасности, изначально разработанных для Интернет (SSL, TLS), так и на новых разработках.

Ресурсный уровень

Ресурсный уровень (Resource Layer) построен над протоколами коммуникации и аутентификации уровня связи архитектуры Грид**.** Ресурсный уровень реализует протоколы, обеспечивающие выполнение следующих функций:

- согласование политик безопасности использования ресурса;
- процедура инициации ресурса;
- мониторинг состояния ресурса;
- контроль над ресурсом;
- учет использования ресурса.

Протоколы этого уровня опираются на функции базового уровня для доступа и контроля над локальными ресурсами. На ресурсном уровне протоколы взаимодействуют с ресурсами, используя унифицированный интерфейс и не различая архитектурные особенности конкретного ресурса. Различают два основных класса протоколов ресурсного уровня:

- **информационные протоколы,** которые получают информацию о структуре и состоянии ресурса, например, о его конфигурации, текущей загрузке, политике использования;
- **протоколы управления,** которые используются для согласования доступа к разделяемым ресурсам, определяя требования и допустимые действия по отношению к ресурсу (например, поддержка резервирования, возможность создания процессов, доступ к данным). Протоколы управления должны проверять соответствие запрашиваемых действий политике разделения ресурса, включая учет и возможную оплату. Они могут поддерживать функции мониторинга статуса и управления операциями.

Список требований к функциональности протоколов ресурсного уровня близок к списку для базового уровня архитектуры Грид. Добавилось лишь требование единой семантики для различных операций с поддержкой системы оповещения об ошибках.

Коллективный уровень

Коллективный уровень (Collective Layer) отвечает за глобальную интеграцию различных наборов ресурсов, в отличие от ресурсного уровня, сфокусированного на работе с отдельно взятыми ресурсами. В коллективном уровне различают общие и специфические (для приложений) протоколы. К общим протоколам относятся, в первую очередь, протоколы обнаружения и выделения ресурсов, системы мониторинга и авторизации сообществ. Специфические протоколы создаются для различных приложений Грид, (например, протокол архивации распределенных данных или протоколы управления задачами сохранения состояния и т.п.).
Компоненты коллективного уровня предлагают огромное разнообразие методов совместного использования ресурсов. Ниже приведены функции и сервисы, реализуемые в протоколах данного уровня:

- сервисы каталогов позволяют виртуальным организациям обнаруживать свободные ресурсы, выполнять запросы по именам и атрибутам ресурсов, таким как тип и загрузка;
- сервисы совместного выделения, планирования и распределения ресурсов обеспечивают выделение одного или более ресурсов для определенной цели, а также планирование выполняемых на ресурсах задач;
- сервисы мониторинга и диагностики отслеживают аварии, атаки и перегрузку.
- сервисы дублирования (репликации) данных координируют использование ресурсов памяти в рамках виртуальных организаций, обеспечивая повышение скорости доступа к данным в соответствии с выбранными метриками, такими как время ответа, надежность, стоимость и т.п.;
- сервисы управления рабочей загрузкой применяются для описания и управления многошаговыми, асинхронными, многокомпонентными заданиями;
- службы авторизации сообществ способствуют улучшению правил доступа к разделяемым ресурсам, а также определяют возможности использования ресурсов сообщества. Подобные службы позволяют формировать политики доступа на основе информации о ресурсах, протоколах управления ресурсами и протоколах безопасности связывающего уровня;
- службы учета и оплаты обеспечивают сбор информации об использовании ресурсов для контроля обращений пользователей;
- сервисы координации поддерживают обмен информацией в потенциально большом сообществе пользователей.

Прикладной уровень

Прикладной уровень (Application Layer) описывает пользовательские приложения, работающие в среде виртуальной организации. Приложения функционируют, используя сервисы, определенные на нижележащих уровнях. На каждом из уровней имеются определенные протоколы, обеспечивающие доступ к необходимым службам, а также прикладные программные интерфейсы (Application Programming Interface – API), соответствующие данным протоколам.

Для облегчения работы с прикладными программными интерфейсами пользователям предоставляются наборы инструментальных средств для разработки программного обеспечения (Software Development Kit – SDK). Наборы инструментальных средств высокого уровня могут обеспечивать функциональность с одновременным использованием нескольких протоколов, а также комбинировать операции протоколов с дополнительными вызовами прикладных программных интерфейсов нижнего уровня.

Обратим внимание, что приложения на практике могут вызываться через достаточно сложные оболочки и библиотеки. Эти оболочки сами могут определять протоколы, сервисы и прикладные программные интерфейсы, однако подобные надстройки не относятся к фундаментальным протоколам и сервисам, необходимым для построения Грид-систем.

Работа грид-систем опирается на программное обеспечение промежуточного уровня: программные компоненты и протоколы, которые обеспечивают требуемый контролируемый доступ к ресурсам. На первом этапе своего существования грид-системы строились или на основе специально разработанных общедоступных компонент или на основе закрытых (проприетарных) технологий. Хотя различные общественные и коммерческие решения были успешны в своих областях применения грида, каждое со своими сильными и слабыми сторонами, они имели ограниченный потенциал как основы для грида нового поколения, который должен быть масштабируемым и интероперабельным, чтобы удовлетворять потребности широкомасштабных научных и производственных проектов.

В последние годы стало ясно, что есть значительное перекрытие между целями вычислительного грида и преимуществ сред, основанных на веб-сервисах. Быстрый прогресс в технологии веб-сервисов и разработке соответствующих стандартов обеспечили эволюционный путь от жесткой и узко-направленной архитектуры грид- систем первого поколения к стандартизированным, сервис-

ориентированным гридам, гарантирующим стабильно-высокое качество обслуживания пользователей (грид промышленного уровня).

14.3. WebServices в Грид

В Грид используются основные элементы технологии WebServices. Основу ее составляют:

- Система WWW-адресации URL
- Протокол передачи гипертекста HTTP
- Язык гипертекстовой разметки HTML и расширяемый язык разметки XML
- протоколы **SOAP** (Simple Object Access Protocol) для управления сообщениями в универсальном XML-формате;
- язык **WSDL** (Web Services Definition Language) описания интерфейса взаимодействия компонент распределенной системы.

Это позволяет функции Грид технологии (служебные и ресурсные) оформить единообразно в виде веб-сервисов, используя для работы сними все стандарты технологии WebServices.

На рис.14.3 показана схематично представлен простой сервисно-ориентированный грид, в котором сервисы используется и для виртуализации ресурсов, и для обеспечения других функциональных возможностей грида.

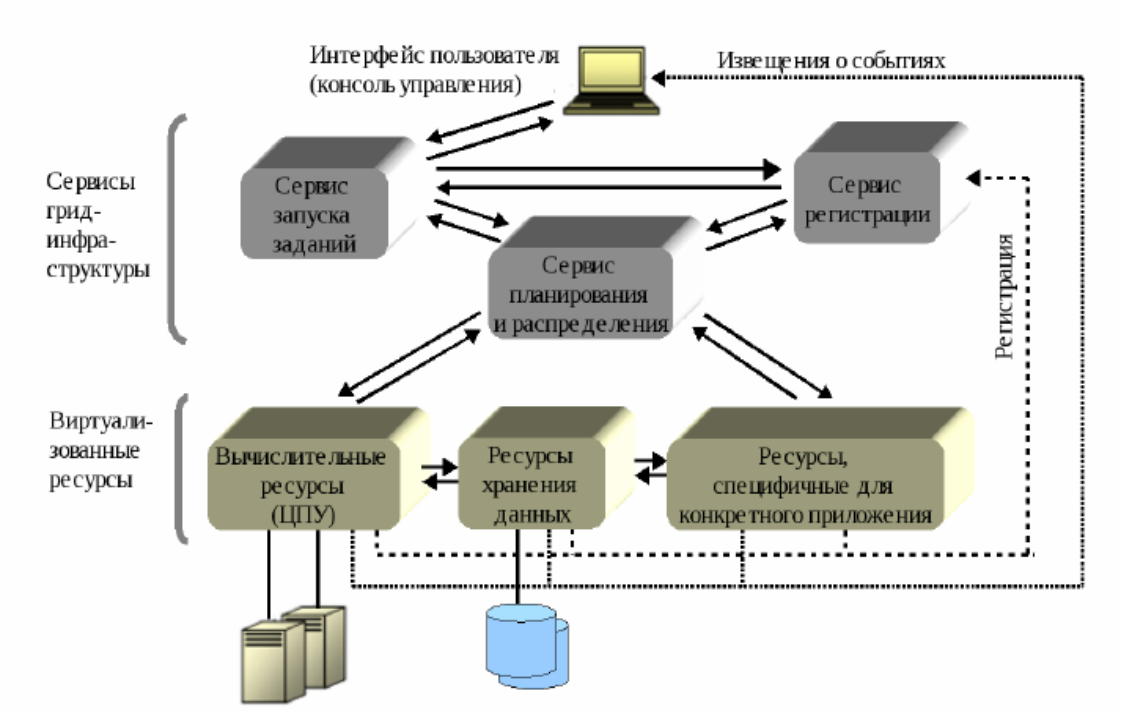


Рис. 14.3 упрощенная схема сервисно- ориентированного грида.

На схеме показана единая консоль и для запуска заданий в грид-среду, и для управления грид-ресурсами. Программное обеспечение интерфейса пользователя (консоли) обращается к сервису регистрации, чтобы получить информацию о существующих грид- ресурсах. Затем пользователь посредством консоли входит в контакт с сервисами, «представляющими» (виртуализующими) каждый ресурс, чтобы запросить периодическое получение данных о работе ресурсов и получение извещений о существенных изменениях в их состоянии (например, если ресурс становится недоступным или сильно загруженным).

Пользователь направляет запрос на запуск задания в службу запуска, которая передает запрос службе распределения заданий (часто называемой планировщиком). Служба распределения контактирует со службой, представляющей приложение, и запрашивает информацию о требованиях к ресурсам для выполнения задания. Затем служба распределения запрашивает у службы регистрации информацию о всех подходящих ресурсах в гриде и напрямую контактирует с ними, чтобы убедиться в их доступности. Если подходящие ресурсы доступны, планировщик выбирает наилучшую доступную совокупность ресурсов и передает информацию о их сервису приложения с запросом на начало выполнения. В противном случае планировщик ставит задание в очередь и выполняет его, когда необходимые ресурсы становятся доступными. Когда выполнение задания заканчивается, сервис приложения сообщает о результате планировщику, который извещает об этом сервис запуска заданий. Сервис запуска заданий, в свою очередь, уведомляет пользователя.

Отметим, что этот пример для ясности сильно упрощен: функционирование реального грида промышленного уровня является намного более сложным чем показано на схеме. Главным результатом его работы должна быть высокая степень автоматизации и оптимизации использовании ресурсов в рамках грид-среды.

В среде Грид явным образом присутствуют следующие элементы:

- Программы пользователя
- Ресурсы (в конечном счете - железом, ОС, кластерным ПО и т.п.)
- Промежуточное программное обеспечение (**Middleware**) , выступающее в роли посредника пользовательскими программами и ресурсами. Middleware включает большой объем прогамного обеспечения, разрабатывается большими организациями и строго стандартизуется, чтобы обеспечить взаимно перекрестное использование частей этого **Middleware** разными разработчиками.

В число наиболее известных пакетов **middleware** входят:

- Globus Toolkit. Условно его можно назвать американским направлением, разрабатываеся разрабатываеися многими организациями, в частности по проекту Globus, котороый возник в Аргоннской национальной лаборатории. Здесь разработаны пакеты GT1,GT2, GT3 и самый совершенный - GT4.
- gLITE. Его условно можно назвать европейским проектом, поскольку его разработка курируется Европейским центром ядерных исследований (CERN). gLITE также явился следствием ряда предыдущих проектов (EDG, LCG и др.).

Различные **middleware** - системы могут различаться по уровню, к примеру, gLITE ставится "поверх" Globus Toolkit.

Очевидно, что для развертывания Грид инфраструктуры необходимо использовать последние версии программного обеспечения, поддерживащие новейшие стандарты, поэтому дальше будут рассмотрены две последние версии middleware: GT4 и gLite.

Лекция 15. Пакет Globus Toolkit

15.1. Состав G T4

В глобальных Грид-системах в качестве средства middleware используют инструментарий Globus Toolkit, разработанный американскими учеными, который стал de facto мировым стандартом. Он включает в себя, в частности, специальный протокол на основе HTTP для использования вычислительных ресурсов GRAM (Grid Resource Allocation Management); расширенную версию протокола для передачи файлов GridFTP; службу безопасности GSI (Grid Security Infrastructure); распределенный доступ к информации на основе протокола LDAP; удаленный доступ к данным через интерфейс GASS (Globus Access to Secondary Storage). Эти службы позволяют построить полнофункциональную Грид систему. Общая структура пакета G T4 представлена на рис.15.1.

