Г.И.ШПАКОВСКИЙ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ И АППАРАТУРА

МИНСК
2012, Апрель

Справка о книге

Шпаковский Г.И. Параллельное программирование и аппаратура. – Минск, БГУ, 2012 г., 184 с.

Книга написана на основе лекций, которые в течение длительного времени читались студентам Белорусского государственного университета (г. Минск). По тематике параллельных вычислений и сетей изданы фундаментальные работы [1 - 4], кроме того существует ряд сайтов по проблемам параллельных вычислений [5 - 7]. Настоящая же книга предназначены для первичного ознакомления с уровнями и механизмами параллельной обработки.

Книга разделена на два раздела: «Архитектура» и «Реализация».

В разделе «Архитектура» приведены примеры классической реализации различных архитектур для мелкозернистого и крупнозернистого параллелизма: конвейерные системы, технологии MMX и VLIW, системы с общей, индивидуальной и смешанной организацией памяти, а также **системы программирования** для них, в частности, MPI, OpenMP и CUDA, даны оценки эффективности вычислений для этих архитектур.

В разделе «Реализация» описаны четыре типа **аппаратных средств**, с помощью которых можно увеличить количество процессоров, так как это ведет к увеличению быстродействия: вычислительные кластеры, грид, многоядерные процессоры и квантовые компьютеры.

Книга предназначена для студентов старших курсов, аспирантов и широкого круга научных работников и инженеров, интересующихся разработкой быстродействующих ЭВМ и параллельных алгоритмов.

Оглавление

ПРЕДИСЛОВИЕ

Настоящая книга написана на основе лекций, которые в течение длительного времени читались студентам Белорусского государственного университета (г. Минск). За прошедшие годы произошли значительные изменения в области реализации систем для параллельных вычислений. Стали широко применяться вычислительные кластеры, появились многоядерные микропроцессоры, грид и квантовые компьютеры. Эти вопросы и рассматриваются в книге, благодаря чему книга имеет некоторую завершенность.

По тематике параллельных вычислений и сетей изданы фундаментальные работыты, в частности, книга В.В.Воеводина и Вл.В. Воеводина «Параллельные вычисления» [1], книга В.Г. Олифер и Н.А. Олифер «Комьютерные сети» [2], книга Дж. Ортеги «Введение в параллельные и векторные методы решения линейных систем» [3], книга М. Нильсена и И. Чанга «Квантовые вычисления и квантовая информатика» [4]. Кроме того существует ряд сайтов [5 – 7] по различным проблемам параллельных вычислений. Настоящая же книга предназначены для первичного ознакомления со уровнями и механизмами параллельной обработки.

Книга имеет два раздела: «Архитектура» и «Реализация».

В разделе «Архитектура» описываются способы повышения быстродействия ЭВМ. Их немного:

1. Повышение тактовой частоты элементной базы и и способы его реализации, например, удлинение конвейера.
2. Параллельное или совмещенное выполнение команд или программ. Эти способы могут быть реализованы разными путями, но суть их при этом не меняеся. Эту неизменность можно назвать архитектурой.

Чтобы пояснить архитектурные принципы, приведены примеры их классической реализации для мелкозернистого и крупнозернистого параллелизма: конвейерные системы, технологии MMX и VLIW, системы с общей и индивидуальной памятью, а так же системы программирования для них, в частности, MPI и OpenMP, даны оценки эффективности вычислений для этих архитектур.

В разделе «Реализация» описаны четыре типа параллельных систем, с помощъю которых можно увеличить количество процессоров, так как это ведет к увеличению быстродействия:

1. Вычислительные кластеры получили широкое распространение благодаря простоте реализации, а при использовании многоядерных микропроцессоров - и большое быстродействие.
2. Многояденые микропроцессоры находятся в стадии своего развития, и могут использоваться как автономно в персоналых ЭВМ, так и в качестве элементной базы для кластеров. Примеры приводятся в основном по приборам компании Intel, которые дают достаточно информации для илюст-

рации основных тем книги, поэтому материалы по другим крупным компаниям IBM, AMD, NVIDIA практически не используются.

3. Грид также является развивающейся структурой, однако, уже существующие реализации позволяют получить практически неограниченное быстродействие.
4. На сегодня существуют только первые попытки физической реализации квантовых компьютеров, но в случае успеха скорость вычислений будет действительно неограниченной, но для определенного и важного класса задач.

В приложении коротко представлена суперкомпьютерная платформа СКИФ (Супер Компьютерная Инициатива Феникс), разрабатываемая в рамках программ Союзного государства Беларуси и России. Уже сегодня платформа позволилат создать высокопроизводительные компьютеры мирового уровня.

Книга предназначена для студентов старших курсов, специализирующихся в области быстродействующих ЭВМ и параллельных алгоритмов, аспирантов и широкого круга научных работников и инженеров, связанных с решением задач с большим объемом вычислений.

Р А З Д Е Л 1. АРХИТЕКТУРА

Глава 1. ПРИНЦИПЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

1.1 Большие задачи.

Все задачи принципиально можно разделить на две группы: P (Polinomial) и NP (Non Polinomial) задачи [3]. P задачи характеризуются объемом вычислений $a^{\,p}$, где a – объем входных данных, p – полином невысокой степени. Такие задачи под силу современным многопроцессорным ЭВМ и называются «большими». NP задачи характеризуются выражением $a^{f(a)}$ и современным машинам «не по зубам». Для решения таких задач могут использоваться квантовые ЭВМ, скорость вычислений которых пропорциональна объему данных.

Напомним некоторые обозначения, используемые для больших машин.

Обозначение	Величина	Достигнутый уровень	Единицы измерения
Кило	10^3		Герцы, байты, FLOP/S
Мега	10^6		
Гига	10^9	ПЭВМ до 5 Гфлопс	
Тера	10^{12}	СуперЭВМ – тера и петафлопсы	
Пета	10^{15}		
Экза	10^{18}		

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

Время решения «больших» задач определяется 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 10^{12} -10^{15} операций с плавающей точкой.

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 [1]. При построении алгоритмов используется принцип дискретизации. Атмосфера разбивается сеткой с шагом 1 градус по широте и долготе на всей поверхности зем-

ного 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 опрераций, то проведение одного эксперимента с моделью атмосферы требуется выполнить 10^{16} – 10^{17} операций с плавающей запятой.

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 вопрос более философский. Но есть и более существенные, сегодняшние задачи, например, проблема разработки ядерного оружия. После подписания в 1963 году договора о запрещении испытания ядерного оружия остался единственный путь совершенствования этого оружия – численное моделирование экспериментов. По количеству операций такое моделирование соизмеримо с моделированием климата. Но это касается больших стран – США, Россия, Евросоюза. А зачем суперЭВМ, например, для Беларуси.

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.

Ниже приведены времена выполнения некоторых больших задач на современной однопроцессорной системе

Проблема	Число операций (флопс)	Время счета
Модель атмосферы	10^{16} - 10^{17}	годы
Модель ядерного взрыва	10^{16} - 10^{17}	годы
Модель обтекания самолета	10^{15} - 10^{16}	месяцы, недели
Краш - тесты	10^{14} - 10^{15}	месяцы, недели
Промышленные конструкции	10^{14} - 10^{15}	недели

1.2. Методы повышения быстродействия

Естественно, для решения больших задач нужны все более быстрые компьютеры. Есть всего два основных способа повышения быстродействия ЭВМ:

1. За счет повышения быстродействия элементной базы (тактовой частоты). Быстродействие процессора растет пропорционально росту тактовой час-

тоты, при этом не требуется изменения системы программирования и пользовательских программ.

2. За счет увеличения числа одновременно работающих в одной задаче ЭВМ, процессоров, АЛУ, умножителей и так далее, то есть за счет параллелизма выполнения операций. Это требует использования сложных систем параллельного программирования. Это крупный недостаток метода.

Параллельные системы по архитектуре разделяются на два класса:

- Конвейерные системы, когда несколько специализированных блоков одновременно работают над частями одного потока команд.
- Параллельные системы, когда множество команд одной программы одновременно выполняются множеством АЛУ или процессоров.

Рассмотрим примеры этих методов.

Тактовая частота (рис.1.1). Тактовая частота зависит от размеров конструктива, кристалла, на котором расположены арифметико-логические элементы. Ниже на рисунке представлена схема совпадения, которая срабатывает при одновременном появлении двух сигналов.

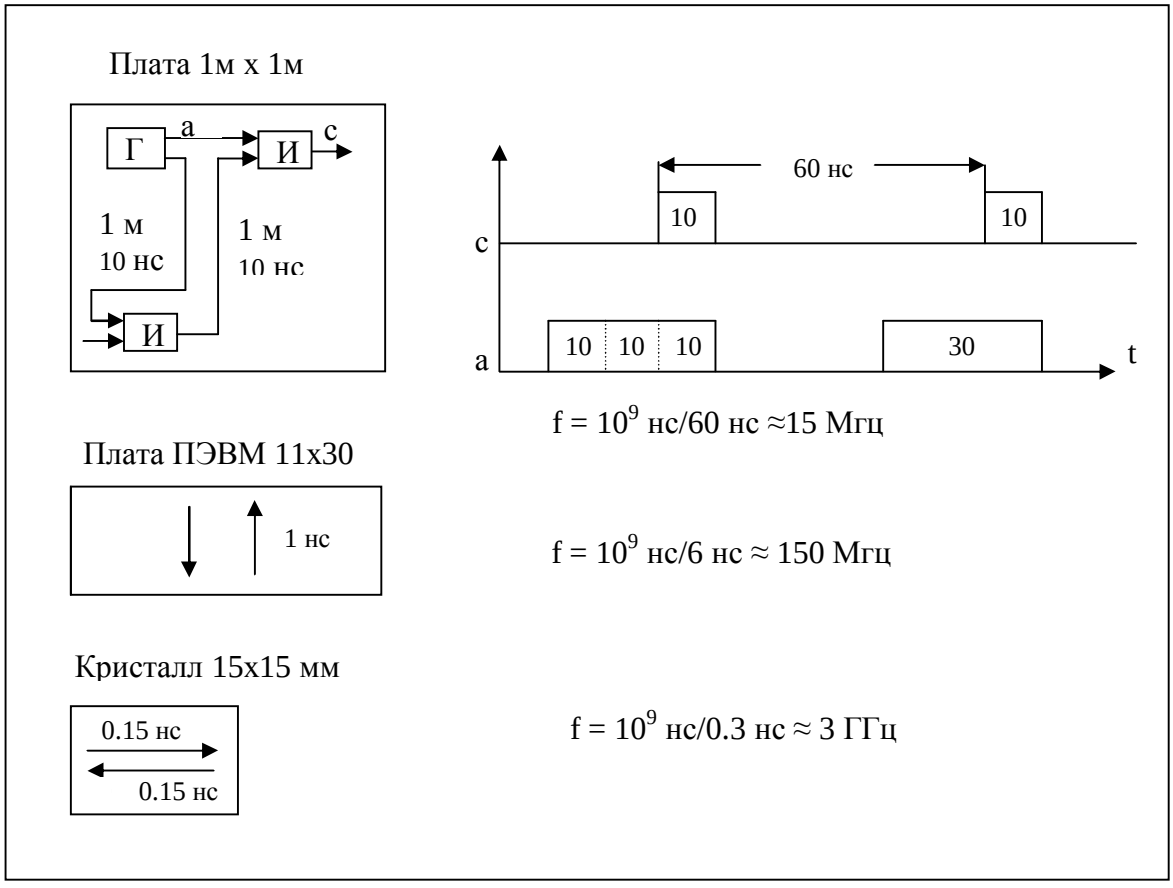


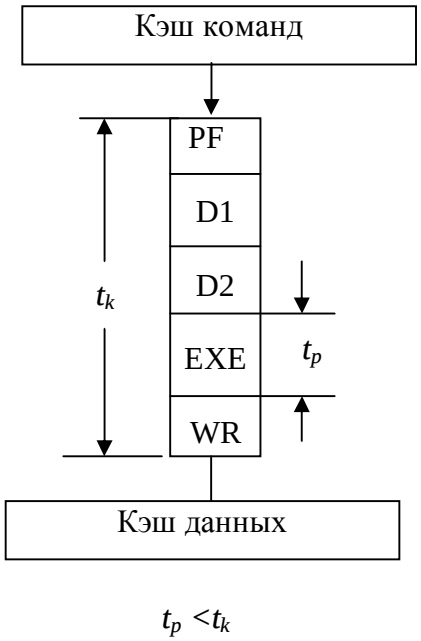
Рис.1.1. Рост частоты синхронизации с умешением раззмеров схемы

Один сигнал поступает непосредственно от генератора, другой – с задержкой, определяемой размером платы, на которой размещена схема. 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, соответственно для платы ПЭВМ и кристалла СБИС частота будет 150 Мгц и 3Ггц. Повышение тактовой частоты является важным средством повышения быстродействия компьютеров, но ограничено фундаментальными физическими законами. Естественно, при переходе на расстояния внутри кристалла, соответственно вырастает и частота.

В реальной комбинационной схеме компьютера за время такта синхронизации сигнал *последовательно* проходит через много логических схем. Это время прохождения определяет длительность такта и частоту синхронизации, которая будет ниже приведенной на рисунке.

Конвейерные системы. Для примера рассмотрим конвейер команд известного микропоцессора Pentium (рис.1.2).



$t_p < t_k$

Рис.1.2. Схема конвейера команд.

Он содержит следующие ступени:

- 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.

Конвейерные системы теряют смысл, когда время передачи информации со ступени на ступень становится соизмеримым со временем вычислений на каждой ступени.

Параллелельные системы. Параллельная машина содержит множество процессоров П, объединенных сетью обмена данными (рис.1.3). Аппаратура одновременно выполненяет более одной арифметико-логической или служебной операций.

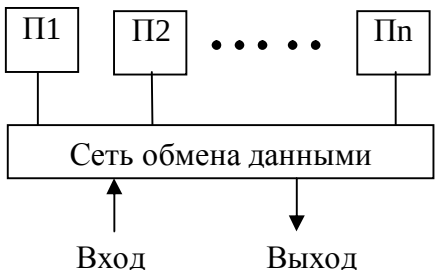


Рис.1.3. Схема параллельной ЭВМ

В параллельных ЭВМ межпроцессрный обмен данными является принципиальной необходимостью, и при медленном обмене однопроцессорный вариант может оказаться быстрее многопроцессорного, поэтому количество процессоров в параллельной ЭВМ определяется скоростью сетей обмена.