Рис. 15.1. Состав пакета GT4

Пакет содержит следующие разделы:

1. Security - Обеспечение безопасности
2. Data Management - Управление данными
3. Execution Management - Управление исполнением заданий
4. Information Service – Информационный сервис
5. Common Runtime –Фундаментальных библиотеки и средств, используются для создания Web-служб и не-Web-служб.

15.2. Обеспечение безопасности

Компонента **GSI (Globus Security Infrastructure)** обеспечивает защиту, включающую шифрование данных, а также аутентификацию (проверка подлинности, при которой устанавливается, что пользователь или ресурс действительно является тем, за кого себя выдает) и авторизацию (процедура проверки, при которой устанавливается, что аутентифицированный пользователь или ресурс действительно имеет затребованные права доступа) с использованием цифровых сертификатов X.509. Можно с полной уверенностью говорить что при внедрении и развертывании Грид систем инфраструктура безопасности является наиболее важным звеном т.к. от нее напрямую зависит безопасность всей системы.

На рис.15.2 инфраструктура безопасности показана в действии. Запрос был поставлен таким образом: “Создать процессы на узлах A и B, которые затем обмениваются файлами с узлом C.”

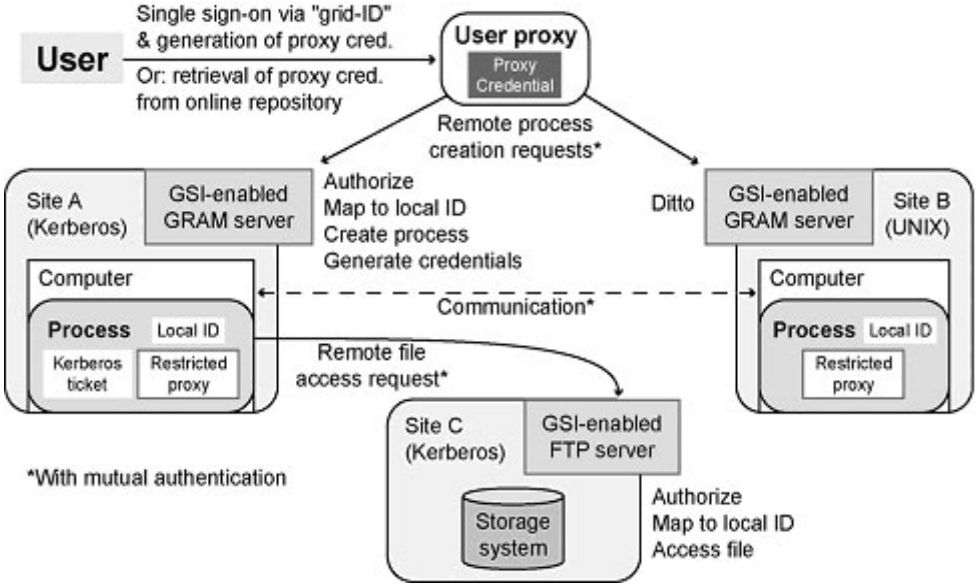


Рис.15.2. Инфраструкткра безопасности в действии.

GSI установлена на большом количестве сайтов. Практически все исследовательские и академические Грид системы используют GSI. Процесс стандартизации для GSI начался с Global Grid Forum.

Свойства системы безопасности. Ими являются: конфиденциальность, целостность и аутентификация. В идеале, защищенный обмен включает все три составляющие, но это нужно не всегда (иногда даже нежелательно).

Конфиденциальность. Безопасный диалог должен быть конфиденциален. Другими словами, только отправитель и получатель должны понимать содержание обмена. Если кто-то ведет прослушивание обмена, он не должен обнаружить в диалоге какой-либо смысл. В общем случае это достигается алгоритмом шифрования-дешифрования.

Целостность. Безопасные коммуникации должны гарантировать целостность передавамых сообщений. Это означает, что на приемном конце должны быть уверены, что принятое сообщение есть в действительности точно то, которое было послано. Надо принимть во внимание, что злоумышленники могут перехватывать сообщение с целью модификации его содержания, а не только с целью его прослушивания. Злоумышленник может изменить сообщение, даже не понимая его смысла.

Аутентификация. Безопасная коммуникация должна гарантировать, что стороны, участвующие в обмене, есть те, за которые они себя выдают. Это относительно легко сделать с помощью некоторых сетевых свойств.

Авторизация. Другой важной концепцией в компьютерной безопасности, хотя обычно не рассматриваемой как одна из основ безопасных коммуникаций, является авторизация. Проще говоря, авторизация относится к механизмам, которые решают, разрешено ли пользователю выполнять некоторую задачу. Авторизация связана с аутентификацией, потому что в общем случае мы должны убедится, что пользователь есть тот, за кого он себя выдает, прежде чем решать, допускать его к некоторой работе или нет.

Криптография с открытым ключом. Обеспечение безопасности в ГРИД является одной из главных забот разработчиков. Во всяком случае в документации по GRID вопросам безопасности отводится от 20 до 30 %.

Системы с одним ключом недопустимы в ГРИД, поскольку, если А хочет передать В или кому-то еще закодированные данные, то он должен передать и ключ, который тем самым становится доступным злоумышленнику. В ГРИД используются система шифрования с открытым ключом, в которой не требуется передавать ключ по каналу.

Система шифрования на основе открытого ключа строится следующим образом. Выбирается большое простое число p (не менее 512 бит) и его первообразный корень $a<p$. Каждый пользователь выбирает случайный секретный ключ x и вырабатывает открытый ключ y по формуле:

$$y=a^x \pmod p. \qquad (15.1)$$

Для любого значения x легко вычислить y, однако, вычислительно неосуществимо выполнение дискретного логарифмирования, а следовательно и определение числа x, для которого значение $a^x \bmod p$ равно заданному значению y.

Открытые ключи хранятся в электронных центрах сертификации (ЦС). ЦС хранит множество записей типа:

Таб.1 – Открытые ключи

Имя	Открытый ключ
А	$У_А$
В	$У_В$
С	$У_С$
…..	…..

Пользователь А, посылая сообщение В, формирует закрытый ключ вида

$$Z_{ab} = (y_B)^{x_A} = (\alpha^{x_B})^{x_A} = \alpha^{x_B x_A} \qquad (15.2)$$

Здесь y_B – открытый ключ В из ЭКС. В свою очередь, В, получив сообщение от А, также формирует секретный ключ

$$Z_{ba} = (y_A)^{x_B} = (\alpha^{x_A})^{x_B} = \alpha^{x_A x_B} \qquad (15.3)$$

Поскольку $Z_{ab} = Z_{ba}$, то оба абонента имеют общий открытый ключ, полученный без его передачи по открытой сети. Таким образом, выражения (1), (2), (3) определяют алгоритм криптографии с открытым ключом.

Алгоритм с открытым ключом является ассиметричным и, вследствие этого, использует два различных ключа вместо одного. В алгоритм с открытым ключом закрытый ключ известен только его владельцу, а открытый ключ известен всем, поскольку он публикуется в общедоступном справочнике.

В случае безопасного обмена с открытым ключом отправитель шифрует сообщение, используя открытый ключ получателя. Зашифрованное сообщение посылается получателю, который бу-

дет дешифровать сообщение с помощью своего закрытого ключа. Только получатель может расшифровать сообщение, поскольку ни у кого нет его закрытого ключа. Заметим, что алгоритм дешифрации одинаков на обоих концах.

Системы с открытым ключом имеют явное преимущество перед симметричными системами: не нужно передавать общий для обоих сторон ключ. Другое важное преимущество состоит в том, что системы с открытым ключом могут гарантировать не только конфиденциальность, но и аутентифиацию и целостность сообщений.

Компоненты системы безопасности в ГРИД. Безопасность в ГРИД обеспечивается многими составляющими.

Цифровая подпись. Целостность в системах с открытым ключом гарантируется благодаря цифровой подписи. Цифровая подпись есть блок данных, который присоединяется к сообщению и который может быть использован, чтобы выяснить, была ли попытка вмешательства в сообщение во время преобразования.

Цифровая подпись для сообщения генерируется в два шага:

- Сначала генерируется цифровая сводка сообщения. Это некоторая «сумма» передаваемого сообщения. Она имеет два важных свойства: она всегда меньше сообщения и даже малейшее изменение в сообщении дает другую сводку.
- Сводка сообщения шифруется закрытым ключом отправителя. Результирующая зашифрованная сводка сообщения называется цифровой подписью.

Цифровая подпись подсоединяется к сообщению и посылается получателю. Получатель далее делает следующее:

- Используя открытый ключ отправителя, дешифрует цифровую подпись.
- Использует тот же самый алгоритм шифрования цифровой сводки, что и отправитель, чтобы получить цифровую сводку полученного сообщения.
- Сравнивает обе сводки сообщения. Если они не совпали в точности, то сообщение было подвергнуто вмешательству третьей стороны. Можно быть уверенным, что цифровая подпись была послана отправителем, поскольку она шифровалась его закрытым ключом.

Таким же способом гарантируется и целостность.

Аутентификация в системах с открытым ключом. Приведенный выше пример в некоторой степени гарантирует аутентичность отправителя, поскольку только открытый ключ отправителя может дешифровать цифровую подпись. Однако, возможно, реально отправитель не тот, за котрого он себя выдает.

Во многих случаях «слабой аутентификации» достаточно. Но иногда нужна аутентификация, при которой нет сомнения в отправителе. Это достигается на основе цифровых сертификатов.

Цифровой сертификат (рис.3) есть цифровой документ, который свидетельствует, что определенный открытый ключ принадлежит владельцу закрытого ключа. Этот документ подписывается третьей стороной, называемой certificate authority (CA). Формат CA представлен на рис.15.3.

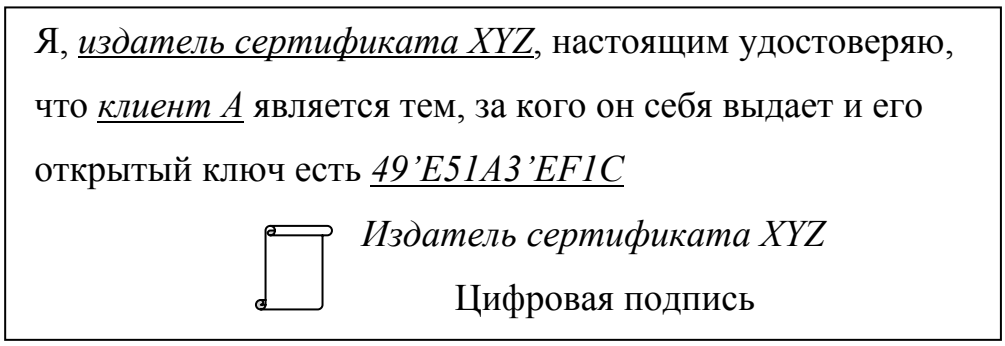


Рис.15.3. Формат цифрового сертификата.

Естественно, сертификат кодируется в цифровом формате. Подпись издателя есть в действительности цифровая подпись, сгенерированная с помощью закрытого ключа CA, поэтому мы можем проверить целостность сертификата, используя открытый ключ CA.

Наличие сертификата позволяет выполнить аутентификацию. Если вы подписываете ваше сообщение вашим закрытым ключом и посылаете получателю копию вашего сертификата, он может быть уверен, что сообщение было послано вами.

X.509 формат. Сертификат есть текстовый файл, который включает много информации, представленной в специфическом синтаксисе. X.509 имеет следующие четыре наиболее важные вещи:

- *Subject*: это «имя» пользователя. Оно кодируется как уникальное имя.
- *Subject's public key*: Это включает не только сам ключ, но и некоторую информацию, например, какой алгортм использован для генерации открытого ключа.
- *Issuer's Subject:* уникальное имя CA.
- *Digital Signature*: Сертификат включает цифровую подпись всей информации в сертификате. Цифровая подпись генерируется с использованиемоткрытого ключа CA.

CA иерархия. Системы с открытым ключом будут иметь список всех CA, которым можно доверять, включая и ваш CA. Вы должны решить, кто сделает такой список. Кто подписывает CA сертификаты? Ответ – другой CA. Это позволяет создать иерархию CA, в которой вы можете доверять самому верхнему CA (рисунок 10). Вы можете по ступеням иерархии проверить все CA, кроме последнего, которому вам приходится доверять абсолютно.

Сертификат - открытый документ, который доступен другим пользователям для проверки вашей идентичности. Объединение открытого и закрытого ключей называется в общем случае мандатом (credentials) пользователя.

Система входа пользователя в -систему достаточно сложна. Это определено многими факторами, но главной проблемой является решение вопросов безопасности (угрозы вторжений и атак злоумышленников). Аутентификация и авторизация пользователей являются путями для решения этой проблемы. Аутентификационные решения для сред виртуальных организаций должны обладать следующими свойствами:

- Единый вход. Пользователь должен регистрироваться и аутентифицироваться только один раз в начале сеанса работы, получая доступ ко всем разрешенным ресурсам базового уровня архитектуры Грид.
- Делегирование прав. Пользователь должен иметь возможность запуска программ от своего имени. Таким образом, программы получают доступ ко всем ресурсам, на которых авторизован пользователь. Пользовательские программы могут, при необходимости, делегировать часть своих прав другим программам.
- Доверительное отношение к пользователю. Если пользователь запросил одновременную работу с ресурсами нескольких поставщиков, то при конфигурации защищенной среды пользователя система безопасности не должна требовать взаимодействия поставщиков ресурсов друг с другом.

Для входа в Грид-систему пользователь должен:

- быть легальным пользователем вычислительных ресурсов в своей организации;
- иметь персональный цифровой сертификат, подписанный центром сертификации;
- быть зарегистрированным хотя бы в одной виртуальной организации.

15.3. Управление данными

Большая часть существующих систем обеспечивают передачу: большого количества данных, типы данных различны, часто часто передача выполняется в режиме реального времени. Возможность доступа и управления такими данными - это другая важная область в вычислительных сетях. Это включает перемещение данных между различными ресурсами, возможность ресурсу иметь доступ к данным по требованию.

Элементами системы управления данными являются:

GridFTP. Стандартный FTP протокол был расширен возможностями параллельной пересылки данных, возможностями разделения файлов, автоматической и ручной настройки размеров буферов, мониторинга состояния передаваемых данных, возможностями автоматической повторной передачи в случае возникновения ошибок.

Reliable File Transfer. Надстройка над протоколом GridFTP позволяющая работать с использованием web сервисов.

Replica Location Service. Утилита, позволяющая хранить информацию о файлах или копиях файлов в системе. Представляет собой реестр в котором содержится информация о физическом расположении файлов и их реплицированных копий. Реестр может быть разделен на несколько машин в сети, что обеспечивает устойчивость к сбоям оборудования.

Data Replication Service. Data Replication Service - сервис более высокого уровня чем RFT и RLS. Он объединяет в себе эти компоненты. Задачей сервиса является проверка на наличие некоторого набора файлов на сервере. В случае негативного ответа он должен отыскать файлы посредством запроса к сервису RLS и инициировать передачу файлов через RFT.

15.4. Управление исполнением заданий

Архитектура средств управления ресурсами **GRMA** (Globus Resource Management Architecture) имеет многоуровневую структуру (рис.15.4).

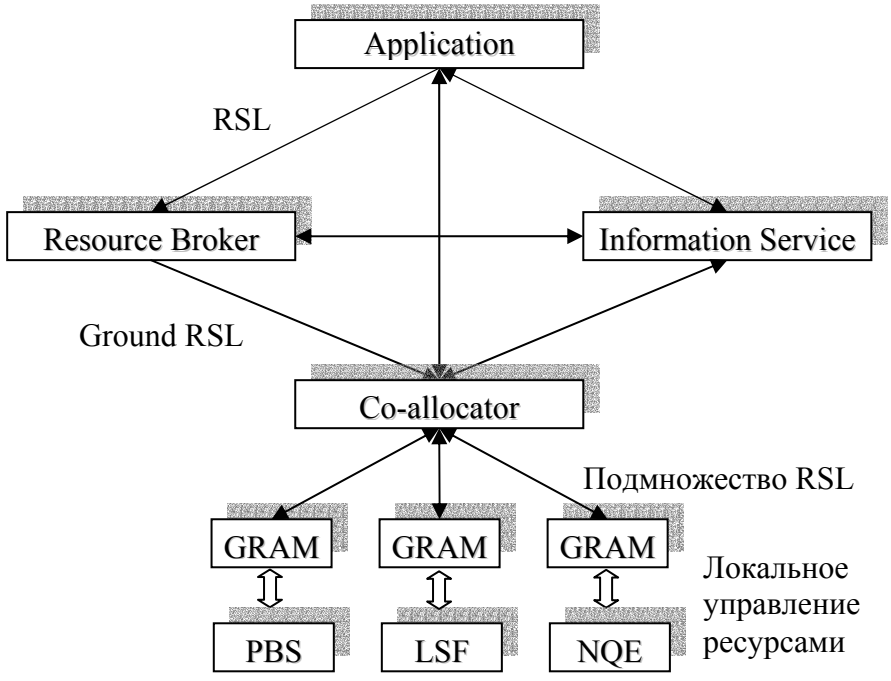


Рис.15.4. Структукра GRMA

Запросы пользовательских приложений выражаются на RSL и передаются брокеру ресурсов, который отвечает за высокоуровневую координацию использования ресурсов (балансировку загрузки) в определенном домене. На основе переданного пользовательским приложением запроса и политики (права доступа, ограничения по использованию ресурсов) ответственного административного домена брокер ресурсов принимает решение о том, на каких вычислительных узлах будет выполняться задача, какой процент вычислительной мощности узла она может использовать и др.

При выборе вычислительного узла брокер ресурсов должен определить, какие узлы доступны в текущий момент, их загрузку, производительность и другие параметры, указанные в RSL-запросе, выбрать наиболее оптимальный вариант (это может оказаться один вычислительный узел или несколько), сгенерировать новый RSL-запрос (ground RSL) и передать его высокоуровневому менеджеру ресурсов (co-allocator)**.** Этот запрос будет содержать уже более конкретные данные, такие, как имена конкретных узлов, требуемое количество памяти и др.

Основные функции высокоуровневого менеджера ресурсов GRMA перечислены ниже:

- коллективное выделение ресурсов;
- добавление/удаление ресурсов к ранее выделенным;
- получение информации о состоянии задач;
- передача начальных параметров задачам.

GRMA производит декомпозицию запросов ground RSL на множество простых RSL запросов и передает эти запросы GRAM. Далее задача пользователя запускается на исполнение. В случае, если один из GRAM возвращает ошибку, задача либо снимается с выполнения, либо запускается повторно.

Высокоуровневый менеджер ресурсов производит декомпозицию запросов *ground RSL* на множество более простых RSL-запросов и передает эти запросы GRAM. Далее, при отсутствии сообщений об ошибках от GRAM, задача пользователя запускается на исполнение. В случае, если один из GRAM возвращает ошибку, задача либо снимается с выполнения, либо попытка запуска производится повторно.

Менеджер GRAM (рис.15.5) предоставляет верхним уровням универсальный API для управления ресурсами узла Грид. Сам GRAM взаимодействует с локальными средствами управления ресурсами узла. Узлом может быть, например, рабочая станция или вычислительный кластер.

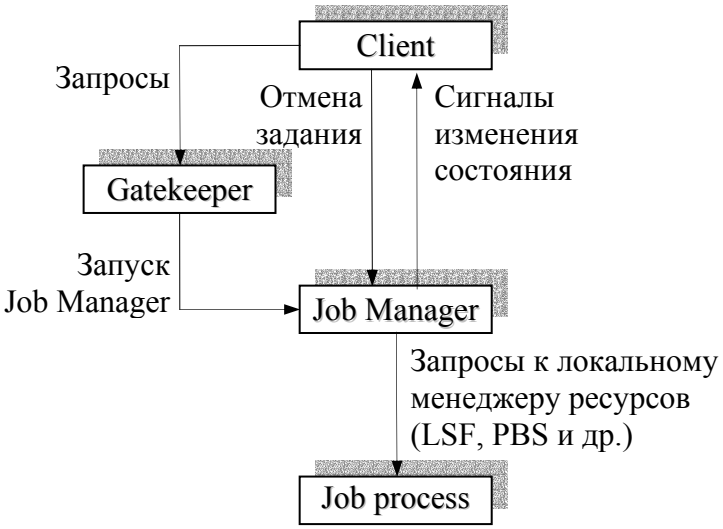


Рис.15.5. GRAM

GRAM – достаточно низкоуровневый компонент Globus Toolkit, являющийся интерфейсом между высокоуровневым менеджером ресурсов и локальной системой правления ресурсами узла.

В настоящее время этот интерфейс может взаимодействовать со следующими локальными системами управления ресурсами:

- PBS (Portable Batch System) – система управления ресурсами и загрузкой кластеров. Может работать на большом числе различных платформ: Linux, FreeBSD, NetBSD, Digital Unix, Tru64, HP-UX, AIX, IRIX, Solaris. В настоящее время существует свободная и обладающая более широкими возможностями реализация PBS, называемая Torque.
- Condor – свободно доступный менеджер ресурсов, разработанный в основном студентами различных университетов Европы и США. Аналогичен вышеперечисленным. Работает на различных платформах UNIX и Windows NT и многими другими планировщиками.

Чтобы на данном вычислительном узле можно было удаленно запускать на исполнение программы, на нем должен выполняться специальный процесс называемый Gatekeeper. Gatekeeper работает в привилегированном режиме и выполняет следующие функции:

- производит взаимную аутентификацию с клиентом;
- анализирует RSL запрос;
- отображает клиентский запрос на учетную запись некоторого локального пользователя;
- запускает от имени локального пользователя специальный процесс, называемый Job Manager, и передает ему список требующихся ресурсов.

После того, как Gatekeeper выполнит свою работу, Job Manager запускает задание (процесс или несколько процессов) и производит его дальнейший мониторинг, сообщая клиенту об ошибках и других событиях. Gatekeeper запускает только один Job Manager для каждого пользователя, который управляет всеми заданиями данного пользователя. Когда заданий больше не остается, Job Manager завершает работу.

Организация GRAM. Мы кратко опишем реализацию GRAM. Эта информация поможет пользователю понять принципы управления GRAM (рис.15.6).

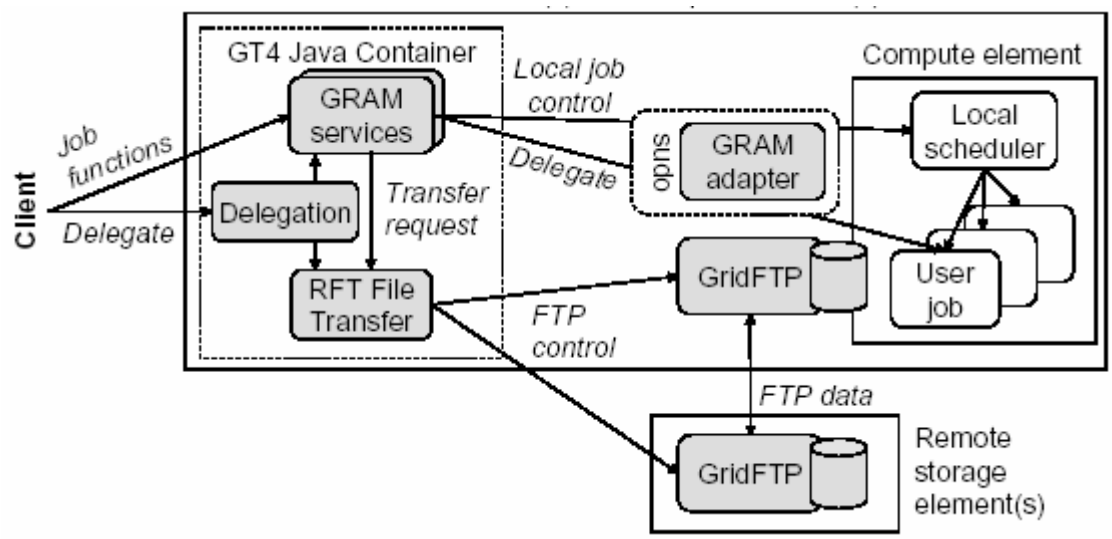


Рис.15.6 - Компоненты GRAM

Основными элементами GRAM являются:

- GRAM специфические службы для создания, наблюдения и управления работами.
- Общецелевые службы для передачи прав и надежной передачи данных RFT.
- Планирующие GRAM адаптеры, предназначенные для передачи запросов в соответствующие локальные планировщики.
- GridFTP серверы для выполнения конвейерных команд.

- Программы командной строки (Таб.2). Интерфейсы пользователя позволяют вызвать компоненты непосредственно из программы пользователя, написанной на Java, C, and Python.

Таб. 2 - Каждая строка представляет ряд программ

Команды	Описание
cas-*	Общая служба авторизации
globus-credential-*	Инсталляция и восстановление прав в службе передачи прав
myproxy-*	Служба MyProxy
grid-ca-sign	Простой издатель для генерации X.509 прав
gsi*	OpenSSH с ssh, scp, stpf клиентами
grid-*	УправлениеX.509 прокси мандатами и файлами размещения
globus-url-copy	GridFTP клиент
rft*	Клиент надежной передачи файлов
globus-rls-*	Клиент RLS, администрация, сервер
globusrun-ws	Клиент GRAM
mds-servicegroup-add	Добавление входа в MDS группу служб
globus-*-container	Запуск и остановка контейнера Globus
wsrf-*	Взаимодействие с WSRF свойствами ресурсов
wsn-*	Создание и управление WS-Notification subscriptions.