1. 3. Формы параллелизма

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

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

$$(\mathrm{InB} \wedge \mathrm{OutA}) \vee (\mathrm{InA} \wedge \mathrm{OutB}) \wedge (\mathrm{OutA} \wedge \mathrm{OutB}) = \varnothing, \qquad (1.1)$$

где In(A) — набор входных, а Out(A) — набор выходных переменных объекта A. Если условие (1.1) не выполняется, то между A и B существует зависимость и они не могут выполняться параллельно.

Если условие (1.1) 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, точнее порядок выполнения операций в задаче, приведен на рис.1.4. В своей первоначальной форме ИГ, тем не менее, не используется ни математиком, ни программистом, ни ЭВМ.

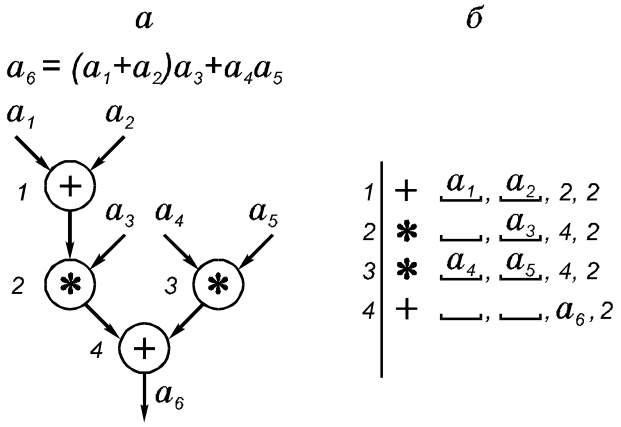


Рис.1.4. Информационный граф математического выражения (*а*) и порядок выполнения операций в выражении (*б*)

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:

- Мелкозернистый параллелизм (он же *параллелизм смежных операций* или *скалярный параллелизм*).
- Крупнозернистый параллелизм, который включает: *векторный параллелизм* и *параллелизм независимых ветвей*..

Мелкозернистый параллелизм (Fine Grain)

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.1.5.

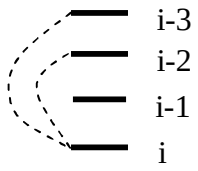


Рис.1.5. Предварительная подготовка операндов для команды i.

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

$$E = \frac{mq^4 p^2}{8e_0^2 e^2 h^2}.$$

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. 1.6.).

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.

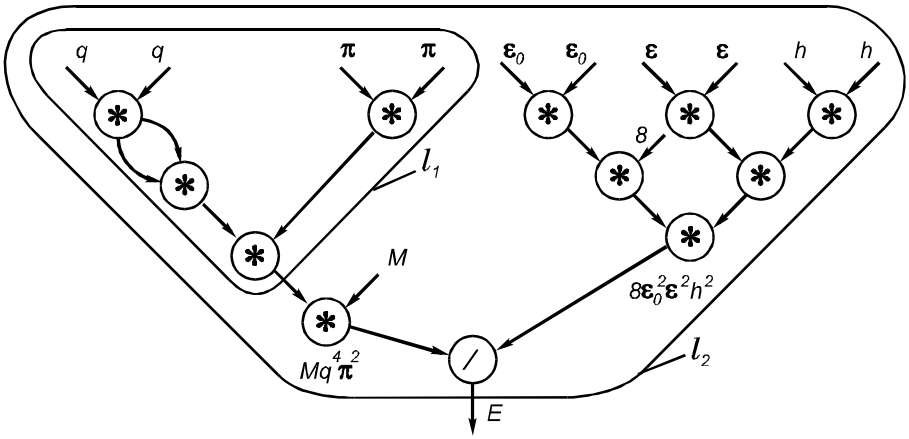


Рис.1.6. ЯПФ вычисления величины *E*

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

Крупнозернистый параллелизм (coarse grain)

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 программные части, независимые 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. 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 $F_1 \ldots F_3$ 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.1.7.

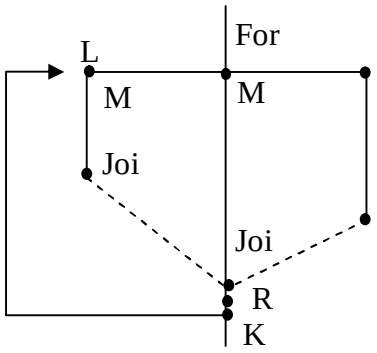


Рис.1.7. Блок-схема выполнения параллельной программы.

Для приведенного примера характерны две особенности:

- Присутствует *синхронизация процессов*, для которой используются оператор JOIN и ячейка R. Состояние R = 0 свидетельствует об окончании процессов разной длительности.
- Производится *обмен данными* (обращение за X*i* из разных ветвей).

Параллелизм вариантов. Это частный, но широко распространенный на практике случай параллелизма независимых ветвей, когда производится решение одной и той же задачи при разных входных параметрах, причем, все варианты должны быть получены за ограниченное время. Например, варианты моделирования используются при анализе атмосферной модели климата.

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 таких задач.

1.4. Эффективность параллельных вычислений (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* [8]:

$$R = \frac{W \cdot t}{(W_{ск} + \frac{W_{пр}}{n}) \cdot t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow[n\to\infty]{} \frac{1}{a} . \qquad (1.2)$$

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 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.

Если система имеет несколько архитектурных уровней с разными формами параллелизма, то *качественно* общее ускорение в системе будет:

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

где r_i - ускорение некоторого уровня.

1.5. Закон Мура и его перспективы.

Количество транзисторов на одном кристалле достигает миллиардов штук. Естественный способ их использовать – строить многопроцессорные системы. Для таких компаний как Intel вопрос создания многопроцессорных систем – это вопрос существования.

Для дельнейшего рассмотрения приведем таблицу размеров вентилей, транзисторов и т.д., используемых в микроэлектронике:

Наименование	**Величина, метры**
Миллиметр, мм	10^{-3}
Микрометр, мкм	10^{-6}
Нанометр, нм	10^{-9}
Атом водорода	$0.25.10^{-9}$
Ангстрем, А	10^{-10}
Пикометр, пм	10^{-12}

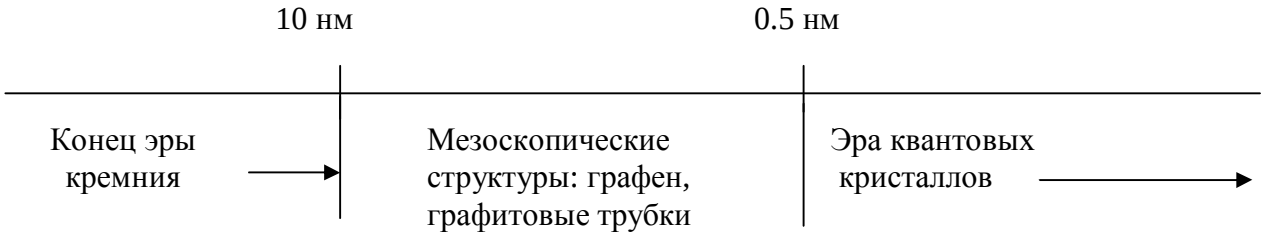
Г. Мур (G. Moor – создатель Intel) на основе развития технологии в компании Intel в 1965 году выдвинул следующее положение, которое сейчас называют законом Мура [9]:

Каждые 2 года количество транзисторов на кристалле удваивается

Этот закон и с некоторыми колебаниями сохраняется длительное время. Число транзисисторов на кристалле увеличится в такой степени,что это позволяет создавать многоядерные процессоры (МЯП), в которых на одном кристалле размещены сотни и тысячи ядер, каждое из которых является полноценным процессором.

Успехи микроэлектроники позволяют сейчас работать с размерами транзисторов, указанными в таблице. Считается, что нанотехнологии начинаютя со 100 нм. Таким образом, можно сказать, что современные микропроцессоры – это область нанотехнологий.

До минимального размера порядка 10 нм транзистор сохраняет свои переключательные и усилительные свойства, что полностью определяет путь развития **кремниевой** наноэлектроники вплоть до 2020 г. Ниже 10 нм кремний теряет проводимость. В диапазоне размеров 5-0.5 нм наступает эра **мезоскопических структур** и приборов. Мезоскопические структуры — электронные приборы, размеры активной области которых сопоставимы с параметрами электрона. При размерах 0.5 нм и менее **— эра квантовых кристаллов.**



Графен – это одиночный плоский лист, состоящий из атомов углерода, образующих решётку из шестиугольных ячеек. Нанотрубки состоят из тех же шестиугольных ячеек, имеют средний диаметр около 1 нм и длину до нескольких сантиметров. **Но отдельный транзистор – это не процессор.** Поэтому квантовые компьютеры могут оказаться ближе по времени, чем мезоскопические структуры.

1.6 Основные этапы развития параллельной обработки

Идея параллельной обработки возникла одновременно с появлением первых вычислительных машин. В начале 50-х гг. американский математик Дж. фон Нейман предложил архитектуру последовательной ЭВМ, которая приобрела классические формы и применяется практически во всех современных ЭВМ. Однако фон Нейман разработал также принцип построения процессорной матрицы, в которой каждый процессор был соединен с четырьмя соседними.

D825. Одной из первых полномасштабных многопроцессорных систем явилась система D825 фирмы “BURROUGHS”. Начиная с 1962 г. было выпущено большое число экземпляров и модификаций D825. Выпуск первых многопроцессорных систем, в частности D825, диктовался необходимостью получения не высокого быстродействия, а высокой живучести ЭВМ, встраиваемых в военные командные системы и системы управления. С этой точки зрения параллельные ЭВМ считались наиболее перспективными. Система D825 содержала до четырех процессоров и 16 модулей памяти, соединенных матричным коммутатором, который допускал одновременное соединение любого процессора с любым блоком памяти.

Практическая реализация основных идей параллельной обработки началась только в 60-х гг. 20 - го столетия. Это связано с появлением транзистора, который позволил строить машины, состоящие из большого количества логических элементов, что принципиально необходимо для реализации любой формы параллелизма.

CRAY. Основополагающим моментом для развития конвейерных ЭВМ явилось обоснование академиком С.А. Лебедевым в 1956 г. метода, названного “принципом водопровода” (позже он стал называться *конвейером*). Прежде всего был реализован конвейер команд, на основании которого практически одновременно были построены советская ЭВМ БЭСМ-6 (1957-1966 гг., разработка Института точной механики и вычислительной техники АН СССР), и английская машина ATLAS (1957-1963 гг.). *Конвейер команд* предполагал наличие

многоблочной памяти и секционированного процессора, в котором на разных этапах обработки находилось несколько команд.

Следующим заметным шагом в развитии конвейерной обработки, реализованном в ЭВМ CDC-6600 (1964 г.), было введение в состав процессора нескольких функциональных устройств, позволяющих одновременно выполнять несколько арифметико-логических операций: сложение, умножение, логические операции.

В конце 60-х гг. был введен в использование *арифметический конвейер*, который нашел наиболее полное воплощение в ЭВМ CRAY-1 (1972-1976 гг.). Арифметический конвейер предполагает разбиение цикла выполнения арифметико-логической операции на ряд этапов, для каждого из которых отводится собственное оборудование. Таким образом, на разных этапах обработки находится несколько чисел, что позволяет производить эффективную обработку вектора чисел.

Сочетание многофункциональности, арифметического конвейера для каждого функционального блока и малой длительности такта синхронизации позволяет получить быстродействие в десятки и сотни миллионов операций в секунду. Такие ЭВМ называются *супер*ЭВМ.

ILLIAC-IV. Идея получения сверхвысокого быстродействия в первую очередь связывалась с *процессорными матрицами* (ПМ). Предполагалось, что, увеличивая в нужной степени число процессорных элементов в матрице, можно получить любое заранее заданное быстродействие

Поскольку в 60-е гг. логические схемы с большим уровнем интеграции отсутствовали, то напрямую реализовать принципы функционирования процессорной матрицы, содержащей множество элементарных процессоров, не представлялось возможным. Поэтому для проверки основных идей строились однородные системы из нескольких больших машин. Так, в 1966 г. была построена система Минск-222, разработанная Институтом математики Сибирского отделения АН СССР и минским заводом ЭВМ им. Г.К.Орджоникидзе. Система содержала до 16 соединенных в кольцо ЭВМ Минск-2. Для нее было разработано специальное математическое обеспечение.

Другое направление в развитии однородных сред, основанное на построении процессорных матриц, состоящих из крупных процессорных элементов с достаточно большой локальной памятью, возникло в США и связано с именами Унгера, Холланда, Слотника. Была создана ЭВМ ILLIAC-IV (1966-1975 гг.), которая надолго определила пути развития процессорных матриц. В машине использовались матрицы 8×8 процессоров, каждый с быстродействием около 4 млн оп/с и памятью 16 кбайт. Для ILLIAC-IV были разработаны кроме Ассемблера еще несколько параллельных языков высокого уровня. Особенно ценным является опыт разработки параллельных алгоритмов вычислений, определивший области эффективного использования подобных машин.

Транспьютер. Совершенствование микроэлектронной элементной базы, появление в 80-х годах БИС и СБИС позволили разместить в одной микросхеме процессор с 4-мя внешними связями, который получил название *транспьютер*. Теперь стало возможным строить системы с сотнями процессоров.

Вычислительные кластеры. Далее развитие и производство супер-ЭВМ пошло широким потоком. Сначала строились монолитные многопроцессорные системы, для которых все разрабатывалось специально для конкретной системы: элементная база, конструктивы, языки программирования, операционные системы. Затем оказалось много дешевле строить вычислительные кластеры на основе промышленные средства, появились многояденые процессора, Грид, квантовые компьютеры.