Команды задания работы в GRAM. В качестве первого примера мы представим команду для исполнения программы “/bin/touth” с аргументом “touched_it”, которая выглядит следующим образом:

```
% globusrun-ws -submit -job-command /bin/touch touched_it
```

Компоненты этой команды задают название команды; флаг задания, указывающий, что это задание; флаг работы, указывающий, что остаток стороки команды задает имя программы и аргумент. Если нет ошибок, то задание будет принято и программа запущена. Затем задание заканчивается и выводится информация о статусе:

```
Job ID: uuid:c51fe35a-4fa3-11d9-9cfc-000874404099
Termination time: 12/17/2004 20:47 GMT
Current job state: Active
Current job state: CleanUp
Current job state: Done
Destroying job...Done.
```

После того, как работа запущена, пользователь остается постоянно подключенным к ней. Это необходимо, например, как для получения статусной информации, так и для того, чтобы прекратить исполнение задания.

Интерактивный режим полезен во многих обстоятельствах, например, в пакетном режиме пользователь по своему усмотрению в любой момент может запросить статусную информацию. Виды статусной информации указаны в таб.3.

Таб. 3 - Состояния работ в GRAM

Состояние	Описание
Unsubmitted	Работа еще не задана
StageLn	Работа ожидает этапа исполнения или входных файлов для окончания
Pending	Локальный планировщик еще не спланировал работу на исполнение
Active	Работа выполняется
Suspended	Выполнение работы было приостановлено
StageOut	Выполнение работы завершено, выходные файлы выведены
CleanUp	Выполнение работы завершено, выполняется очистка задач
Done	Работа завершена успешно
Failed	Работа потерпела неудачу

До сих пор предполагалось, что исполняемая программа расположена на исполнительном копьютере, входные данные передаются как аргумент и результаты сохранены в файлах на исполнительном копьютере. Однако, бывают ситуации, когда исполняемую программу или другие файлы нужно передать в целевой компьютер или получить от него результат. Для этого в описании задания используются специальные директивы файловых передач, которые основаны на RFT синтаксисе, в них указываются адреса передающей и принимающей сторон (URL). Пример задания на передачу файлов приведен ниже.

```
<fileStageIn>
  <transfer>
    <sourceUrl>gsiftp://submit.host:2888/tmp/mySourceFile</sourceUrl>
    <destinationUrl>file:///${GLOBUS_USER_HOME}/my_file</destinationUrl>
  </transfer>
</fileStageIn>
```

Существует много форматов, созданных различными разработчиками, поэтому за структурой RSL надо обращаться к документации конкретного разработчика.

Планирование MPI вычислений в системе Condor

Condor – одна измногих специализированная пакетная система для управления выполнением интенсивных компютерных работ, которая может использоваться под **GRAM**. Как и другие системы подобного рода, Condor обеспечивает механизм очередности, политику планирования, схему приоритетов и классификацию ресурсов. Пользователь задает компьютерную работу для Condor d RSL, Condor ставит работу в очередь, запускает ее и затем информирует пользователя о результатах. Порядок выполнения работы таков:

Подготовка кода. Работа для Condor должна быть пригодной для выполнения в фоновом пакетном режиме. Программа, которая выполняется в фоновом режиме, не пригодна для интерактивного ввода-вывода. Condor может перенаправить консольный вывод и ввод с клавиатуры в/из файла. Для этого создайте нужный файл, который содержит необходимые строки для программного входа. Тогда определенная программа с этим файлом будет выполняться правильно.

Среды Condor. Condor имеет несколько сред для времени исполнения (Universe): Standard, Vanilla, PVM, MPI, Globus, Java, Scheduler. Пользователь должен выбрать одну из них.

Задание файла описания. Этот файл содержит такие сведения, как имя программы, которая будет выполняться, файлы для использования данных экрана и клавиатуры, тип платформы, требуемой для выполняемой программы и много другой информации.

Задание работы. Работа для Condor задается командой condor_submit. После этого Condor сомостоятельно выполняет действия по запуску и выполнению задания.

Работа со средой MPI. MPI является средой программирования для кластеров с индивидуальной памятью и обменом сообщениями. Она устанавливает окружение, в котором параллельные программы можно синхронизовать на основе коммуникаций. Выполнение MPI программ под Condor облегчает усилия программистов. MPI программы для работы с компилируются с использованием *mpicc.*

Condor может быть сконфигурированием так, чтобы машины для работы с MPI были выделены. Выделенные машины – это такие машины, на которых программа выполняется от начала до конца. Чтобы упростить планирование выделенных ресурсов, одна машина становится планировщиком этих ресурсов. Это ведет к дальнейшим ограничениям в том, что выделенные машины должны работать под выделенным планировщиком.

Задание работы. После того, как программа написана и скомпилирована и ресурсы Condor сконфигурированы, работа может быть задана. Каждая работа Condor требует файла описания задания. Простейший файл описания таков:

```
##############################################
## submit description file for mpi_program
##############################################
universe = MPI
executable = mpi_program
machine_count = 4
queue
```

Эта работа описывает среду как MPI, извещает Condor, что потребуются выделенные ресурсы. Команда mashine_count задает число требуемых для выполнения работы машин. Четыре машины, выбранные для работы, должны быть по умолчанию одинаковой архитектуры и с той же операционной системой.

Такм образом, задание на выполнение работы проходит следующие ступени:

1. Задание работы для Condor на RSL.
2. GRAM получает от RSL задание и инициирует работу планировщика Condor.
3. Condor планирует работу на заданных ресурсах, запускает и наблюдает за выполнением работы.

15.5. Информационный сервис

Основным элементом системы информационного сервиса является подсистема обновления и отыскания сервисов **MDS** (Monitoring and Discovery Service), в которой зарегистрированы все ресурсы Грид. Информация о ресурсах может содержать, например, как данные о конфигурации или состоянии как всей системы, так и отдельных ее ресурсов (тип ресурса, доступное дисковое пространство, количество процессоров, объем памяти, производительность и прочее). Вся информация логически организована в виде дерева, и доступ к ней осуществляется по стандартному протоколу **LDAP** (Lightweight Directory Access Protocol). Каждая запись в структуре LDAP содержит свойства отдельного ресурса. Пользовательское приложение может запросить сервер для установления типа архитектуры, поддерживаемой операционной системы, сведений о менеджере задач, которые доступны на заданном ресурсе. Каждый организационный домен, входящий в Грид, должен запустить сервис информации о ресурсах (GRIS). Функциональность данного сервиса включает в себя:

- Просмотр и получение информации, которая может быть точно сопоставлена с конкретным ресурсом или объектом.
- Возможность делать запросы и искать информацию для получения набора связанных ресурсов или объектов.
- Создание новой информации по запросу, которая не сохранена в системе.

- Переадресация событий для передачи информации основанной на динамических событиях внутри нфраструктуры Grid.
- Сбор информации для тематического получения и организации информации.
- Фильтрация информации для уменьшения ее количества и увеличения производиельности.
- Хранение, резервное копирование, кеширование информации для болеепростогопоследующего доступа к ней и для увеличения производительности.
- Безопасность, защита и шифрование для обеспечения работы важных задач, таких какконтроль доступа, аутентификация.
- Для поддержки данной функциональности пользователями **Grid** системы могут быть использованы несколько сервисов.

MDS имеет децентрализованную, легко масштабируемую структуру и работает как со статическими, так и с динамически меняющимися данными, необходимыми пользовательским приложениям и различным сервисам Грид-системы. Иерархическая структура MDS представлена на рис. 15.7.

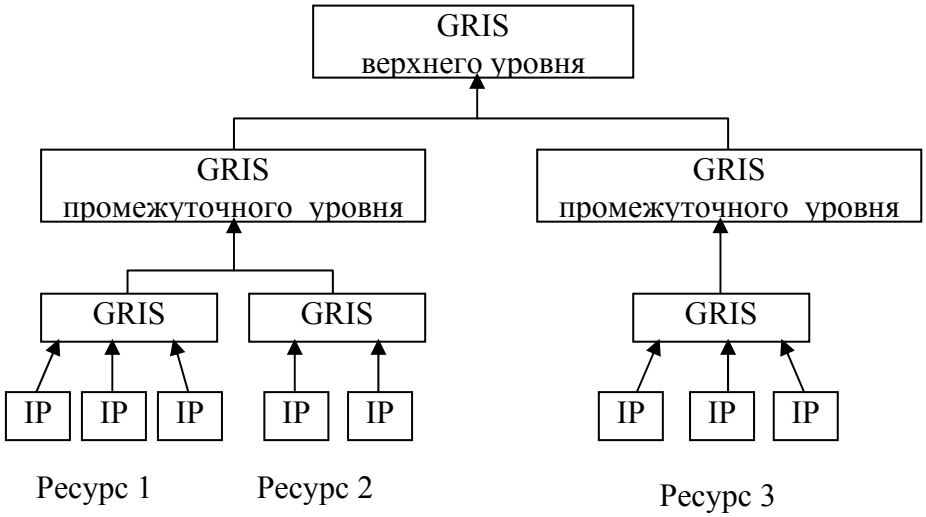


Рис. 15.7. Иерархическая структура MDS

MDS состоит из трех основных компонент:

1. IP (Information Provider) – является источником информации о конкретном ресурсе.
2. GRIS (Grid Resource Information Service) – предоставляет информацию об узле Грид-системы, который может быть как вычислительным узлом, так и каким-либо другим ресурсом. GRIS опрашивает индивидуальные IP и объединяет полученную от них информацию в рамках единой информационной схемы;

Лекция 16. Пакет gLite

gLite - европейский пакет middleware для Грид. Основное отличие пакета gLite от GT4 состоит в том, что помимо инструментальных средств, в него входит более широкий набор служб. gLite существенно опирается на опыт ряда крупных европейских проектов: EDG, LCG, Alien, Nordugrid и создается коллективно – в его разработке участвуют много исследовательских центров Европы. Все это проекты, включая gLite, имеют много общего, поскольку в той или иной степени основаны на одной базе – системе Globus Toolkit. gLite является основой проекта **EGEE** (Enabling Grids for E-SciencE – "Развёртывание гридов для е-науки"), придя на смену комплексу LCG (Large Hadron Collider Computing Грид), предназначенного для большого адронного коллайдера, строящегося в CERN для изучения фундаментальных свойств субатомных частиц и сил.

16.1. Состав gLite

Ниже перечислены все основные элементы LCG и их назначение:

CE (Computing Element) – набор программ для установки на управляющий узел вычислительного кластера. Данный элемент предоставляет универсальный интерфейс к системе управления ресурсами кластера и позволяет запускать на кластере вычислительные задания;

SE (Storage Element) – набор программ, предназначенный для установки на узел хранения данных. Данный элемент предоставляет универсальный интерфейс к системе хранения данных и позволяет управлять данными (файлами) в Грид-системе;

WN (Worker Node) – набор программ, предназначенный для установки на каждый вычислительный узел кластера. Данный элемент предоставляет стандартные функции и библиотеки LCG задачам, выполняющимся на данном вычислительном узле;

UI (User Interface) – набор программ, реализующих пользовательский интерфейс Грид-системы (интерфейс командной строки). В этот элемент входят стандартные команды управления задачами и данными, некоторые из которых рассмотрены в следующей главе;

RB (Resource Broker) – набор программ, реализующих систему управления загрузкой (брокер ресурсов). Это наиболее сложный (и объемный) элемент LCG, предоставляющий все необходимые функции для скоординированного автоматического управления заданиями в Грид-истеме;

PX (Proxy) – набор программ, реализующих сервис автоматического обновления сертификатов (myproxy);

LFC (Local File Catalog) – набор программ, реализующих файловый каталог Грид-истемы. Файловый каталог необходим для хранения информации о копиях (репликах) файлов, а также для поиска ресурсов, содержащих требуемые данные;

BDII (Infornation Index) – набор программ, реализующих информационный индекс Грид-системы. Информационный индекс содержит всю информацию о текущем состоянии ресурсов, получаемую из информационных сервисов, и необходим для поиска ресурсов;

MON (Monitor) – набор программ для мониторинга вычислительного кластера. Данный элемент собирает и сохраняет в базе данных информацию о состоянии и использовании ресурсов кластера.

VOMS (VO Management Service) – набор программ, реализующих каталог виртуальных организаций. Данный каталог необходим для управления доступом пользователей к ресурсам Грид-системы на основе членства в виртуальных организациях.

К базовым системам gLite относятся следующие системы:

• **Система управления заданиями WMS** (Workload System). Задачей подсистемы управления загрузкой является принятие запросов на запуск заданий, поиск соответствующих ресурсов и контроль их выполнения. Благодаря работе WMS, сложность управления приложениями и ресурсами

в гриде скрыта от пользователей. Их взаимодействие с WMS ограничено описанием характеристик и требований запроса через ориентированный на пользователя язык описания заданий (Job Description Language, JDL), и к направлению такого запроса через предоставленные интерфейсы.