Некоторые этапы развития параллельных ЭВМ качественно можно представить следующей таблицей:

№	Название ЭВМ	Годы	Новизна	Программы
1	D825 – одна из первых многопроцессорных систем	1962	Доказана возможность построения многопроцессорных систем	Первая ОС для многопроцессорных систем - ASOR
2	Матричный процессор ILLIAC IV	1972	Реализована ОКМД машина	Параллельный язык Glupnir
3	Векторно- конвейерная ЭВМ CRAY	1976	Предложены конвейерные вычисления	Предложен ЯВУ векторного типа
4	Транспьютер T414	1985	Разработан процессор на кристалле со связями для мультисистем	Язык описания параллелизма OCCAM
5	Кластер Beowulf	1994	Сборка на серийном оборудовании	Использованы обычные сетевые ОС
6	Грид	1998	Неограниченная возможность расширения	GlobusToolkit, gLite
7	Многоядерные процессоры	2001	Разработаны МЯ процессоры с общей и индивидуальной памятью	OpenMP и MPI. Нужны новые разработки
8	Квантовый компьютер Orion компании D-Wave	2007	Кубит, эспоненциальная скорость за счет суперпозиции	Алгоритмы Шора, Гровера. Языки моделирования

Глава 2. МЕЛКОЗЕРНИСТЫЙ ПАРАЛЛЕЛИЗМ

Мелкозернистый параллелизм обеспечивается за счет параллелизма внутри базовых блоков (ББ), которые являются частями программ, не содержащими условных и безусловных переходов. Этот вид параллелизма реализуется блоками одного процессора: различными АЛУ, умножителями, блоками обращения к памяти, хранения адреса, переходов и так далее.

2.1. Принципы распараллеливания и планирования базовых блоков.

Размер ББ и его увеличение. Базовые блоки невелики по размеру (5 – 20 команд) и даже при оптимальном планировании параллелизм не может быть большим. На рис.2.1. точками представлена экспериментальная зависимость ускорения от размера базового блока. Поле полученных в экспериментах результатов ограничено контуром (точками представлены значения для некоторых $ББ_i$). На основе рис.2.1. с учетом вероятностных характеристик контура результатов можно получить следующую качественную зависимость:

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

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.

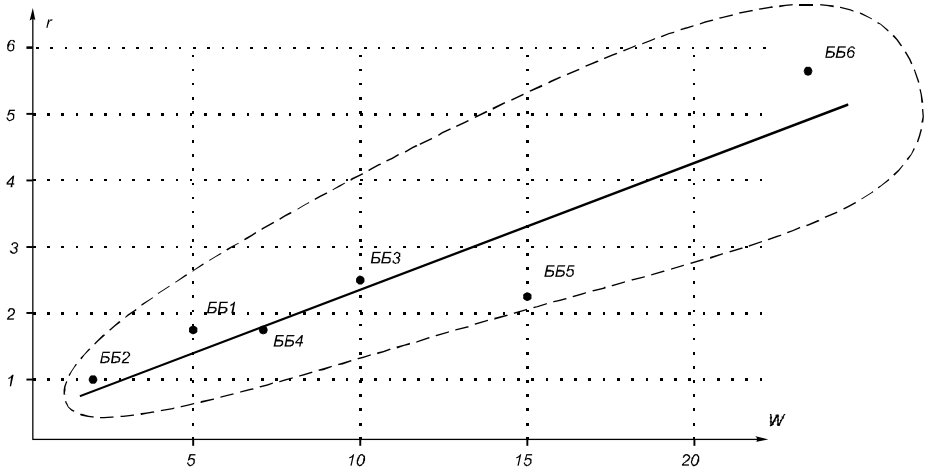


Рис.2.1. Рост ускорения в зависимости от размера базового блока

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+1) + B(I+1)
      C(I+2) = A(I+2) + B(I+2)
      C(I+3) = 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.

Метод Фишера. Существует большое количество методов увеличения параллелизма при обработке базовых блоков [10]. Но в большинстве случаев тело цикла содержит операторы переходов. Достаточно универсальный метод планирования трасс с учетом переходов предложил в 80-е годы J.Fisher [11]. Рассмотрим этот метод на примере рис.2.2. На схеме в кружках представлены номера вершин, а рядом – вес вершины (время ее исполнения); на выходах операторов переходов проставлены вероятности этих переходов.

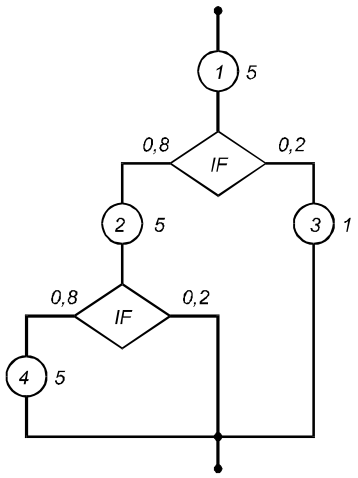


Рис.2.2. Пример выбора трасс

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.

Чтобы исключить возможность неправильных вычислений, вводятся компенсационные коды. Рассмотрим примеры (рис.2.3). Пусть текущая трасса (рис.2.3,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 вводит компенсирующую операцию 1 (рис.2.3,б).

Пусть теперь операция 3 перемещается выше оператора 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 оператора IF было бы допустимым.

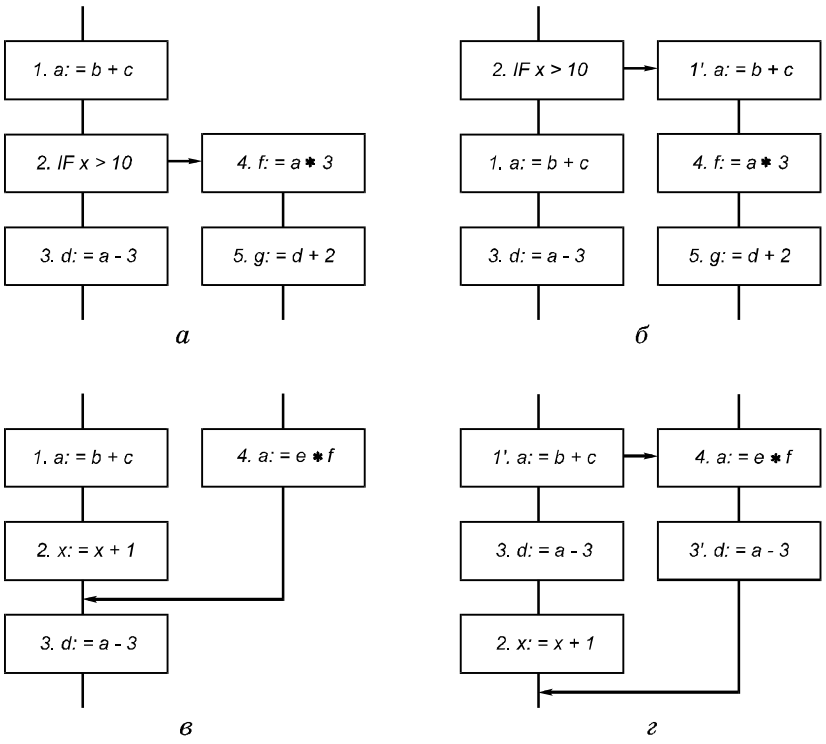


Рис. 2.3. Способы введения компенсационных кодов

Рассмотрим переходы в трассу извне. Пусть текущая трасса содержит операции 1, 2, 3 (рис.2.3,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.2.3,г).

Порядок планирования трасс для получения конечного результата таков:

1. Выбор очередной трассы и ее планирование (распараллеливание и размещение по процессорам).
2. Коррекция межтрассовых связей по результатам упаковки по процессорам 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 исполнения, то компиляция программы считается законченной, в противном случае осуществляется переход на пункт 1.

Основным объектом распараллеливания в области скалярного параллелизма являются базавые блоки. Если ББ представлен на язык высокого уровня, тогда распараллеливанию подвергаются арифметические выражения. Если ББ представлен на ассемблер или в машинных кодах, то распараллеливается отрезок программы. Эти операции отличаются. Далее будут рассмотрены оба подхода, которые используются в компиляторах для автоматического распараллеливания.

Алгоритм автоматического распараллеливания арифметических выражений (по А.В.Вальковскому)**.** В этом разделе описан метод, использующий отношение старшинства между операциями. Идея метода в том, что в выражении выделяются подвыражения, содержащие самые высокоприоритетные операции. Выделенные подвыражения одинаковой глубины «склеиваются» в более крупное подвыражение, таким образом «вырастает» выражение, все подвыражения которого имеют равномерную глубину.

Обычно в качестве знаков арифметических операций используют: +, -, *, /, !, μ. Две последние операции — возведение в степень и унарный минус. При неформальном изложении знак умножения * иногда опускается. Вводится старшинство, или приоритет, операций:

$$Pr(\mu) = 4;\ \ Pr(!) = 3;\ \ \ Pr(*) = Pr(/) = 2;\ \ Pr(+) = Pr(-) = 1.$$

Опишем один вариант алгоритма более подробно, используя для наглядности только лишь выражения с операциями +, *.

Для хранения текущей информации нам потребуется следующая память: ячейки *x* — для сканируемых операндов, *y* — для сканируемых операций, *L* — для операции, стоящей слева от текущего операнда, *R* — для аналогичной операции справа, Out — для выходного выражения, T_i — ячейки для записи промежуточных результатов. Кроме того, воспользуемся вектор-стеком: St = (St_1, St_2) — компонента St_1 для операндов; St_2 — для операций; Sc — процедура сканирования следующего символа входной строки; символ —> обозначает засылку. Считается, что входное выражение слева и справа ограничено пробелами, Рг (▬) = 0. Первоначально в *R* и Out содержится пробел.

Шаг 1. Sc —> *x* , *Sc*—> *y*, *R* —> *L*, *y* — > *R*. Сканируется очередной операнд и знак операции после него. Операция (пробел) слева от операнда *x* засылается в ячейку *L*, справа от *x* — в ячейку *R*.

Шаг 2. Если **(**St=***Ø*** OR** Рг *(L)*< Рг ***(R)*** **OR** Рг (St_2)<Рг (L)**) AND** Рг ***(R)≠ Ø***, то на шаг 3, иначе на шаг 4.

Шаг 3. *x* —> St_1 , y —> St_2 , переход на шаг1. Запоминаем операнд и операцию и переходим к сканированию следующей пары (операнд, операция).

Шаг 4. Если Рг *(L)* = Рг (St_2), то на шаг 5, иначе на шаг 6.

Шаг 5. Out T_k y —> Out. Если Рг (y) = 0, то конец разбора, иначе па шаг 1. Здесь промежуточная ячейка T_k = (St_1 St_2 *x)*. Таким образом, когда найдены две операции одного старшинства, первая из них с соседними операндами группируется в промежуточный результат T_k, вслед за ним выписывается вторая операция, и все это приписывается к выходной строке. Далее T_k воспринимается алгоритмом как атомный операнд.

Шаг. 6. Out *x y* —> Out, если *y* = "_", то конец разбора, иначе на шаг. 1. В случае если не находится двух операций одного старшинства и приоритеты пошли на убывание, операнд и операция просто приписываются к выходной строке.

Приведенный алгоритм описывает один из проходов, после которого некоторые из подвыражений «свертываются» в промежуточные результаты T_k. После этого алгоритм повторяется до тех пор, пока все выражение не сведется к единственному T_k.

Ниже показано выражение ***e = a + b + c * d* I* f + g*** после серии последовательных проходов.

1. Входная строка: ▬*a+ b +c*d*l*f+g*▬ .
2. Результат 1-го прохода: ▬ $T_1 + T_2 * T_3 + g$▬ , где
   $T_1 = (a + b)$, T_2 = (c*d), T_3 = (*1* *f).
3. Результат 2-го прохода: ▬ $T_4 + T_5$▬ , где
   $T_4 = (T_2 * T_3)$, $T_5 = (T_1 + g)$.
4. Результат 3-го прохода: ▬T_6▬; $T_6 = (T_4 + T_5)$.
5. Выходная строка: ▬ (((c * *d)* (l * f)) + ((a + b*)* + *g))* ▬

Ниже представлена таблица, подробно описывающая весь процесс преобразования указанного выражения.

Проход	Такт	x	y	L	R	Out	T_i	St_1	St_2
1	1	a	+	▬	+			a	+
	2	b	+	+	+	▬T_1 +	T_1=a+b		
	3	c	*	+	*	▬T_1 +		c	*
	4	d	*	*	*	▬T_1 +T_2*	T_2=c*d		
	5	l	*	*	*	▬T_1 +T_2*		l	*
	6	f	*	*	*	▬T_1 +T_2*T_3+	T_3=l*f		
	7	g	▬	*	▬	▬T_1 +T_2*T_3+g▬			
2	8	T_1	+	▬	+	▬T_1 +		T_1	+
	9	T_2	*	+	*	▬T_1 +T_2*		T_1 T_2	+ *
	10	T_3	+	*	+	▬T_1 +T_4+	T_4=T_2*T_3	T_1	+
	11	g	▬	+	▬	▬ T_4+T_5	T_5=T_1+g		
3	12	T_4	+	▬	+	▬ T_4+		T_4	+
	13	T_5	▬	+	▬	▬ T_6▬			

Дерево выходного выражения изображено ниже на рис. 2.4.

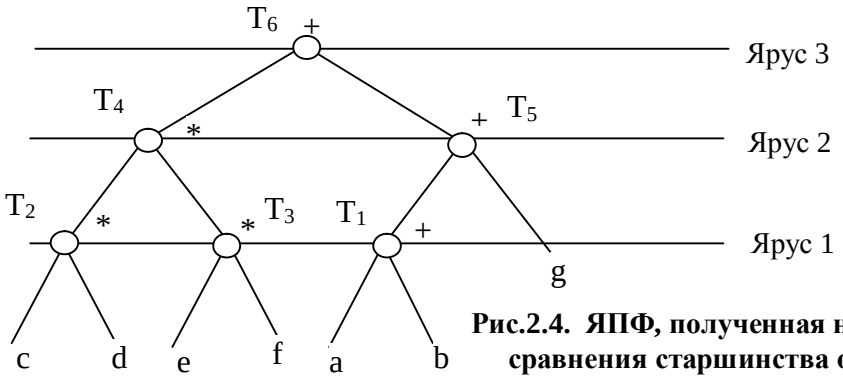


Рис.2.4. ЯПФ, полученная на основе сравнения старшинства операций

Доказано, что описанный алгоритм дает в результате выражение с минимальным временем выполнения. Однако он имеет ряд существенных недостатков, главный из которых—многопроходность: он требует столько проходов, каково время выполнения сгенерированного выражения. Чтобы избавиться от многопроходности, нужно в процессе разбора «нести» попутно информацию об уровне формируемых конструкций. Имеется однопроходная версия метода.

Алгоритм распараллеливания программы базового блока. Построение ЯПФ для отрезка программы производится путем определения зависимости пар смежных команд при анализе программы сверху вниз согласно алгоритму на рис.2.5.

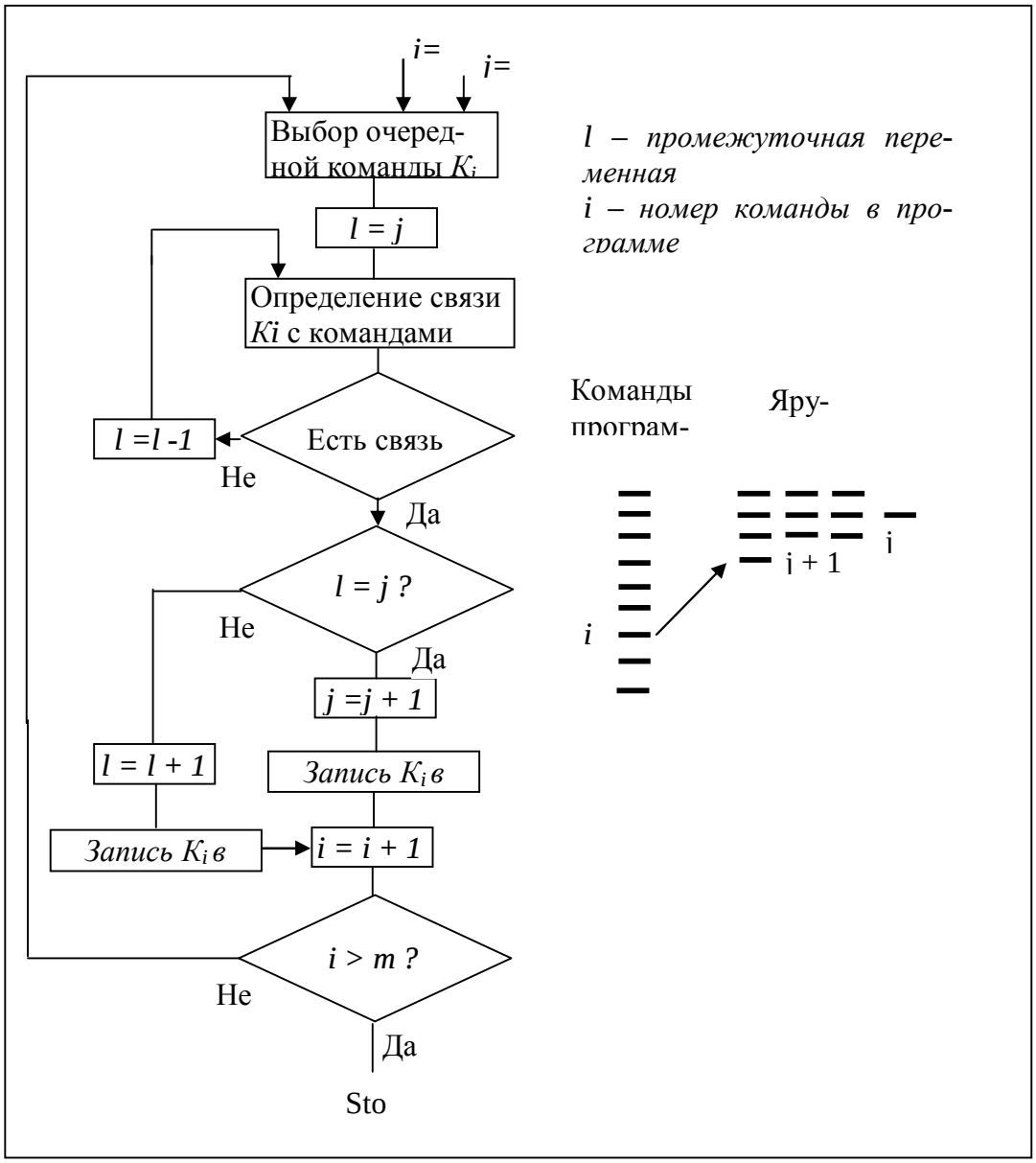


Рис.2.5. Алгоритм распараллеливания арифметических выражений на основе старшинства операций

Планирование ЯПФ базовых блоков. Спланировать – означает указать, на каком процессоре и в каком такте будет запущен на исполнение некоторый блок вычислений. На рис.2.6 представлен пример планирования одного яруса ЯПФ, содержащего 4 блока разной длительности вычислений на 2 процессора. В варианте планирования б) ярус ЯПФ выполняется за 10 тактов, в варианте в) – за 6 тактов. Конечно, этот пример очень простой, но он показывает, что планирование может давать значительно отличаюшиеся по эффективности результаты.

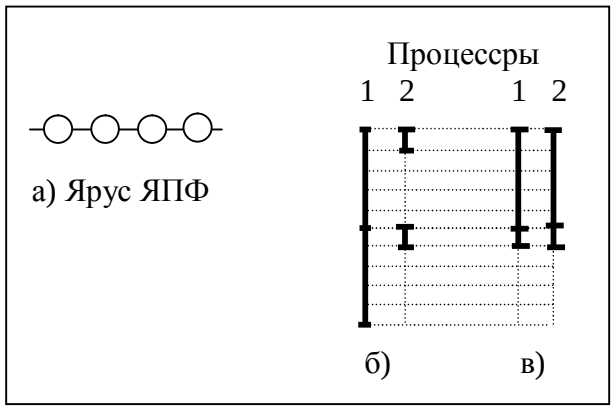


Рис.2.6. Пример планирования некоторой ЯПФ

Алгоритмы планирования связаны с природой *графа управления*: состоит ли он из одного или нескольких базовых блоков, является ли граф управления циклическим или ациклическим графом в случае нескольких базовых блоков.

Алгоритмы для одиночных базовых блоков называются *локальными*, остальные – *глобальными*. Алгоритмы ациклического планирования связаны с графами управления , которые не содержат циклов, или, что более типично, с циклическими графами, для которых существует самостоятельное управление на каждой обратной дуге графа управления.

Метод списочных расписаний. Планирование для единичного ББ заключается в построении как можно более короткого плана. Поскольку эта проблема имеет переборный характер, внимание было сосредоточено на эвристических алгоритмах планирования. Среди таких алгоритмов наиболее подходящими оказались *списочные расписания.* Алгоритмы планирования по списку при линейном росте времени планирования при увеличении числа планируемых объектов, дают результаты, отличающиеся от оптимальных всего на 10-15%.

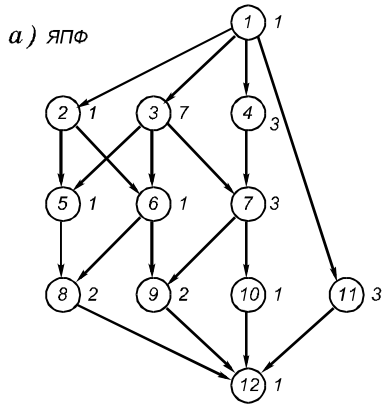
В таких расписаниях оператором присваиваются приоритеты по тем или иным эвристическим правилам, после чего операторы упорядочиваются по убыванию или возрастанию приоритета в виде линейного списка. В процессе планирования затем осуществляется назначение операторов процессорам в порядке их извлечения из списка.

На рис.2.7,а представлена исходная ЯПФ. Номера вершин даны внутри кружков, а время исполнения — около вершин. На рис.2.7,б приведены уровни

вершин. Под уровнем вершины понимается длина наибольшего пути из этой вершины в конечную. Построим для примера два расписания из множества возможных. Для каждого из них найдем характеристики всех вершин по соответствущим алгоритмам и примем значения этих характеристик в качестве величин приоритетов этих вершин.

В первом расписании величина приоритета вершины есть ее уровень. Это расписание УР. Во втором расписании величина приоритета вершины есть ее уровень без учета времени исполнения, то есть здесь веса всех вершин полагаются одинаковыми (единичными).

а) ЯПФ



УР		*УР Без учета веса*	
П	*В*	*П*	*В*
14	*1*	*5*	*1*
13	*3*	*4*	*4*
9	*4*	*4*	*2*
6	*7*	*4*	*3*
5	*2*	*3*	*5*
4	*6*	*3*	*7*
4	*5*	*3*	*6*
4	*11*	*2*	*10*
3	*9*	*2*	*9*
3	*8*	*2*	*8*
2	*10*	*2*	*11*
1	*12*	*1*	*12*

б) Приоритеты и списки вершин

(П— приоритеты, В — номера вершин)

в) Уровни вершин

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*номера вершины*
14	*5*	*13*	*9*	*4*	*4*	*6*	*3*	*3*	*2*	*4*	*1*	*уровни*

г) Реализация расписаний (Р1, Р2 — процессоры, Х — простой процессора)

УР (по уровню вершин)

P1	*1*	*3*	*3*	*3*	*3*	*3*	*3*	*7*	*7*	*7*	*9*	*9*	*12*
P2	*X*	*4*	*4*	*4*	*2*	*11*	*11*	*6*	*5*	*8*	*8*	*10*	*X*

УР без учета веса вершин

P1	*1*	*4*	*4*	*4*	*11*	*11*	*11*	*X*	*X*	*5*	*6*	*8*	*8*	*9*	*9*	*12*
P2	*X*	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*7*	*7*	*7*	*10*	*X*	*X*	*X*

Рис.2.7. Примеры двухпроцессорных списочных расписаний

Об оптимальности расписания можно судить по количеству простоев процессоров. Первое расписание дает 2, второе — 5.

2.2. Классификация Фишера для мелкозернистого паралеллизма

Мелкозернистый параллелизм – это параллелизм в обработке смежных команд, операторов, небольших векторов данных.

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 можно классифицировать следующим образом (таб.2.1):

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. 2.1. 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

Некоторые из этих процессоров рассмотрены в книге [11].

Суперскалярные процессоры. 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*. За счет уменьшения объема работы в одном такте возникает возможность повысить частоту тактирования процессора. При этом, чем длинее конвейер, тем выше частота. Современные конвейеры имеют длину до 30 этапов.

Дальнейшая цель – построение процессора с несколькими конвейерами, что позволяет запускать несколько команд каждый такт. Такой процессор называется *суперскалярным*.В качестве примера приведем схему микропроцессора Pentium (рис.2.8).

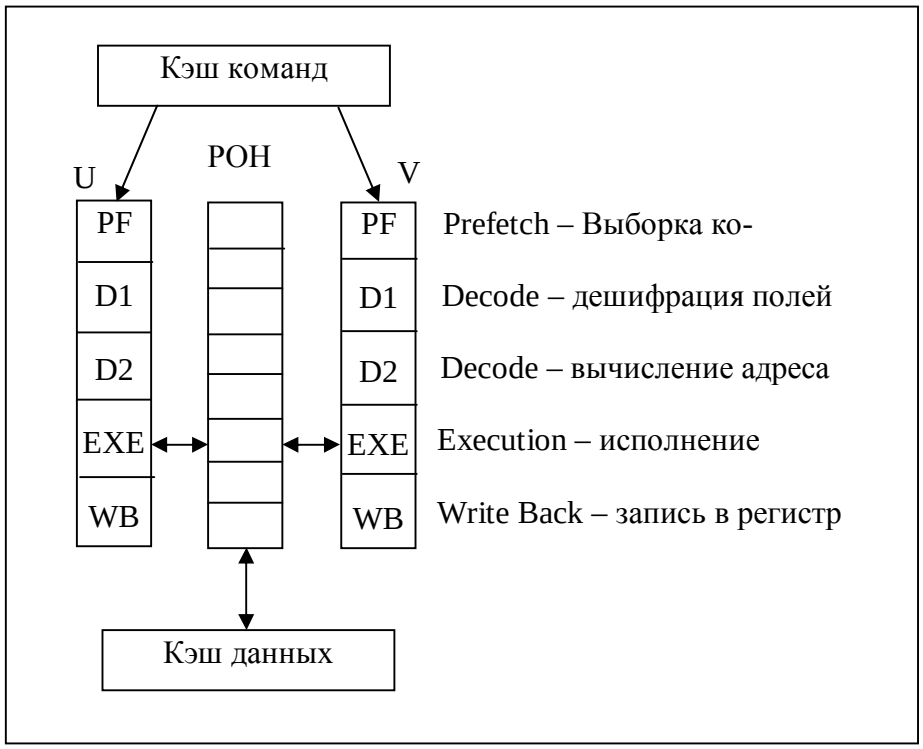


Рис.2.8. Схема микропроцессора Pentium

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.

Процессоры потока данных. Существует два типа управления выполнением программы: *управление от потока команд* (IF – Instruction Flow) и *управление от потока данных* (DF – Data Flow). Если в ЭВМ первого типа используется традиционное выполнение команд по ходу их расположения в программе, то применение ЭВМ второго типа предполагает активацию операторов по мере их текущей готовности.

В случае DF все узлы информационного графа задачи представляются в виде отдельных операторов:

КОП О1, О2, АЗ, БС

где О1, О2 — поля для приема первого и второго операндов от других операторов; АЗ – адрес (имя) оператора, куда посылается результат; БС – блок событий. В БС записывается число, равное количеству операндов, которое нужно принять, чтобы начать выполнение данного оператора. После приема очередного операнда из БС вычитается единица, когда в БС оказывается нуль, оператор начинает выполняться. Программа полностью повторяет ИГ, но ее операторы могут располагаться в памяти в произвольном порядке. Выполняться они будут независимо от начального расположения строго в соответствии с зависимостью по данным. Это и есть управление потоком данных. Считается, что такая форма представления ИГ обеспечивает наибольший потенциальный параллелизм.

В случае зависимостной архитектуры компилятор или программист выявляют параллелизм в программе и представляют его аппаратуре путем описания зависимостей между операциями в машинной программе. Аппаратура все же еще должна определить на этапе исполнения, когда каждая операция становится независимой от всех других операций, и тогда выполнить планирование.