• **Система управления данными**. Прикладное задание должно быть в состоянии обратиться к своим данным, независимо от фактического местоположения вычислительного ресурса. Наиболее распространенным таким интерфейсом является Менеджер ресурсов хранения данных (Storage Resource Manager, SRM) [48].

• **Система информационного обслуживания и мониторинга грид-системы** решает задачу сбора и управления данными о состоянии грида, получая информацию от множества распределенных источников – поставщиков. Подсистема предназначена для постоянного контроля функционирования грида и обеспечения своевременного реагирования на возникающие проблемы.

• **Система безопасности и контроля прав доступа**. Чтобы грид был эффективной структурой для распределенных вычислений, пользователи, пользовательские процессы и службы грида должны работать в безопасной среде. Для этого взаимодействия между компонентами грида должны быть взаимно аутентифицированы (аутентификация - процесс подтверждения заявленных свойств объекта, на основании удостоверяющих его документов); любое действие должно совершаться только после соответствующей авторизации – сопоставления объекта, совершающего действие, и набора прав, предоставленных этому объекту для работы в грид-среде.

• **Система протоколирования.** Подсистема протоколирования отслеживает процесс выполнения заданий, осуществляемый под управлением WMS. Она собирает извещения о событиях от различных компонентов WMS и обрабатывает их, чтобы представить обобщенное текущее состояние (статус) задания.

• **Система учета.** Эта подсистема предназначена для учета использования вычислительных ресурсов (таких как процессорное время, использование оперативной памяти и так далее), и, в частности, может использоваться для формирования информации о стоимости использования данного грид-ресурса данным пользователем, если взаимоотношения пользователей и провайдеров ресурсов основаны на экономической модели.

16.2. Управления исполнением заданий

Основная и наиболее развитая часть в составе комплекса gLite – это система управления заданиями (Workload Management System, **WMS**), представленная на рис.16.1. Ее назначение – поддержка выполнения программ на распределенных и организованных в виде грида компьютерах

Для пользователя грида вычислительная деятельность выглядит таким образом: он запускает задание, которое доставляется на один или, в случае многопроцессорных заданий, несколько компьютеров из общего ресурсного пула грид, задание выполняется, а результаты могут быть получены на рабочее место, с которого осуществлялся запуск. Реализованные в WMS технологии поддерживает распределенную обработку такого рода, обеспечивая: автоматическое выделение подходящих исполнительных компьютеров (это основное отличие WMS от базовых средств управления заданиями, в которых требуется явное указание исполнительных ресурсов); перемещение программы, входных и диагностических файлов; создание среды выполнения, в том числе домашней директории, на исполнительном компьютере.

Формы пользовательской деятельности в гриде и на отдельном компьютере отличаются в нескольких отношениях:

- Программный код задания выполняется на исполнительном компьютере без участия пользователя (в пакетном режиме). Сам код в рядовом случае не требует адаптации к условиям Грид. В некоторых случаях может потребоваться его дополнение прологом/эпилогом, которые выполняют подготовительные/завершающие операции, например, доставку обрабатываемых данных.
- Задание представляется WMS в виде формализованного описания, составленного на языке JDL (Job Description Language). Способы описания заданий подробно обсуждаются далее.

- Ресурсы Грид используются коллективно множеством пользователей, поэтому задание не обязательно начинает выполняться сразу после запуска: оно может ждать освобождения ресурсов, занятых другими заданиями. Ожидающие ресурсов задания хранятся в очередях (WMS или ресурсных центров CE).
- Вопросы безопасности в гриде gLite решаются на основе принципов GSI. Перед тем как начать работать с WMS (или любой другой системой gLite), пользователь должен сгенерировать временный прокси-сертификат с помощью команд grid-proxy-init или voms-proxy-init.

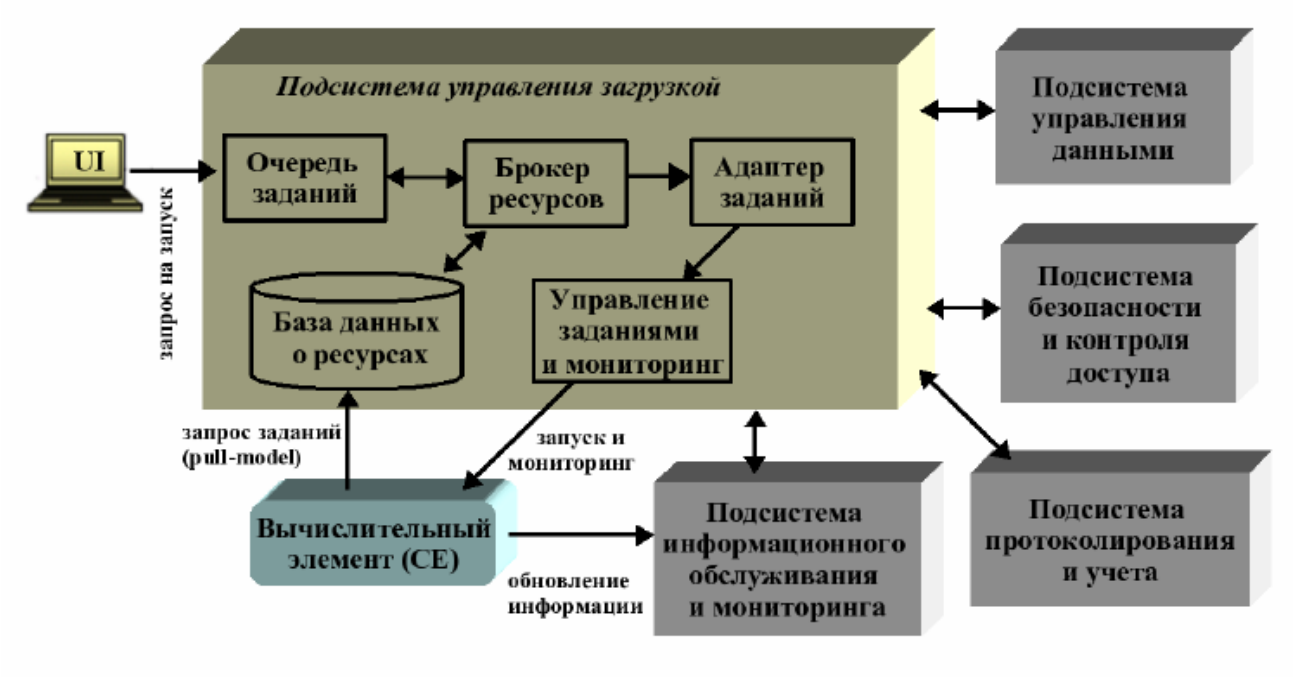


Рис.16. 1. Общая структура системы управления заданиями

Компоненты системы управления исполнением заданий

После отправки с помощью интерфейса пользователя, запрос обрабатывается несколькими компонентами WMS. Их точный состав может варьироваться в зависимо сти от конкретного ППО, но обычно в него входят:

- менеджер загрузки (возможно с дополнительным прокси-сервером);
- планировщик/брокер ресурсов;
- адаптер заданий;
- модуль, ответственный за выполнение фактических операций по управлению заданиями (направление на выполнение, удаление задания, и т.д.);
- монитор log-файлов WMS.

Менеджер загрузки - основной компонент WMS. После получения запроса, он должен предпринять соответствующие действия, чтобы выполнить его. Чтобы сделать это, он взаимодействует с другими компонентами WMS, набор которых зависит от типа запроса. Для повышения производительности системы (количества запросов, которые могут быть обработаны за единицу времени), этот компонент дополняется прокси-менеджером загрузки ответственным за прием поступающих запросов от интерфейса пользователя (например, на запуск или удаление задания), анализ их корректности и, если результат анализа положителен, передачу запроса менеджеру загрузки.

Основным типом запросов к WMS являются запросы на выполнение заданий. А главными результатами их обработки подсистемой являются:

- правильный подбор грид-ресурса для выполнения конкретного задания на основании описания задания на языке JDL и информации о доступных ресурсах.

- направление задания на выбранный ресурс.

Первую задачу решает компонент, называемый **планировщиком или брокером** ресурсов (Resource Broker, RB). Существование подходящих ресурсов для данного задания зависит не только от состояния ресурсов, но также и от политики их использования, которой следуют администраторы ресурса и/или администратор виртуальной организации, к которой принадлежит пользователь.

Брокер ресурсов может следовать различной стратегии при распределении заданий по ресурсам:

- один крайний вариант стратегии - после поступления запроса на выполнение задания, как можно скорее подбираются наиболее подходящие ресурсы и, как только решение принято, задание передается на выбранный ресурс для выполнения («нетерпеливое планирование», в англоязычной литературе называется также push-model);
- другой крайний вариант - задания находятся в менеджере загрузки, пока какой- либо ресурс не становится доступным, после чего освободившемуся ресурсу подбирается наиболее соответствующее задание (из ожидающих в менеджере) и передается этому ресурсу для выполнения («ленивая политика планирования», pull-model);
- возможны промежуточные комбинированные варианты стратегии подбора ресурсов.

Для выполнения своих задач брокер использует базу данных о ресурсах, которая обновляется в результате либо получения уведомлений от ресурсов, либо активного опроса ресурсов системой информационного обслуживания и мониторинга, или некоторой комбинации этих двух механизмов обновления. Кроме того, специальный компонент WMS, отвечающий за взаимодействие с информационной подсистемой, может быть сконфигурирован так, чтобы определенные уведомления об изменении состояний ресурсов могли вызвать начало работы брокера. Это обеспечивает реализацию «ленивой политики планирования» (например, после уведомления, что какой-то ресурс освободился брокер инициализирует отправку на него ожидающего задания).

Другим важным компонентом, повышающим устойчивость работы WMS, является модуль управления очередью заданий, который дает возможность некоторое время хранить запрос на выполнение задания, даже если никакие ресурсы, соответствующие его требованиям, не доступны немедленно. Запросы, по которым не удалось сразу подобрать ресурсы, будут повторены либо периодически (при «нетерпеливом планировании») или как только уведомления о новых доступных ресурсах появляются в хранилище данных (при «ленивой политике планирования»). В противном случае, такие ситуации могли бы привести к немедленному прекращению выполнения задания из-за отсутствия подходящего ресурса.

Далее следует **модуль, ответственный за выполнение фактических операций** по управлению заданиями (направление на выполнение, удаление задания, и т.д.), выполняемых по запросам менеджера загрузки. В gLite в качестве этого модуля используется Condor C.

Монитор log-файлов ответственен за просмотр сообщений модуля управления заданиями и перехват событий об изменении состояний задания.

Описание задания

Пользователь взаимодействует с WMS со своего рабочего места, на котором установлен интерфейс (User Interface, UI) этой системы, включающий следующий набор команд:

glite-job-submit <jdl_file> запуск задания, описание которого содержится в файле <jdl_file
glite-job-cancel <job_Id> снятие задания в любой момент процесса обработки.
glite-job-status <job_Id> получение состояния задания..
glite-job-output <job_Id> получение суммарной диагностики от всех х этапов обработки.

Как видно из этого списка, набор команд достаточно полон: он позволяет запустить задание, отслеживать ход обработки и при необходимости снять его. Возвращаясь к особенностям работы в

гриде, обратим внимание на то, что обработка команд производится распределенным образом различными службами WLMS, которые взаимодействуют между собой по сети. В связи с этим время выполнения команд достаточно велико, порядка минуты.

Файл описания задания – JDL-файл задается в качестве параметра команды job-submit и содержит характеристики задания в виде пар: <атрибут> = <значение>. Примера на (рис.16.1).

```
[Type = "Job";

JobType = "Normal";

Executable = "myexe";

StdInput = "myinput.txt";

StdOutput = "message.txt";

StdError = "error.txt";

InputSandbox = {"/users/pacini/example/myinput.txt",
"/users/pacini/example/myexe"};

OutputSandbox = {"message.txt", "error.txt"};

Requirements = other.GlueCEInfoLRMSType == "PBS";
Rank = other.FreeCPUs;]
```

Рис. 16.1. Пример файла описания задания.

Среди атрибутов можно выделить три основные группы, которые определяют: тип задания, используемые в задании файлы и способ выбора ресурсов.

Тип задания описывается двумя атрибутами: Type и JobType. WMS поддерживает работу как с простыми заданиями, так и с составными, атрибут Type имеет, соответственно, значения “Job” или “DAG”. Тип DAG (direct acyclic graph of dependent jobs) отвечает заданию, в котором должны быть выполнены в определенной последовательности ряд простых заданий.

Для простого задания применим второй атрибут - JobType. Оно может быть обыкновенным (“Normal”), а также:

- интерактивным (“Interactive“). Задание такого рода поддерживает связь с точкой запуска
- параллельным (MPICH), то есть требующим для выполнения нескольких процессоров и др.