Но прежде рассмотрим вопрос о влиянии количества регистров общего назначения РОН на величину параллелизма. Существует несколько видов памяти с разным временем доступа и объемом хранимой информации:

Тип памяти	Время доступа, такты	**Объем**
РОН	1	обычно до 16 байт
Кэш	1 - 2	сотни килобайт
Оперативная память	20 - 50	гигабайты

Регистры – самая быстрая память. Обычно их немного. Рассмотрим следующую ситуацию. Пусть требуется сложить четыре числа - $\sum_{i=1}^{4} a_i$. Рассмотрим эту операцию для двух случаев: в микропроцессоре имеется 2 или 4 РОН. В программе для двух РОН из-за недостатка регистров приходится промежуточные результаты записывать в память, поэтому программы удлиняется и в нейт нет параллелизма. Если предположить, что одно обращение к памяти занимает 3 такта, а сложение – 1 такт, то приведенная программа выполняется за 30 тактов.

Программа представлена ниже.

```
mov r1, [a1]
mov r2, [a2]
r1=r1+r2
mov [b1], r1

mov r1, [a3]
mov r2, [a4]
r1=r1+r2
mov [b2], r1

mov r1, [b1]
mov r2, [b2]
r1=r1+r2
mov [c], r1
```

Увеличение числа регистров до 4 –ех устраняет использование промежуточных переменных и позволяет параллельное выполнение операций:

```
mov r1, [a1]
mov r2, [a2]
mov r3, [a3]
mov r4, [a4]
r1=r1+r2, r3=r3+r4
r1=r1+r2
mov [c], r1
```

Итого, 17 тактов, что значительно меньше, чем в предыдущем случае. При увеличении объема данных и числа РОН выигрыш будет намного больше.

Микропроцессор Pentium Pro компании Intel (рис.2.9) построен по принципу управления от потока данных (DF), отсюда и получил название потокового. Главное в потоковом процессоре – выполнить команду сразу, как только станут

доступны входные операнды и освободятся необходимые функциональные устройства.

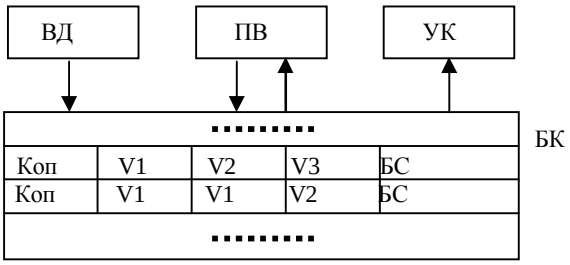


Рис. 2.9. Структура МП Pentium Pro с произвольым порядком выполнения команд

Блок ВД (выборка и декодирование команд) по существу является микропрограммной частью компилятора. Он обеспечивает:

- Чтение команд из КЭШ и их преобразование из формата *i86* в формат DF (управление от потока данных)
- Запись команд в буфер команд в буфер команд БК. В БК команды представлены в трехадресном формате. Поскольку в системе команд *i*86 мало РОН (всего 8), то для устранения ложных зависимостей по данным в МП Pentium Pro введено 40 дополнительных регистров, которые недоступны программисту. Они используются аппаратурой для временного хранения результатов. Обозначим эти регистры временного хранения через V. Тогда на рисунке V1, V2 и VP обозначают соответственно номера регистров для хранения первого, второго операндов и результата.

Блок ПВ (планирование и выполнение) является центральным блоком МП Pentium Pro. Именно он выполняет команды в порядке их готовности. ПВ содержит несколько АЛУ и устройств обращения к памяти. За один такт ПВ выполняет следующие действия:

- Выделяет в БК команды, готовые к исполнению.
- Планирует и назначает на исполнение до пяти команд, поскольку в ПВ имеется пять исполнительных устройств.
- Выполняет эти команды.
- Передает результаты в блок БК, вычитает единицу из БС и в случае воз можности устанавливает в командах признак готовности.

Чтобы блок ПВ мог выполнять за один такт до 3...5 команд, необходимо, чтобы в БК находилось до 20...30 команд. По статистике среди такого объема команд в среднем имеется имеется 4...5 команд условных переходов. Следовательно, в БК находится некоторая трасса выполнения команд. Выбор таких наиболее вероятных трасс является новой функцией МП с непоследовательным выполнением команд. Эта функция выполняется в блоке ВД на основе расширенного до 512 входов буфера истории переходов.

Поскольку реально вычисленный в ПВ адрес перехода не всегда совпадает с предсказанным в блоке ВД, то вычисление в ПВ выполняется условно, т. е. результат записывается в регистр временного хранения. Только после того, как установлено, что переход выполнен правильно, блок удаления команд УК выводит из БК все выполненные команды, расположенные за командой условного перехода, преобразует их в формат системы *i*86 и производит запись результатов по адресам, указанным в исходной программе.

Независимостные архитектуры. К таким архитектурам относятся процессоры со сверхдлинной командой (СДК), однако для них чаще используется название – процессоры VLIW (Very Long Instruction Word). Чтобы выполнить операции параллельно, система должна определить, являются ли операции независимыми от других. Суперскалярный и потоковый процессоры представляют два способа извлечения этой информации на этапе исполнения. В случае потокового процессора эта явная информация используется, чтобы определить момент запуска команды. Суперскалярный процессор делает то же самое, однако, из-за отсутствия явной информации он вынужден сначала определять зависимости между командами. Напротив, для независимостных архитектур компилятор определяет параллелизм в программе и привязывает его к аппаратуре путем указания, какие операции независимы от других. Эта информация есть прямое указание для аппаратуры, какие операции можно проводить в том же самом цикле. Вместо описания всех независимых команд обычно описывают только часть этих независимых команд, которые предполагается выполнить одновременно.

Если архитектура дополнительно требует, чтобы программа описывала, где (в каком функциональном устройстве) и когда (в каком такте) операция выполнялась, тогда оборудованию вообще не надо тратить времени на принятие этих решений, и выходная запись последовательности исполненных операций полностью совпадает с входной программой. Построенные к этому времени VLIW *процессоры* обладают именно такой архитектурой. Программа для VLIW процессора точно описывает, на каком функциональном устройстве и в каком такте операция должна быть запущена, так чтобы она была независима от всех запускаемых в данном такте операций и от других еще незаконченных операций.

Для VLIW процессора важно ввести различие между командой и операцией. Операция есть единица вычислений (сложение, выборка из памяти, переход) для последовательной машины. VLIW команда (сверхдлинная команда) есть ряд операций, которые предполагается запустить одновременно. Задачей компилятора является определение того, какие операции должны входить в команду. Этот процесс называется *планированием.* Все допустимые операции упаковываются в единую команду. Порядок операций внутри команды задает устройства, на которых каждая операция должна выполняться.

Первым примером такой архитектуры является архитектура микропроцессора компании Intel под названием *Itanium* (ранее Merced), разработанная в соответствии с концепцией *IA-64* (Intel Architecture для 64 разрядов) и структура СДК микропроцессора C6 компании Texas Instruments.

2.3. VLIW.

Архитектура процессора С6 (Texas Instruments). Особенностью процессора является наличие двух практически идентичных вычислительных блоков, которые могут работать параллельно и обмениваться данными. Каждый блок содержит четыре фунциональных устройства, которые также могут работать параллельно.

На рис.2.10 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 А и В, 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.

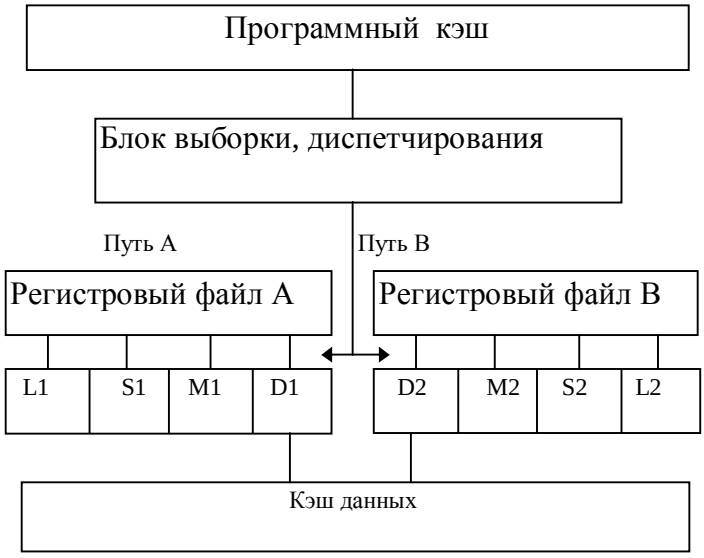


Рис.2.10. Структура микропроцессора С6

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.2.11.

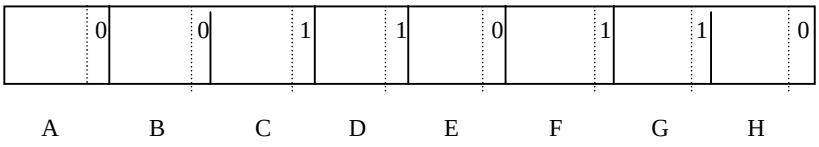


Рис.2.11. Частично последовательный пакет выборки

Состав исполнительных пакетов для этой команды представлен ниже:

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

2.4. Технология MMX

В 1997 году компания Intel выпустила первый процессор с архитектурой SIMD с названием Pentium MMX (MultiMedia eXtension), который реализует частный случай векторной абработки. При разработке MMX специалистами Intel как в области архитектуры, так и в области программного обеспечения был обследован большой объем приложений различного назначения: графика, пол-

ноэкранное видео, синтез музыки, компрессия и распознавание речи, обработка образов, сигналов, игры, учебные приложения. Анализ участков с большим объемом вычислений показал, что все приложения имеют следующие общие свойства, определившие выбор системы команд и структуры данных:

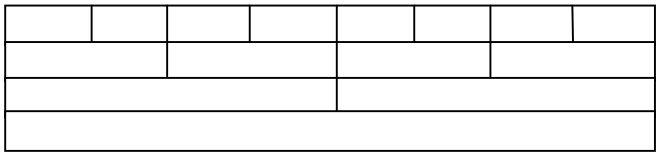
- небольшая разрядность целочисленных данных (например, 8-разрядные пикселы для графики или 16-разрядное представление речевых сигналов);
- небольшая длина циклов, но большое число их повторений;
- большой объем вычислений и значительный удельный вес операций умножения и накопления;
- существенный параллелизм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

Это и определило новую структуру данных и расширение системы команд.

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) - 8
W (Word) - 16
DW (Double Word)
Q (Quad Word) - 64

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.

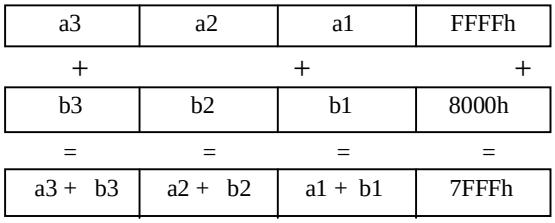
Арифметику насыщения легче всего определить, сравнивая с режимом циклического возврата. В режиме циклического возврата результат в случае переполнения или потери значимости обрезается. Таким образом перенос игнорируется. В режиме насыщения, результат при переполнении или потери значимости, ограничивается. 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-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.

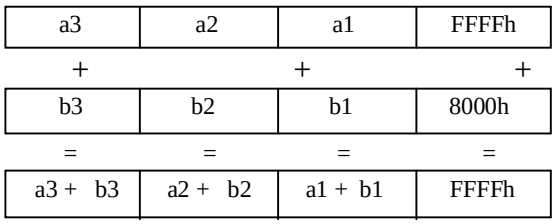
Пример 1. Пример представляет упакованное сложение слов без переноса. По этой команде выполняется одновременное и независимое сложение четырех пар 16-разрядных операндов. На рисунке самый правый результат превышает значение, которое можно представить в 16-разрядном формате. FFFFh + 8000h дали бы 17-разрядный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



Paddw (Packed Add Word)

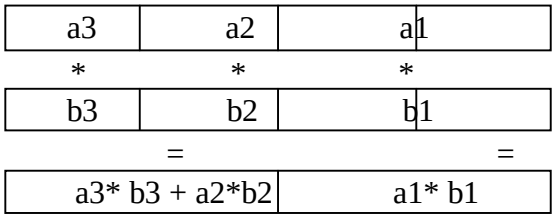
Пример 2. Этотпример иллюстрирует сложение слов с беззнаковым насыщением. Самая правая операция дает результат, который не укладывается в 16 разрядов, поэтому имеет место насыщение. Это означает, что если сложение

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)

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).

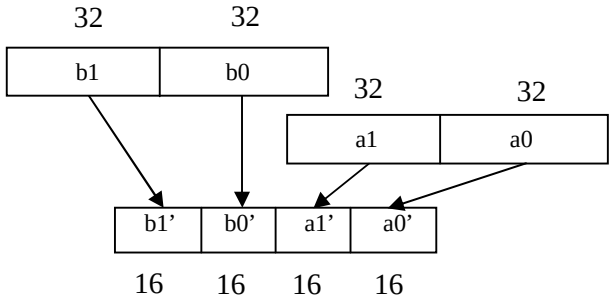
23	45	16	34
gt?	gt?	gt?	gt?
31	7	16	67
=	=	=	=
0000h	FFFFh	0000h	0000h

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.

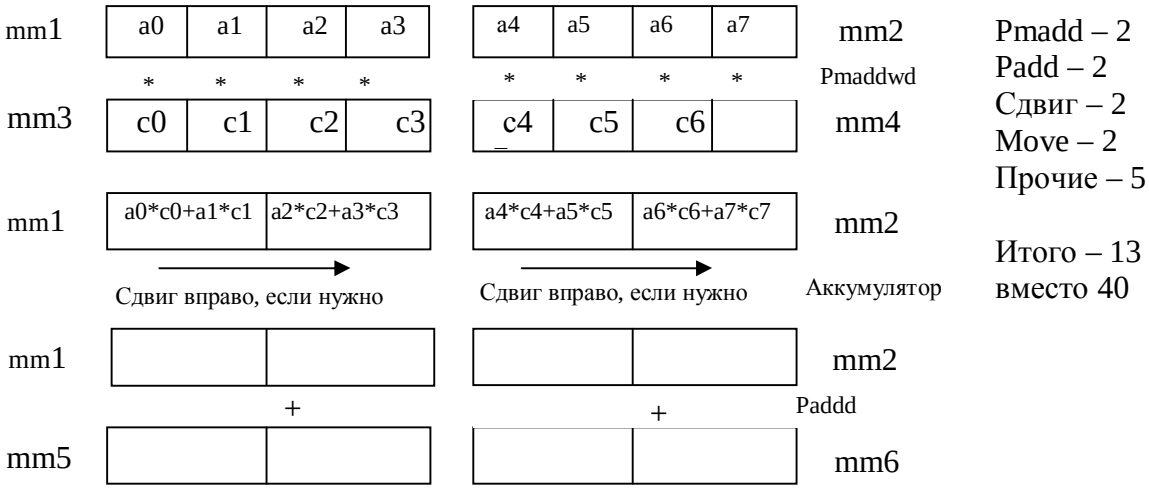
Такой фильтр обычно требует ряда умножений коэффициентов на соответствующие значения пикселов с последующей аккумуляцией полученных значе-

ний. 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.