Простые задания

WMS обеспечивает **перемещение файлов** между компьютером рабочего места и исполнительным компьютером, но ограничивается, однако, минимально необходимым набором стандартных файлов операционной системы Unix: исполняемым файлом задания, входных данных, диагностических сообщений и файлом сообщений об ошибках.

Входные файлы (исполняемый и файл данных), лежащие в файловой системе рабочего компьютера пользователя UI, передаются сначала на сервер WMS, а затем при запуске задания на исполнительном компьютере загружаются в созданную для этого задания домашнюю директорию.

Выходные файлы диагностики и ошибок порождаются при выполнении задания в домашней директории исполнительного компьютера, и доставляются по его окончании не в точку запуска, а на сервер WMS, откуда их можно получить командой glite-job-output. Аналогично входным файлам, выходные должны быть описаны в атрибутах OutputSandbox, StdOutput и StdError.

Поскольку перемещение файлов происходит через временный буфер на сервере WMS, предполагается, что все они будут небольшого размера. Тем самым, рассматриваемые средства доставки файлов направлены на минимально необходимую поддержку удаленного выполнения программ. Что касается прикладных данных, которые обрабатываются или производятся приложением, то для работы с ними в программе должны использоваться средства системы **Управления данными**.

Одно из самых важных достижений WMS – технология виртуализации, реализующая автоматическое **распределение заданий по исполнительным ресурсам**. Распределение производится с точностью до ресурсного центра, то есть определяется не конкретный исполнительный компьютер, а некоторый CE. При этом в определенной степени учитывается состояние и состав его ресурсов (число заданий в очереди, платформа исполнительных компьютеров и т.д.). Эти сведения поставляются диспетчеру заданий WMS информационной службой gLite.

WMS обеспечивает ограниченную поддержку многопроцессорных параллельных заданий, использующих **протокол MPI**. Для таких заданий в JDL-файле задается атрибут NodeNumber, определяющий число необходимых процессоров и установку исполнительной среды MPICH.

```
(other.GlueCEInfoTotalCPUs $>=$ NodeNumber) &&
Member(other.GlueHostApplicationSoftwareRunTimeEnvironment,"MPICH")
```

Схема обработки заданий

Рис.16.2 илюстрирует процесс выполнения в Грид некоторого простого задания. Последовательные шаги исполнения представлены в тексте после рисунка.

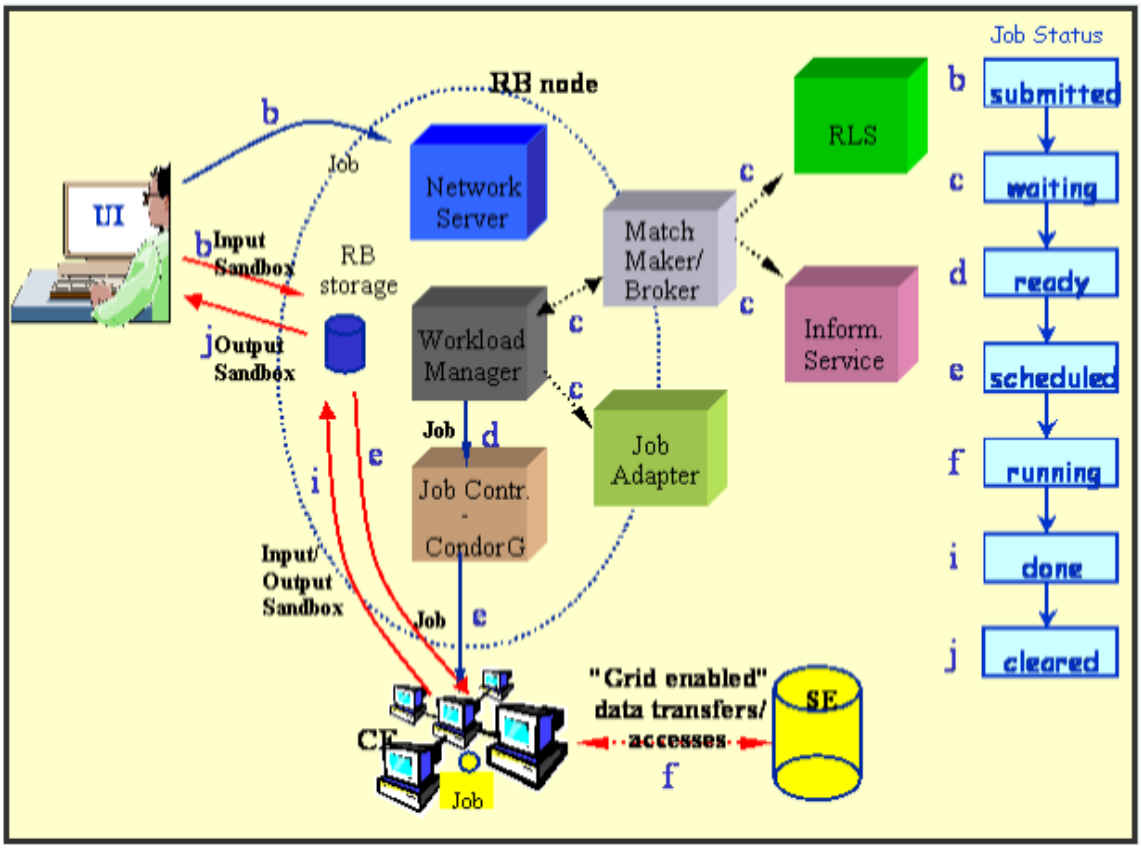


Рис. 16.2. Исполнения задания в gLite. Справа представлены состояния выполнения задания

1. После получения цифрового сертификата от Certification Authority, регистрации в BO (Виртуальная Организация), Пользователь может использовать gLite. Он регистрируется в UI и создает прокси сертификат, чтобы представлять себя в операциях по безопасности.

2. Пользователь запрашивает работу через UI к Resource Broker. В описании работы указаны один или более файлов, которые нужно скопировать из в RB. Этот ряд файлов называется ***Input Sandbox***. Событие регистрируется в LB и работе присваивается статус ЗАПРОШЕНО.

3. WMS ищет наилучший доступный CE для выполнения работы. Чтобы сделать это, он запрашивает ***Information Supermarket (ISM***, внутренний кэш информации, который в этой систеие читает из BDII, чтобы определить статус вычислительных ресурсов и ресурсов памяти и в File Catalogue, чтобы найти место любого требуемого входного файла. Другое событие регистрируется в LB и устанавливается статус работы ОЖИДАНИЕ.

4. RB готовит работу для запроса выполнения, создавая оболочку скрипта, который будет переслан весте с другими параметрами в выбранный CE. Событие регистрируется в LB и работе присваивается статус ГОТОВО.

5. CE получает запрос и посылает работу на исполнение в локальный LRMS.

6. Событие регистрируется в LB и работе присваивается статус СПЛАНИРОВАНО.

7. LRMS урпавляет исполнением работы на Worker Nodes. Файлы входного Sandbox копируются из RB в доступный WN, где работа исполняется. Событие регистрируется в LB и работе присваивается статус ИСПОЛНЕНИЕ.

8. Во время исполнения работы файлы Грид могут быть доступны из SE, используя либо RFIO или протокол перкрытия, или после копирования их в локальную файловую систему на WN с помощью Data Management.

9. работа может создавать новые выходные файлы, которые могут быть перезагружены в Грид и стать доступными для других пользоателей. Это может быть достигнуто с помощью Data Management, который описывается позже. Перезагрузка файла означает копирование его в Storage Element и регистрацию в каталоге файлов.

10. Если работа закончена без ошибок, вызод (не большой файл данных, а маленькие файлы, созданные пользователем в так еазываемом ***Output Sandbox)*** передаются назад в RB узел. Событие регистрируется в LB и работе присваивается статус СДЕЛАНО.

11. С этой точки пользователь может извлекать выходной файл его работы в UI. Событие регистрируется в LB и работе присваивается статус ОЧИЩЕНО.

12. Запросы на статусы работы могут быть адресованы LB из UI. Итак, из UI возможно делать очереди BDII для статусов ресурсов.

13. Если сайт, к которому послана работа, неспособен принять или выполнить работу, работа может быть автоматически и многократно переадресована другим подходящим CE. Пользователь может получить информацию об истории работы, запрашивая службу LB.

16.3. Россия в проекте EGEE

Для участия России в этом проекте был образован консорциум РДИГ (Российский Грид для интенсивных операций с данными) или RDIG (Russian Data Intensive Grid). Консорциум РДИГ, согласно принятой в проекте EGEE структуре, входит в проект в качестве региональной федерации «Россия» («Russia»). Цели РДИГ:

- создание национальной инфраструктуры Грид в интересах организаций из различных областей науки, образования и промышленности;
- пропаганда технологии Грид и обеспечение возможности обучения и подготовки специалистов для использования Грид в новых научных областях и экономике;
- обеспечение полномасштабного участия России в создании глобальной компьютерной инфраструктуры Грид.

Участники РДИГ (участники проекта EGEE):
- ИФВЭ (Институт физики высоких энергий, Протвино);
- ИМПБ РАН (Институт математических проблем биологии, Пущино);
- ИТЭФ (Институт теоретической и экспериментальной физики, Москва);
- ОИЯИ (Объединенный институт ядерных исследований, Дубна);
- ИПМ РАН (Институт прикладной математики, Москва);
- ПИЯФ РАН (Петербургский институт ядерной физики, Гатчина);
- РНЦ «Курчатовский институт» (Москва);
- НИИЯФ МГУ (Научно-исследовательский институт ядерной физики МГУ) и др.

В рамках РДИГ Россия участвует в выполнении экспериментов на Большом Адронном коллайдере, осуществляются работы физики, геофизики, биомедицины, выполняет тестирование инфраструкткры РДИГ и др.

Лекция 17. Учебные системы для практического освоения параллельных вычислений на кластере и в Грид.

Возможны два варианта выполнения этих работ:

1. **С выходом в интернет**. Так делается, например, в России. Когда пользователь по через виртуальную организацию входит в Грид и выпоняет там определенные действия. С уществуют и специальные учебные Грид – это итальянская система GILDA (Grid Infn Laboratory for Dissemination Activities), являющегося виртуальной лабораторией для демонстрации возможностей технологии Грид. На сайте http://egee.pnpi.nw.ru/doc/UG_corr.doc представлены оба эти варианта. И оба варианта
2. **Без выхода в интернет**, создаются на базе локальной сети. Такую учебную сеть можно создать в небольшой организации или вузовской кафедре. Во всяком случае. Этот вариант позволяет научиться: создавать сеть, инсталлировать системное ПО (сложная работа) и производить параллельные вычисления. Именно этот мобильный вариант и будет рассматриваться дальше.

Для того, чтобы построить параллельную систему на базе локальной сети, нужно:

1. Построить локальную сеть компьютеров, чаще всего она уже имеется.
2. Инсталлировать системное программное обеспечение для вариантов и **Грид.**
3. Создать параллельную пронрамму,выполнить настройки системы и пронраммы.
4. Приизвести вычисления и оценить их эффектовность.

Ниже будет представлен поцесс инсталляции мобильного варианта Грид, полагая. Что локальная сеть с аппаратурой и программным обеспечением уже созданана:

1. На сайте **www.globus.org** по адресу **http://www.globus.org/toolkit/downloads/4.0/** представлен перечень и адреса операционных систем и других прораммных комплексов, которые можно использовать для инсталляции одного из вариантов Грид.
2. По адресу **http://www.globus.org/toolkit/docs/4.0/admin/docbook/quickstart.html** расположено описание порядка и всех деталей инсталляции по варианту **quickstart** (быстрый старт). Для выполнения этого пункта часть программ можно установить предварительно (пункт 1), а часть – в процессе выполнения пункта 2. Но при выполнении пункта 2 всякий раз надо убеждаться, установлено ли необходимое программное обеспечение.
3. После завершения инсталляции Грид необходимо поверх него установить пакет MPICH-G2, котрый является специальной реализацией MPI для Globus Toolkit Таким средством является пакет MPICH-G2 (**www.globus.org/grid_software/computation/mpich-g2.php**) - это специальная реализация MPI для Globus Toolkit.
4. Наконец, следует запустить на Грид с установленным MPICH-G2 тест для измерения производительности параллельных систем **HPL,** предназначенный для решения систем линейных алгебраических уравнений и являющийся развитием теста LINPACK. Информация по тесту **HPL** размещена по адресу **http://www.netlib.org/benchmark/hpl/index.html,** алгоритм теста размещен в лекции 13, а перевод описания теста приводится ниже.

Тест HPL

Тест применяется при формировании списка наиболее производительных вычислительных систем в мире (список **Top 500**). Тест решает случайно заданную систему линейных уравнений методом LU-разложения, используя 64-битную арифметику с плавающей запятой и позволяет судить о производительности системы при решении такой задачи. .