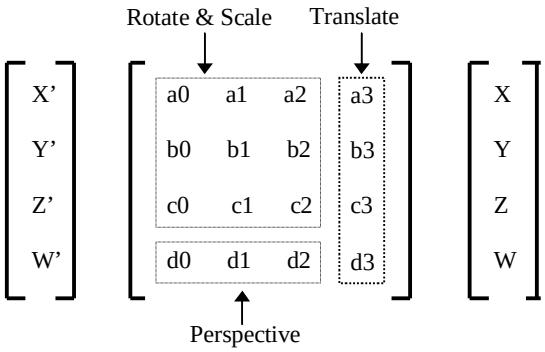
Приводимый ниже пример показывает, как команда PMADD помогает сделать алгоритм более быстрым. Команда PMADD позволяет выполнить сразу четыре умножения и два сложения для 16-разрядных операндов.Для завершения умножения с накоплением требуется еще команда PADD.



Если предположить, что 16-разрядное представление входных данных является удовлетворительным по точности, то для получения точечного 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.

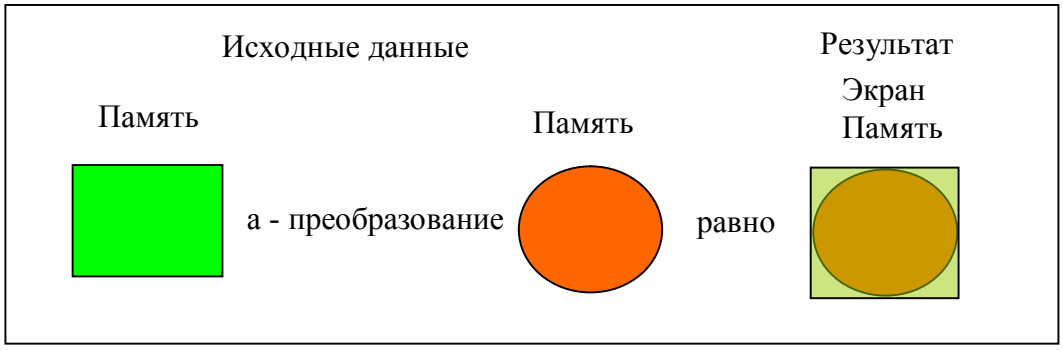


X’ = a0*x + a1*y + a2*z

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 изобра-

жения 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:

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

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 ММХ.

Команда распаковки принимает первые 4 байта для красного цвета, преобразует их в 16-разрядные элементы и записывает в 64-разрядный ММХ 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 для технологии ММХ 525 млн. операций, а без нее - 1.4 млрд. операций.

Альфа-смешивание позволяет разработчикам игр реалистично представить движение гоночного автомобиля в тумане, 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- мешивании. Техника обработки изображений с помощъю альфа-смешивания позволяет создавать различные комбинации для двух объектов А и Б (таб.2).

Таблица 2. 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 A есть там, где B непрозрачен A*alpha(B)
A out B	Образ A есть там, где B 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)))

Технология MMX значительно увеличивает возможности мультимедийных приложений, обеспечивая параллельную обработку для новых типов данных.

2.5. Расширения SSE

Технология MMX имеет ряд расширений, называемых SSE (*Streaming SIMD Extensions*). SSE - это расширение инструкций процессора для потоковой обработки в режиме SIMD (*Single Instruction Multiple Data*), т.е. когда требуется применять однотипные операции к потоку данных. Технология SSE позволила преодолеть проблемы MMX - при использовании MMX невозможно было одновременно использовать инструкции сопроцессора, так как его регистры задействовались для MMX и работы с вещественными числами.

В общем случае, к архитектуре процессора добавляется ряд инструкций и несколько 128-битных регистров с различной интерперетацией. Изначально каждый регистр трактуется как два значения с плавающей точкой двойной точности (2*64-бит). Однако, операции могут применяться практически ко всем типам, "помещающимся" в 16 байт. Это означает, например, что появляется возможность одновременно сложить или умножить с помощью всего одной инструкции два операнда из четырех чисел с плавающей точностью одинарной точности, двух - с двойной, двух 64-битных целочисленных, 16 8-битных целых и т.п.

Существуют следующие расширения: SSE, SSE2, SSE3 и SSE4, о которых имеется обширная информация в интеренете.

Глава 3. КРУПНОЗЕРНИСТЫЙ ПАРАЛЛЕЛИЗМ

Крупнозернистый параллелизм обеспечивается за счет параллелизма независимых программных ветвей, подпрограмм, потоков (нитей) внутри программ, служебных программ и реализуется процессорами или ядрами многоядерных процессоров.

3.1. Классификация Флинна

Существует много классификаций для крупноформатного параллелизма [1]. Одна из них – классификация Флинна, предложенная в 1970 году, получила большое распространение. Флинн ввел два рабочих понятия: поток команд и поток данных. Под потоком команд упрощенно понимают последовательность выполняемых команд одной программы. Поток данных – это последовательность данных, обрабатывемых 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. Здесь может использоваться конвейерная конвейерная обработка в единственном потоке команд.

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, реализованноq в машинах CRAY.

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 и *(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. Этот класс разделяется на два больших подкласса:

1. С общей для всех процессоров памятью. Это упрощает оборудование, но количество процессоров ограничено пропускной способностью памяти.
2. С индивидуальной памятью для каждого процессора. Число процессоров не ограничивается, но межпроцессорный обмен данными снижает потенциальное ускорение.

Примером MIMD является рассмотренная в разделе 2 технология MMX для мелкозернистого параллелизма. С течение времени появилась разновидность этой технологии, называемая SPMD (Single Program – Multiple Data), которая

используется в многопроцессорных системах с обшей памятью SMP (Symmetric Multiprocessing).

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

3.2. Арифметические конвейеры

Векторно-конвейерная ЭВМ CRAY [12] относится к классу ОКМД ЭВМ. Но ее отличием от других ЭВМ этого класса является наличие развитой системы арифметических конвейеров, которые и обеспечивают ее быстродействие.

На рис.3.1 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.3.2 представлена схема конвейера для сложения потока чисел с плавающей запятой, а на рис.3.3 – для умножения чисел. Такие конвейеры можно построить практически для всех команд.

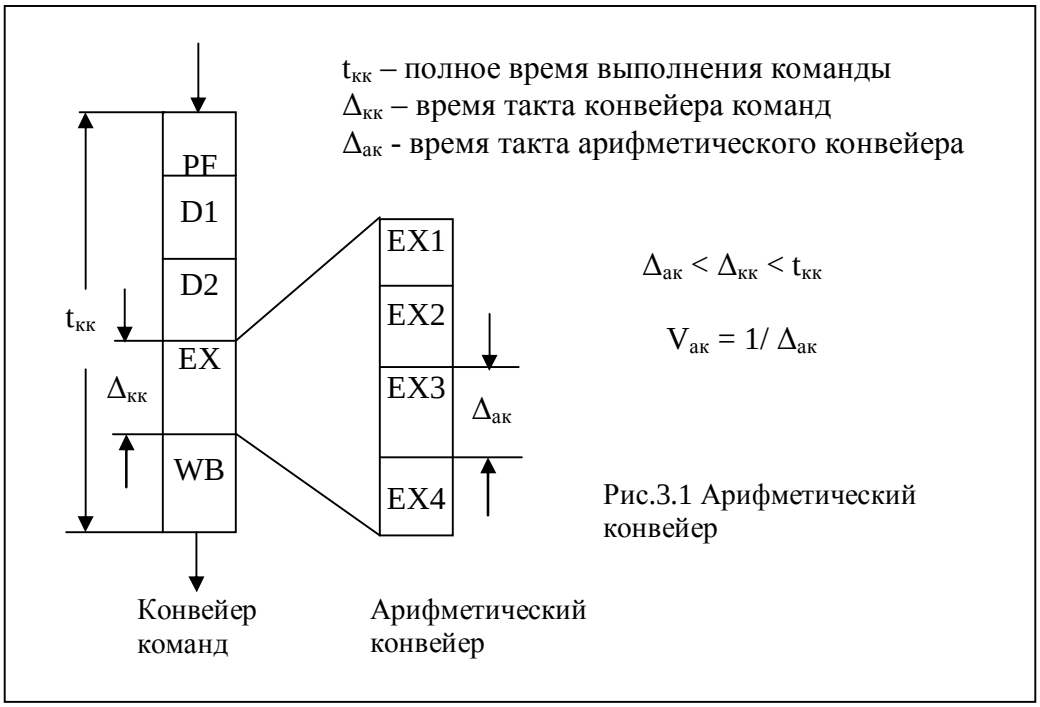


Рис.3.1 Арифметический конвейер

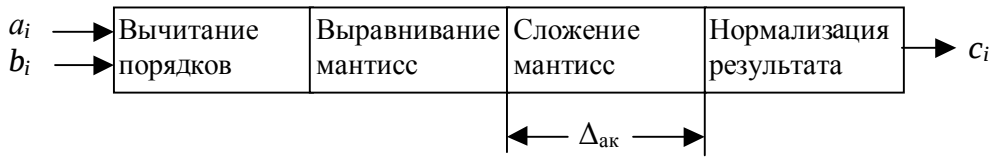


Рис.3.2. Арифметический конвейер для сложения чисел с плавающей запятой

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 CRAY-1, которая 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”.

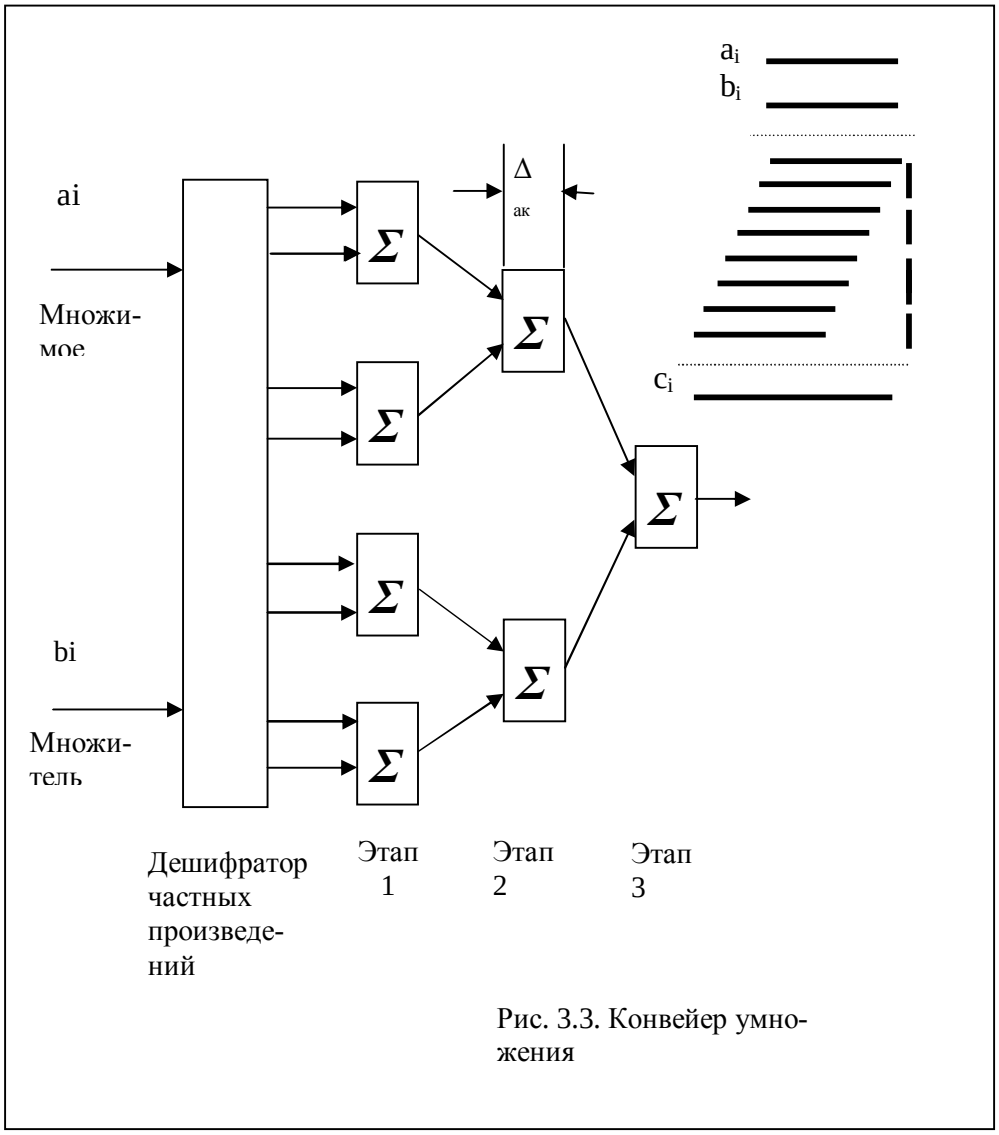


Рис. 3.3. Конвейер умножения

ОКМД ЭВМ CRAY [1,12] обладает рядом особенностей:

- Класс – ОКМД, система с общей памятью. Все АЛУ – конвейерные.
- Большой объем векторных регистров (сотни в новых машинах) снижает нагрузку на общую память.
- Выполнение команд в скалярном процессоре управляется потоком данных.
- Со временем эта архитектура была реализована в виде одного микропроцессора в составе многопроцессорной ЭВМ.Такие ЭВМ занимают по быстродействию первые места в списке TOP 500.