На этой странице дается высокоуровневое описание алгоритма, использованного в пакете. HPL допускает много вариантов для различных операций. По умолчанию операции и варианты могут выбираться во время исполнения. Для получения хороших характеристик было решено оставить за пользователем возможность экспериментально определять ряд параметров для данной машин-

ной конфигурации. С точки зрения точности вычислений все возможные комбинации строго взаимно эквивалентны, хотя результат может быть слегка другой (в битах).

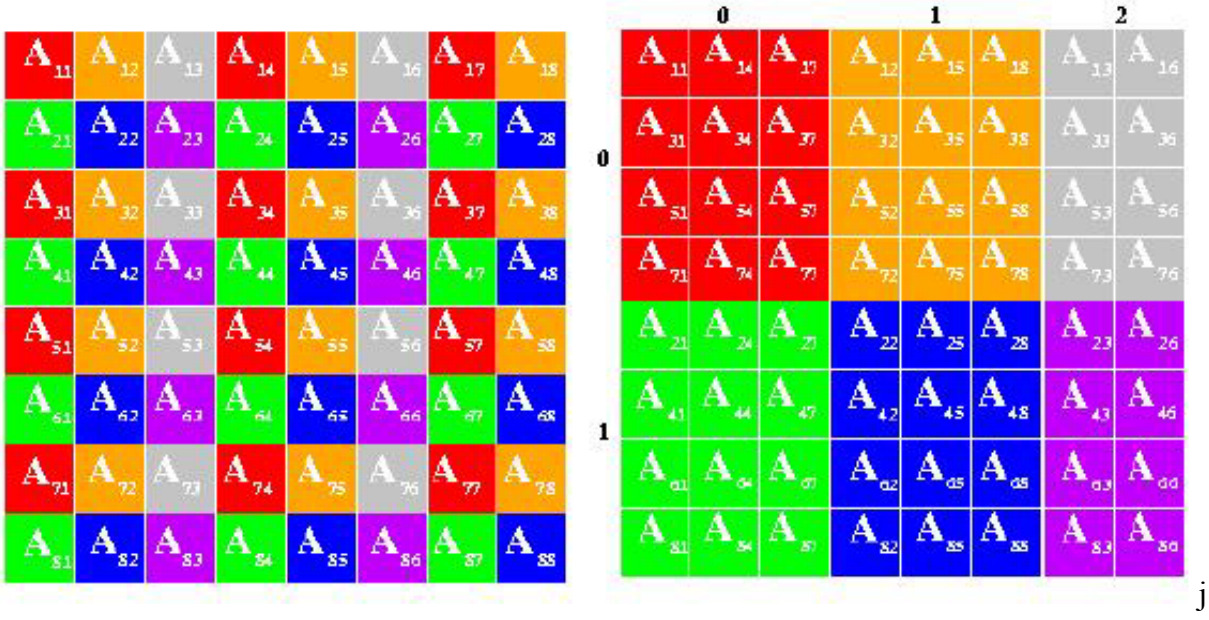
Панель – это место, где проводятся вычисления между двумя широковещаниями

- Main Algorithm
- Panel Factorization
- Panel Broadcast
- Look-ahead
- Update
- Backward Substitution
- Checking the Solution

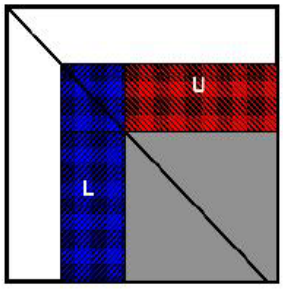
Основной алгоритм (Main Algorithm)

Этот пакет решает линейную систему порядка n: A x = b для первого вычисления LU факторизации с выбором ведущего элемента столбца (row partial pivoting) n-by-n+1 коэффициента матрицы [A b] =[[L,U] y]. Решение системы получается в два этапа, сначала ищется Ly=b, затем – конечный результат Ux=y. Нижняя треугольная матрица L далее не нужна и не возвращается.

Данные размещаются в двумерной сетке P-by-Q процессов согласно блочно-циклической схеме, чтобы гарантировать хорошую балансировку нагрузки и масштабируемость алгоритма. n-by-n+1 коэффициенты матрицы сначала разделены логически в nb-by-nb блоки, которые циклически налагаются на P-by-Q решетку процессов. Это сделано для обеих размерностей матрицы.

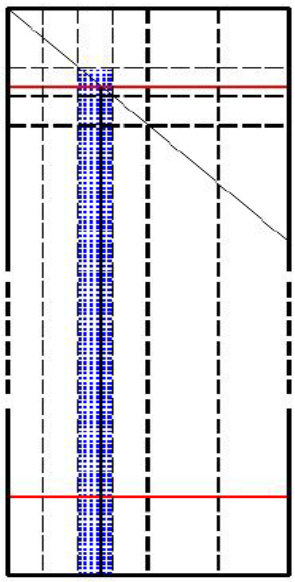


Для главного цикла LU факторизации был выбран правосторонний вариант. Это означает, что факторизуется каждая итерация цикла панели из nb столбцов и конечная подматрица корректируется. Заметим, что это вычисление вследствие этого логически разделено на такие же блоки размером nb, как и матрица данных.



Панель факторизации (Panel Factorization)

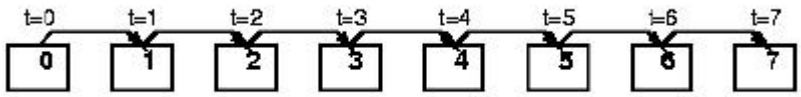
На данной итерации главного цикла и вследствие декартовских свойств схемы распределения каждая панельная факторизация имеет место в одном столбце процессов. Эта частная доля вычислений лежит на критическом пути общего алгоритма. Пользователю предлагается выбор из трех вариантов матричного умножения (Crout, left- and right-looking), основанного на рекурсивных вариантах. Программное обеспечение также позволяет пользователю выбрать, в каком множестве субпанелей текущая панель должна быть разделена во время рекурсии. Более того, можно также выбрать критерий остановки во время исполнения рекурсии в терминах числа столбцов, оставленных для факторизации. Когда этот порог достигается, субпанель будет затем факторизована одним из трех Crout, left- and right-looking матрично-векторных вариантов. Наконец, для каждого панельного столбца поиск ведущего элемента (the pivot search), обмен и рассылочная операция ведущего столбца объединяются в одну единственную операцию. A binary-exchange (leave-on-all) редукция выполняет эти три операции одновременно.



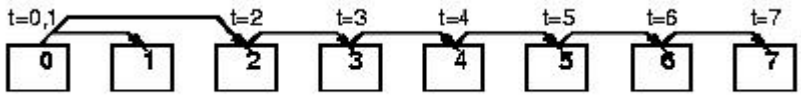
Рассылка панелей (Panel Broadcast)

Как только факторизация панели была выполнена, эта панель столбцов пересылается столбцам других процессов. Имеется много алгоритмов возможных широковещания и программное обеспечение предлагает 6 вариантов. Эти варианты описаны ниже, полагая для удобства, что процесс 0 является источником широковещания. Пусть значек “ ->” означает “послать в “.

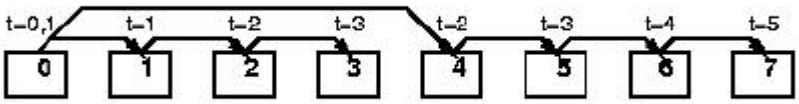
- **Возрастающее кольцо:** 0 -> 1; 1 -> 2; 2 -> 3 и так далее. Это классика. Он имеет предупреждение, что процесс 1 должен послать сообщение.



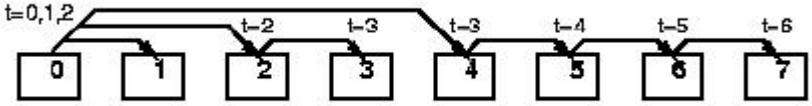
- **Возрастающее кольцо (модифицированное):** 0 -> 1; 1 -> 2; 2 -> 3 и так далее. Процесс 0 посылает два сообщения и процесс 1 только получает одно сообщение. Этот алгоритм всегда лучше, если не наилучший.



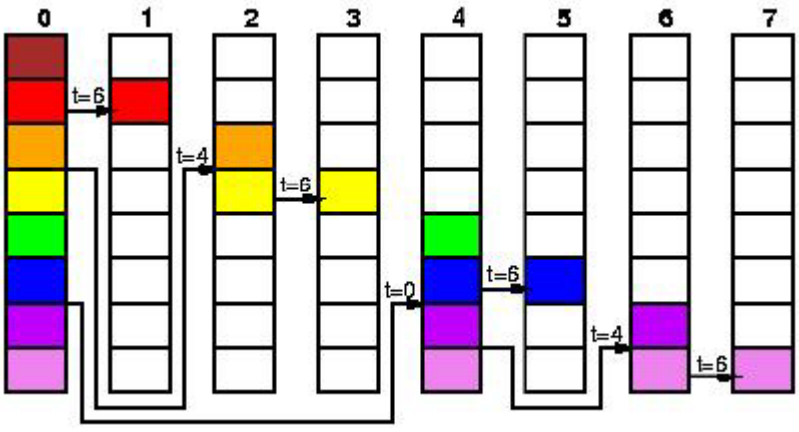
- **Возрастающее-2-кольцо:** Q процессов разделяются на две части: : 0 -> 1 и 0 -> Q/2; Затем процесы 1 и Q/2 действуют как источники двух колец 1 -> 2, Q/2 -> Q/2+1; 2 -> 3, Q/2+1 -> to Q/2+2 и так далее. Этот алгоритм имеет преимущество в уменьшении времни за счет того, что процесс 0 посылает 2 сообщения.



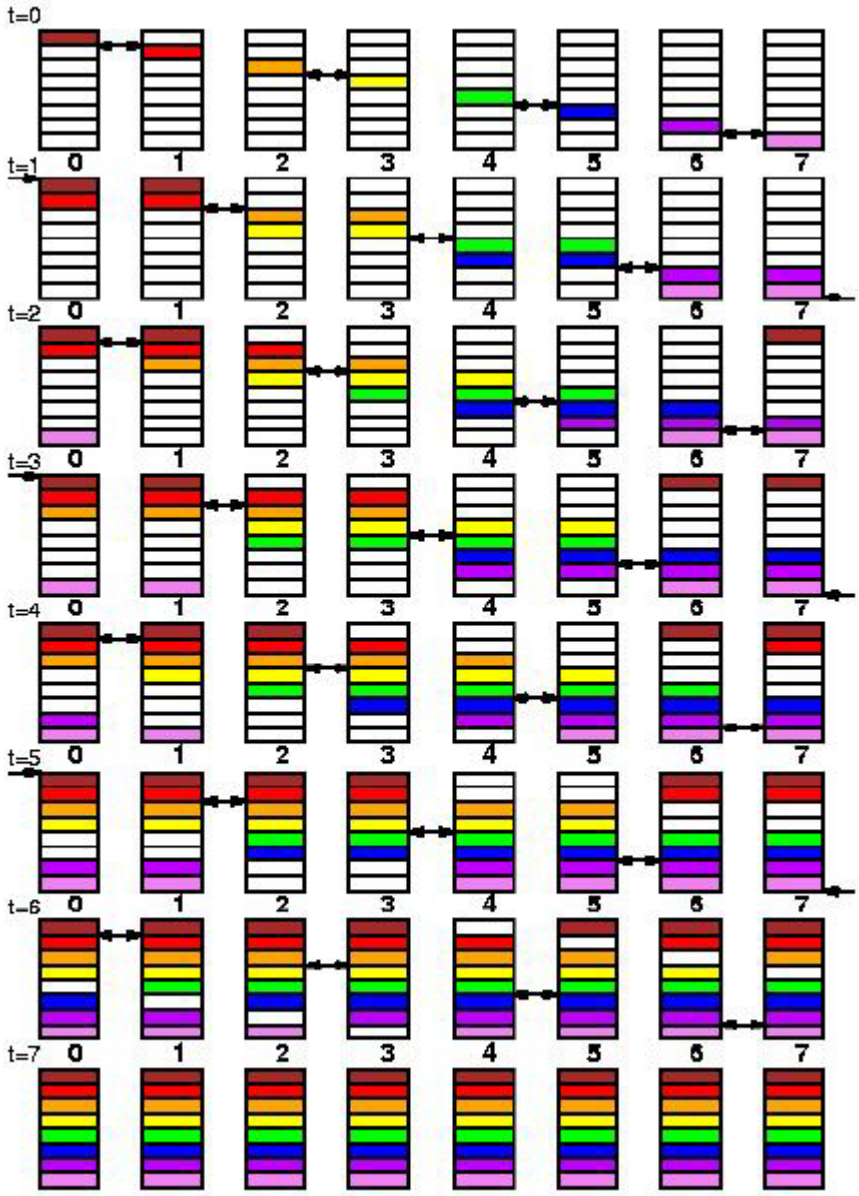
- **Возрастающее-2-кольцо (модифицированное):** Как можно ожидать**,** сначала 0 -> 1, затем Q-1 оставшихся процесса разделяются на две равные части: 0 -> 2 and 0 -> Q/2; Процесс 2 и Q/2 действуют затем как источники двух колец: 2 -> 3, Q/2 -> Q/2+1; 3 -> 4, Q/2+1 -> to Q/2+2 и так далее. Этот алгоритм вероятно наиболее серьезный соперник модифицированным вариантам.