ЭВМ CRAY состоит из скалярного и векторного процессоров (рис.3.4). Скалярный процессор выбирает из памяти и выполняет скалярные и векторные команды, но скалярные – целиком, а арифметическую часть векторных команд передает в векторный процессор.

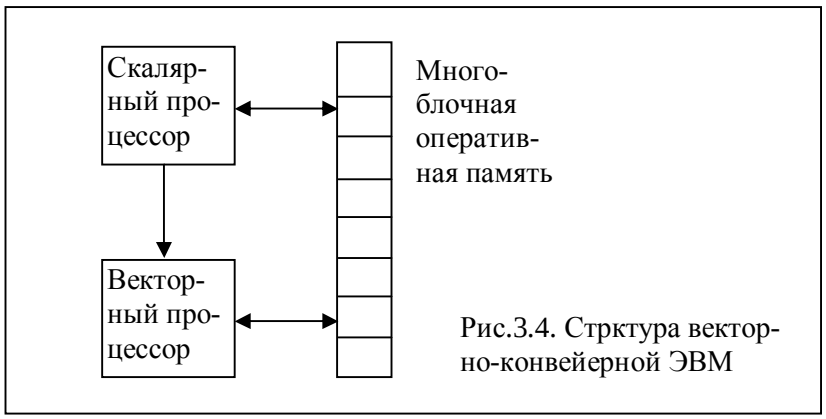
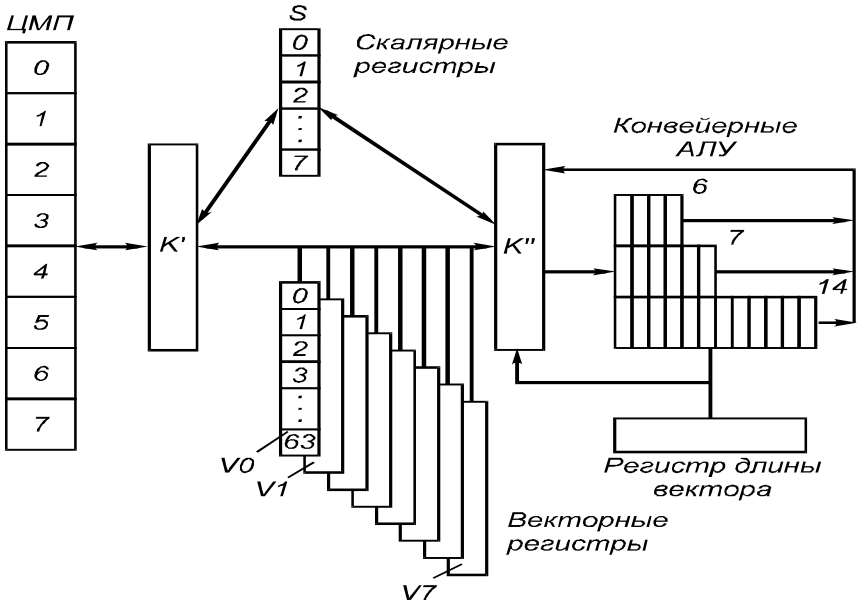


Рис.3.4. Стрктура векторно-конвейерной ЭВМ

Схема векторного процессора ЭВМ CRAY представлена ниже.



3.5. Структура ЭВМ CRAY

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)
  (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
   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. 3.6, 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.

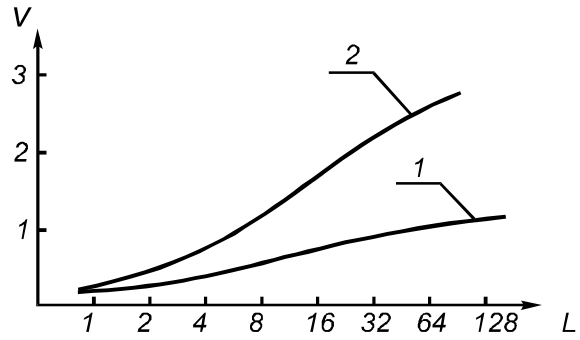


Рис.3.6. Зависимость быстродействия векторного процессора от длины вектора: $m_{ЦМП} = 4$; $m_* = 7$; $m_+ = 6$

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.3.7).

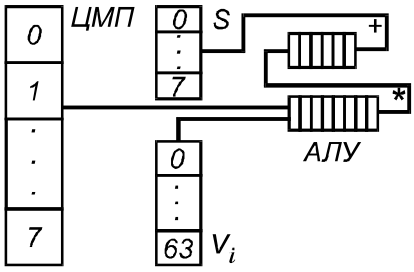


Рис.3.7. Выполнение операции зацепления в конвейерном процессоре

Для команды ЗЦ получаем:

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

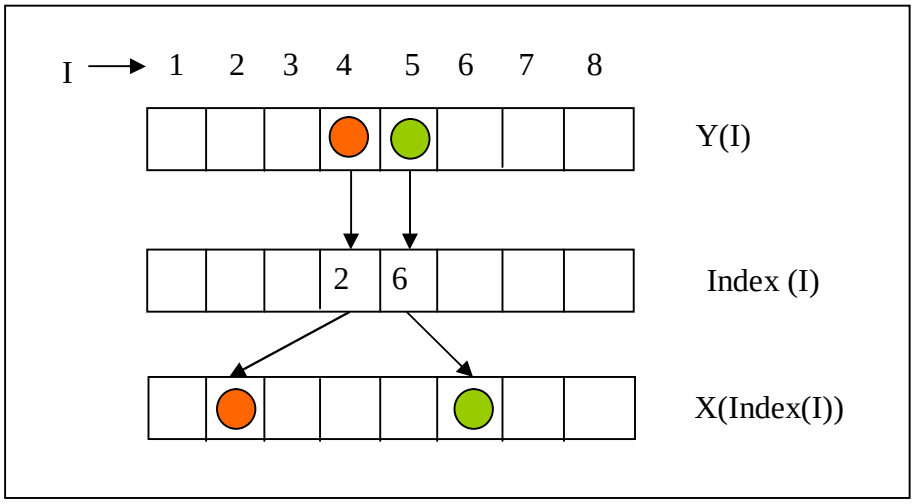
где n — число одновременно выполняемых операций. В случае команды ЗЦ $n = 3$ (рис.6, кривая 2). При $L>>m_i$ и Δt = 1...2 нс в зацеплении быстродействие равно 1500...3000 млн оп/с.

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 работать эффективно. 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).

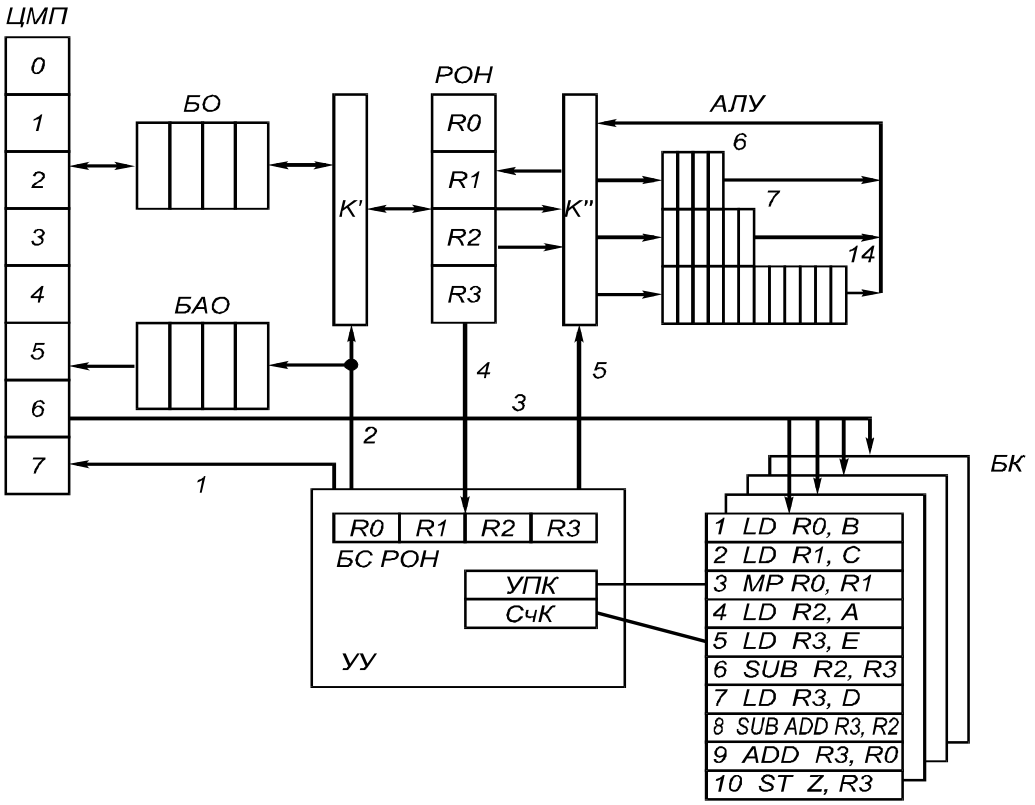


Рис. 3.8 Организация скалярного конвейерного процессора:

Рассмотрим структуру и функционирование *скалярного конвейерного процессора* в целом (рис.8). На рисунке обозначено:

- ЦМП — центральная многоблочная память;
- БАО — буфер адресов операндов;
- АЛУ — конвейеризованные АЛУ для операций с плавающей точкой;
- 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.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. Буфер команд имеет многостраничную структуру, что позволяет во время работы УУ с одной страницей производить заранее смену других страниц. Для изучения работы процессора

(рис.3.8) 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 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.3.9). 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.

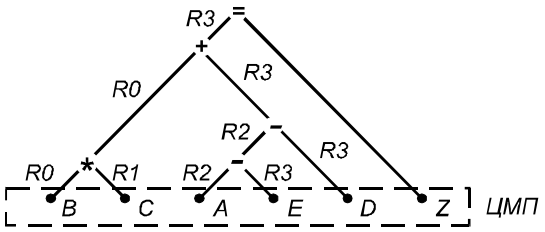


Рис.3. 9. Информационный граф программы: Ri — регистр общего назначения

В табл.1 приведено описание нескольких тактов работы процессора. Принято, что выборка операнда из ЦМП занимает четыре такта, сложение – 6, а умножение – 7 тактов. Считается, что за один такт процессора устройство управления запускает на исполнение одну команду или просматривает в программе до четырех команд. 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. Произво-

дится анализ условий запуска следующей (по состоянию СчК) команды. Команда 4 может быть запущена и запускается.

Таблица 3.1.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	-

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. Предположим, что такт процессора равен 10 нс. Тогда на выполнение одной команды тратится 15 нс, что соответствует быстродействию $V = 70$ млн оп/с.

3.4. Многопроцессорные системы с общей памятью или Symmetric Multiprocessing (SMP)

Многопроцессорным ЭВМ посвящена большая литература. Эту информацию можно посмотреть на сайте [6].

Схема многопроцессорной системы с общей памятью представлена ниже:

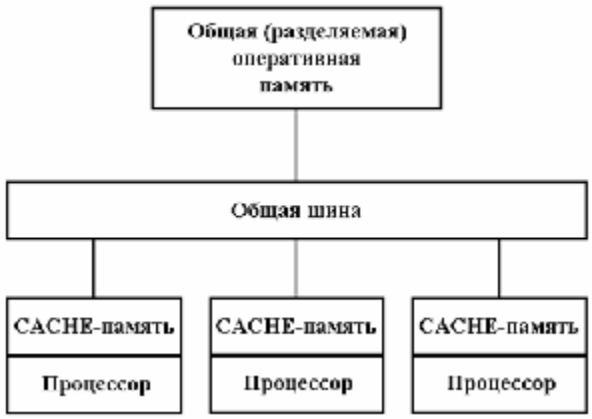


Рис. 3.10. Схема многопроцессорной системы с общей памятью

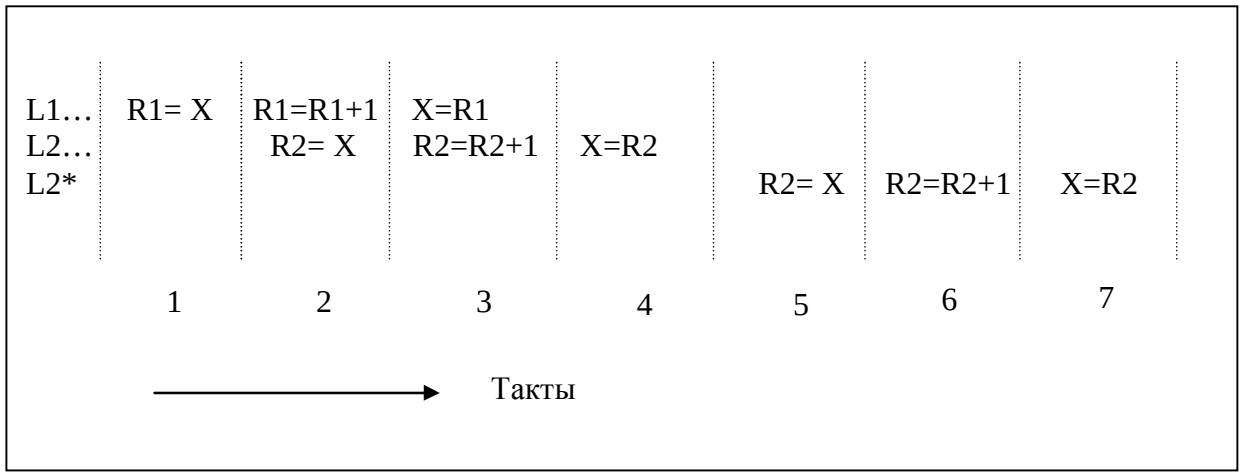
Наличие общей памяти вызывает как положительные, так и отрицательные последствия:

- 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. Для устранения этого существенного недостатка используются развитые системы кэширования, то есть внутрипроцессорной быстродействующей памяти для временного хранения промежуточных результатов.
- Несколько процессоров могут одновременно обращаться к общим данным и это может привести к получению неверных результатов. Чтобы исключить такие ситуации, необходимо ввести систему управления доступом в оперативную память, разрешающую обращение к памяти только одному процессу. Это является отличительной особенностью систем с общей памятью.