- **Long (bandwidth reducing)**: в противоположность предыдущим вариантам, этот алгоритм и его повторитель (follower) синхронизуют все процессы участвующие в операции. Сообщение разделяется в Q равных частей, которые рассыпаны по Q процессам.

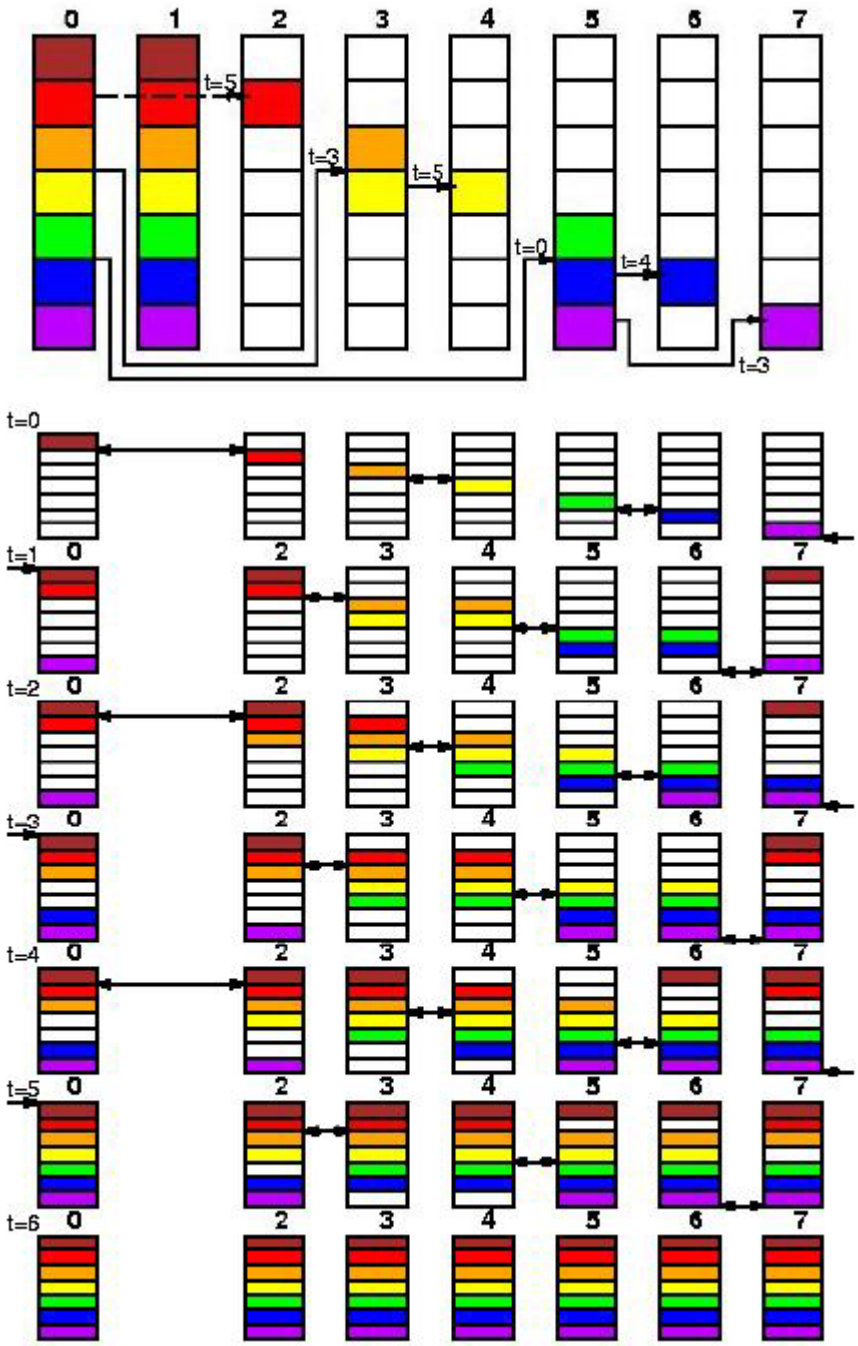


Части затем сворачиваются в Q-1 шагов. Фаза рассеивания использует двоичное дерево и фаза сворачивания исключительно использует взаимные обмены сообщениями. В нечетных шагах 0 <-> 1, 2 <-> 3, 4 <-> 5 и так далее; в четных шагах Q-1 <-> 0, 1 <-> 2, 3 <-> 4, 5 <-> 6 и так далее.



Большинство сообщений обмениваются, однако общий объем коммуникацийне зависит от Q, делая этот алгоритм удобным для больших сообщений. Этот алгоритм становится конкурентноспособным, когда узлы “очень быстрые “ и сеть (сравнительно) “очень медленная “

- **Long (bandwidth reducing modified)**: то же, что и выше, за исключением того, что 0 -> 1 первый, и затем Long variant используется на процессах 0,2,3,4 .. Q-1.



Кольцевые варианты отличаются пробным механизмом, который активирует их. Другими словами, процесс вовлечен в широковещание and different from the source asynchronously probes for the message to receive. Когда сообщение способно продолжать широковещание and otherwise the function returns. Это позволяет перекрывать широковещание с обновлением данных. Это уменьшает пустое время, затрачиваемое процессами, ожидающими for the factorized panel. Этот механизм необходим, чтобы приспособить computation/communication performance ratio.

Просмотр вперед (Look-ahead)

Как только панель имела широковещание или в течение этой операции, замыкающая подматрица корректируется, используя последнюю панель в цепи просмотра вперед: как было замечено ранее, панель факторизации лежит на критическом пути, который означает, что когда kth panel факторизована и затем выполнила широковещание, следующая наиболее срочная задача для окончания есть факторизация и широковещание of the k+1 th panel. Эта техника часто называют в литературе

«просмотр вперед». Этот пакет позволяет выбрать различную глубину просмотра. По соглашению, глубина просмотра 0 соответствует отсутствию просмотра. В этом случае замыкающая субматрица обновляется панелью текущего широковещания. Просмотр вперед требует некоторой памяти, чтоб хранить все панели, участвующие в просмотре. Глубина просмотра 1 (или 2) повидимому самая подходящая.

Обновление (Update)

Обновление остаточной подматрицы последней панелью в просмотровой цепи делается в две фазы. Первое, ведущие элементы должны быть использованы, чтобы сформировать текущую строку панели U. U затем должна быть решена верхней треугольной матрицей столбцовой панели. U наконец должна быть разослана широковещательно каждой процессной строке так, чтобы локальное rank-nb update могло иметь место. Мы выбираем комбинирование обмена и широковещания U по обновленной стоимости решения. Для этой коммуникационной операции пригодны два коммуникационных алгоритма.

- **Binary-exchange:** это модифицированный вариант binary-exchange (leave on all) reduction operation. Каждый столбец процесса выполняет те же самые операции. Алгоритм работает следующим образом. Он претендует на уменьшение строковой панели, но вначале только полноценная копия владеет текущим процессной строкой. Другие процессные строки будут вкладывать строки из A, которыми они владеют, чтобы скопировать в U и заместить them строками, которые были оригинально в текущей строке процесса. Полностью оперция выполняется за log(P) шагов. Для простоты примем, что P есть степень двух. На шаге k процессная строка p обменивается сообщениями со строкой p+2^k . Существенны два случая. Первое, одна из этих двух строк получила U на предыдущем шаге. Обмен имеет место. Один процесс обменивает его локальные строки A into U. Обе процессные копии в U удаляют строки A. Второе, ни одна из этих процессных строк не получила U, обмен имеет место, и оба процесса просто добавляют те удаленные строки к списку, который они аккумулировали до сих пор. На каждом шаге сообщение размера U обменивается по крайней мере парой строк.

- **Long**: это вариант уменьшения bandwidth, дополняющий ту же задачу. Строчная панель есть первое расширение (используя дерево) среди процессных строк, по отношению к ведущему массиву. Это есть scatter (V variant for MPI users). Локально каждая процессная строка затем обменивает эти строки со строками A, которыми она владеет, и которые принадлежат U. Каждая строка перемешивает U и обрабатывает с вычислительной частью обновления. Пара замечаний: процессные строки логарифмически сортируются перед распространением, так чтобы процесс, получающий наибольший номер строки, есть первый в дереве.Это делает комуникационный объем оптимальным на этой фазе. Наконец, перед сверткой (rolling) и после локального обмена имеет место фаза балансирования, во время которой локальные части U однородно распространяются по процессным строкам, используется древовидный алгоритм. Эта операция необходима, чтобы поддерживать свертку оптимальной, даже если ведущие строки не равны в распределенных столбцах. Этот алгоритм имее сложность в терминах объема обмена, который в основном зависит от размера U. В частности, число строк действует только на число обмениваемых сообщений.Это поэтому будет превосходить предыдущий вариант для больших проблем на больших машинных конфигурациях.

Первый вариант есть модифицированная leave-on-all reduction операция . Второй вариант имеет коммуникационный объем сложности, который в основном зависит от размера U (номер процессных строк влияет на число сообщений обмениваемых) и следовательно нужно выполнять пре-

дыдущий вариант для больших проблем на больших машинных конфигурациях. В дополнение оба предыдущмих варианта могут быть объединены in a single run of the code.

Пользователь может выбрать любой из приведенных выше вариантов. В дополнение, возможна их смесь. Алгоритм "binary-exchange" будет использован, когда U содержит по крайней мере определенное число столбцов. Выбирая наименьший блочный размер рекомендуется, когда используется просмотр вперед.

Обратная подстановка (ackward Substitution)

Факторизация теперь закончена, остается сделать обратную подстановку. Для этого мы выберем вариант просмотра вперед на один шаг. Правосторонний подход опережает в процессных столбцах in a decreasing-ring fashion, так что мы решаем Q * nb элементов одновременно.На каждом шаге эта сжатая часть правой стороны обновляется. Процесс, как и выше, один владеюший диагональным блоком матрицы A обновляет сначала последний nb элемент x, направляет его в предыдущий процессный столбец, затем широковещает его в встолбец in a decreasing-ring fashion as well. Решение затем обновляется и посылается затем в предыдущий столбец. Решение линейной системы повторяется в каждом процессорном столбце.

Проверка решения

Чтобы проверить полученный результат, входная матрица и правая сторона регенерируются. Три остатка вычисляются и решение рассматривается как "numerically correct", когда все эти объекты меньше, чем пороговое значение порядка 1.0. В выражениях ниже eps есть относительная (distributed-memory) машинная точность.

- ||Ax-b||_oo / (eps * ||A||_1 * N)
- ||Ax-b||_oo / (eps * ||A||_1 * ||x||_1)
- ||Ax-b||_oo / (eps * ||A||_oo * ||x||_oo)

Для использования теста HPL в Грид используется пакет MPICH-G2 [16].

Источники информации

1. В. Воеводин, Вл. Воеводин. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 609 с.
2. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы:Учебник для вузов. 2-е изд. СПб.: Питер, 2003. 864 с.
3. Argonne National Laboratory (http://www.mcs.anl.gov/mpi).
4. Проект Globus по разработке Грид (http:/www.globus.org)
5. Библиотеки BLAS, LAPACK, BLACS, ScaLAPACK: (http://www.netlib.org).
6. Научно-исследовательский вычислительный центр МГУ (http://www.parallel.ru).
7. Интернет-портал по грид-технологиям (http://www. gridclub.ru)
8. Союзная программа СКИФ (skif.bas-net.by/)
9. Белгосуниверситет, Минск (http://www.cluster.bsu.by)
10. Шпаковский Г. И. Организация параллельных ЭВМ и суперскалярных процессоров: Учеб. пособие. Мн.: Белгосуниверситет, 1996. 284 с.
11. Шпаковский Г.И., Серикова Н.В. Программирование для многопроцессорных систем в стандарте MPI: Пособие / Мн.: БГУ, 2002. -323 с. ISBN 985-445-727-3
12. Шпаковский Г.И., А.Е. Верхотуров, Н.В. Серикова. Организация работы на вычислительном кластере : учебное пособие для студентов естественнонауч. фак. БГУ. Мн.: БГУ, 2004. -182 с.
13. Вержбицкий В.М. Основы численных методов. М.: Высш. школа, 2005 г. 848 с.
14. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем: Пер. с англ. М.: Мир, 1991.367 с.
15. Design and Implementation of LAPACK and ScaLAPACK (www.cs.berkeley.edu/~demmel/cs267/lecture12/lecture12.html)
16. MPICH-G2 (http://www.globus.org/grid_software/computation/mpich-g2.php)
17. Пособие по работе в Грид (ожидается)