Управление доступом к памяти. 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*.

Семафоры. Чтобы исключить упомянутую выше ситуацию, необходимо ввести систему синхронизации параллельных процессов.

Выход заключается в разрешении входить в критическую секцию (КС) только одному из нескольких асинхронных процессов. Под критической секцией понимается участок процесса, в котором процесс нуждается в ресурсе. Решение проблемы критической секции было предложено в виде семафоров. Семафором называется переменная *S*, связанная, например, с некоторым ресурсом и принимающая два состояния: 0 (запрещено обращение) и 1 (разрешено обращение). Над *S* определены две операции: *V* и *P*. Операция *V* изменяет значение *S* семафора на значение *S* + 1. Действие операции *P* таково:

- Если $S \neq 0$, то *P* уменьшает значение на единицу;
- Если $S = 0$, то *P* не изменяет значения *S* и не завершается до тех пор, пока некоторый другой процесс не изменит значение *S* с помощью операции *V*;
- Операции V и P считаются неделимыми, т. е. не могут исполняться одновременно.

Приведем пример синхронизации двух процессов, в котором process 1 и process 2 могут выполняться параллельно. Процесс может захватить ресурс только тогда, когда S:=1. После захвата процесс закрывает семафор операции P(S) и открывает его вновь после прохождения критической секции V(S).

```
begin
semaphore S;
S:=1;
process 1:
  begin
     L1:P(S);
     Критический участок 1;
     V(S);
     Остаток цикла, go to L1
  end
process 2:
   begin
      L2:P(S);
      Критический участок 2;
      V(S);
      Остаток цикла, go to L2
   end
end
```

Таким образом, семафор S обеспечивает неделимость процессов *L*i и, значит, их последовательное выполнение. Это и есть решение задачи взаимного исключения для процессов *L*i.

Определение требуемого быстродействия памяти по частоте процессора. Для соблюдения баланса вычислений в процессоре необходимо, чтобы в этом цикле время вычислений в процессоре равнялось времени обращения к памяти. Это минимальное условие обозначает следующее:

$$N*T_{пр} = M*T_{пм}$$

где N – число операций процессора (равно 6 для примера,приведенного выше), M – число обращений к памяти (равно 2), $T_{пр}$ и $T_{пм}$ соответственно - время работы процессора и памяти. Следовательно, для нашего примера требуемая частота памяти должна равняться:

$$F_{пм} = \frac{M}{N} \times F_{пр}$$

Пусть для примера $F_{пр}$ = 1 ГГц, M = 2, N = 6 (как в программе выше), и из памяти выбираются 64-разрядные числа, тогда

$$F_{пм} = 0.33*1ГГц = 330\ МГц,$$

а требуемая пропускная способность памяти q равняется

$$q = 8*F_{пм} = 8*330 = 2.84\ Гбайт/с$$

В многоядерном процесоре Nehalem используется 8 процессоров, поэтому приведенное выше выражениеменяется так:

$$F_{ïï} = 8 \cdot \frac{M}{N} \times F_{ïð}$$

Кроме того, частота Nehalem составляет не 1, а 3 ГГц , поэтому требования к пропускной способности памяти возрастают. Положение спасает развитая система кэшей, что и организовано в процессоре Nehalem.

Следует проводить различие между:

- Системой ОКМД.
- Системой с общей памятью.
- Системой SMP.

Для программирования систем собщей памятью используется высокоуровневый язык **OpenMP,** который позволяет в явном виде указываеть наличие в программа мест, которые можно распараллеливать. Само же распараллеливание производится в процессе трансляции на основе содания потоков, реализующих само распараллеливания.

Пример симметричной мультипроцессорной ЭВМ HP 9000 (взято из [6]).

- **Архитектура.** Система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти либо с помощью общей шины (базовые 2-4 процессорные SMP-сервера), либо с помощью crossbar-коммутатора (HP 9000). Аппаратно поддерживается когерентность кэшей.
- **Примеры.** HP 9000 V-class, N-class; SMP-сервера и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.).
- **Масштабируемость** Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число - не более 32 в реальных системах. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры.
- **Операционная система.** Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы/нити по процессорам (scheduling), но иногда возможна и явная привязка.
- **Модель программирования.** Программирование в модели **общей памяти**. (POSIX threads, OpenMP). Для SMP-систем существуют сравнительно эффективные средства автоматического распараллеливания.

3.5. Многопроцессорные системы с индивидуальной памятью или Массивно-параллельные системы (MPP)

Проблема масштабируемости решается в системах с распределенной (индивидуальной) памятью (рис.3.11), в которых число процессоров практически не ограничено.

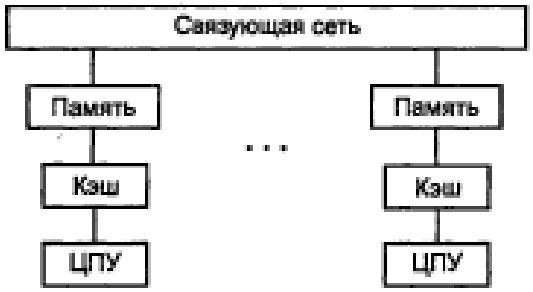


Рис. 1.3. Структура машин с распределенной памятью

Рис.3.11. Схема Эвм с индивидуальной памятью.

Сетевой закон Амдала. Главным фактором, снижающим эффективность таких машин, является потери времени на передачу сообщений.

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

$$R = T_1 / T_n ,$$

где T1 – время решения задачи на однопроцессорной системе, а Tn – время решения той же задачи на n – процессорной системе.

Пусть W = Wск + Wпр, где W – общее число операций в задаче, Wпр – число операций, которые можно выполнять параллельно, а Wск – число скалярных (нераспараллеливаемых) операций. Обозначим также через t время выполнения одной операции. Тогда получаем известный закон Амдала [8]:

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

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

Основной вариант закона Амдала не отражает потерь времени на межпроцессорный обмен сообщениями. Перепишем закон Амдала:

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

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

Это выражение

$$R_C = \frac{1}{a + \frac{1-a}{n} + c}$$

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

Коэффициент сетевой деградации вычислений *c:*

$$c = \frac{W_c \cdot t_c}{W \cdot t} = c_A \cdot c_T ,$$

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.

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

$$z = \frac{R_c}{n} = \frac{1}{1 + c \cdot n} \xrightarrow[c \to 0]{} 1$$

Пример *Z* для ЭВМ СКИФ и первого кластера Beowulf дан на рис.3.12.

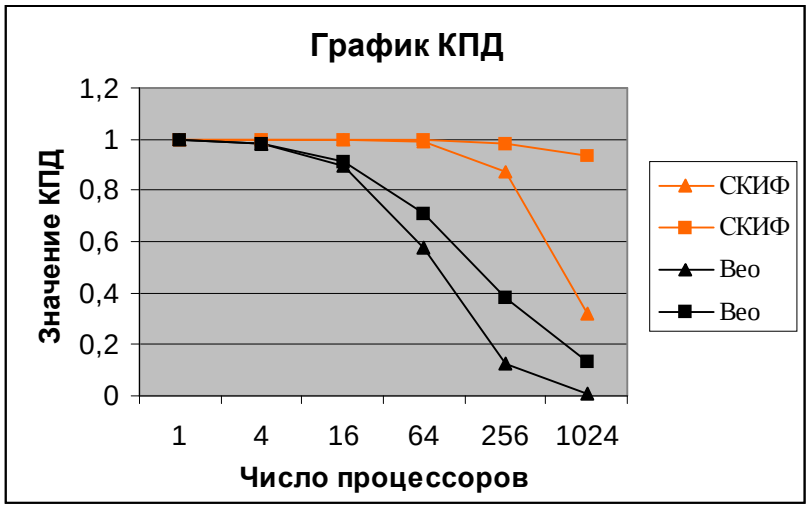


Рис. 3.12. График коэфициента полезного действия для СКИФ и Beowulf. Квадратами отмечены кривые без учета латентности коммутатора, треугольниками – с учетом наличия латентности (2 мкс – для СКИФ, 80 мкс – для Beo).

Приграмирование для систем с передачей сообщений. Наиболее распространенной библиотекой параллельного программирования в модели передачи сообщений является **MPI (Message Passing Interface).** MPI [8] является биб-

лиотекой функций межпроцессорного обмена сообщениями и содержит около 300 функций.

Эффективность систем с обменом сообщениями определяется качеством параллельного алгоритма. Если в нем нет параллелизма, то ЭВМ с множеством процессоров будеработать даже медленнее однопроцессороной ЭВМ.

Пример параллельных ЭВМ с обменом сообщениями (взят из [6]).

Архитектура. Система состоит из однородных *вычислительных узлов*, включающих:

- один или несколько центральных процессоров (обычно RISC),
- локальную память (прямой доступ к памяти других узлов невозможен),
- коммуникационный процессор или сетевой адаптер,
- иногда - жесткие диски (как в SP) и/или другие устройства В/В.

К системе могут быть добавлены специальные узлы ввода-вывода и управляющие узлы. Узлы связаны через некоторую коммуникационную среду (высокоскоростная сеть, коммутатор и т.п.)

Примеры. IBM RS/6000 SP2, Intel PARAGON/ASCI Red, CRAY T3E, Hitachi SR8000, транспьютерные системы Parsytec.

Масштабируемость. Общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain).

Операционная система. Существуют два основных варианта:

1. Полноценная ОС работает только на управляющей машине (front-end), на каждом узле работает сильно урезанный вариант ОС, обеспечивающие только работу расположенной в нем ветви параллельного приложения. Пример: Cray T3E.
2. На каждом узле работает полноценная UNIX-подобная ОС (вариант, близкий к кластерному подходу). Пример: IBM RS/6000 SP + ОС AIX, устанавливаемая отдельно на каждом узле.

Модель программирования. Программирование в рамках модели **передачи сообщений** (MPI, PVM, BSPlib)

3.6. Смешанные архитектуры

В реальных компъютерных системах как правило смешанные архитектуры, использующие в одной многопроцессорной системе различные архитектурные варианты. Показательной с этой точки зрения является организация графических процессоров. Ниже представлена структурная схема графического устройства компании NVIDIA, включающая 8 универсальных процессоров. Под управление каждого процессора находится 16 АЛУ, которые одновременно вы-

полняют векторную операцию, заданную универсальным процессором, что соответствует архитектуре SIMD. Сами процессоры могут работать по одной и той же или разным программам. Такую архитектуру называют multi – SIMD.

Рис.1. Общая схема архитектуры G80.

Здесь: ВПВ – выборка из потока вершин (Vertex Thread Issue); ВПГ – выборка из потока геометрии (Geometry Thread Issue); ВПП – выборка из потока пикселей (Pixel Thread Issue); ПУП – процессор управления потоками (Thread Processor); SP - потоковый процессор (Streaming Processor); TF – блок фильтрации текстур (Texture Filtering Unit); L1 – кэш-память первого уровня; L2 – кэш-память второго уровня; FB – шина обмена данными (Front Bus).

На этом общеизвестном рисунке представлена схема графического процессора G80 компании NVIDIA, на котором впервые заявлена поддержка технологии CUDA. В дальнейшем появились более совершенные графические процессоры, но в G80 представлено все, что обеспечивает возможность выполения неграфических вычислений. На рисунке много обрзначений, отражающих графическую природу G80, для неграфических применений важно следующее:

- G80 содержит 8 унивесальных процессоров (УП), каждый из которых управляет 16-ю потоковыми процессорами (зеленые квадратики внутри УП), которые по существу являются скалярными АЛУ. Таким образом, на кристалле имеется 128 АЛУ, то есть 128 ядер, пригодных для неграфических вычислений. В современных процессорах число таких ядер составляет сотни и даже тысячи.
- G80 имеет собствнную глобальную память (синие прямоугольнички), доступную всем УП. Эта память достаточно медленная.
- Каждый УП имеет разделяемую быструю память с быстрым доступом для всех АЛУ данного УП.

- В параллельных системах для обмена данными обычно используется быстрый коммутатор. От его быстродействия зависит эффективность вычислений. В G80 нет коммутатора. Быстрый обмен данными осуществляется только через разделяемую память внутри КП, а дальний обмен выполняется медленн, через кэши и глобальную память. Из этого следует, что что эффективность примениненения графических процессоров зависит от параллелизма задачи. Графические процессоры идеальны для задач с массовым параллелизмом (математические задачи, моделирование , графика и др.).

Графические процессоры по своей организации относятся к подклассу multi – SIMD, которая весьма удобна для создания многоядерных процессоров, поскольку в ней основной объем оборудования составляют АЛУ. Известно, что АЛУ требуют для своего построения в несколько раз меньше транзисторов, чем универсальный процессор. По своей функциональности АЛУ является полноценным ядром, следовательно, при той же площади кристалла графической микросхемы на нем можно разместить в несколько раз больше ядер, чем на кристалле с MIMD архитуктурой. Поэтому архитектура multi – SIMD получает широкое распространение. Однако, привязка к графике заметно усложняет программирование.

В подклассе multi – SIMD смешаны архитектуры:

- SIMD + MIMD = multi SIMD
- Крупное зерно (универсальный процессор) + Мелкое зерно (АЛУ)

Глава 4. СРЕДСТВА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

4.1. Параллельные алгоритмы

Вычислительные алгоритмы обладают различным уровнем параллелизма. Для некоторых классов задач имеется качественная оценка величины параллелизма. Некоторые результаты такой оценки представлены в следующей таблице [12].

Характеристики некоторых параллельных алгоритмов

N пп	Наименование алгорита	Время вычислений	Число процессоров
	Алгебра		
1	Решение треугольной системы уравнений, обращение треугольной матрицы	$0(\log^2 n)$	
2	Вычисление коэффициентов характеристического уравнения матрицы	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
3	Решение системы линейных уравнений, обращение матрицы	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
4	Метод исключения Гаусса	$0(\log^2 n)$	$0(n^{\omega+1})$
5	Вычисление ранга матрицы	$0(\log^2 n)$	полиномиальное
6	Подобие двух матриц	$0(\log^2 n)$	
7	Нахождение *LU*-разложения симметричной матрицы	$0(\log^3 n)$	$0(n^4 / \log^2 n)$
	Комбинаторика		
1	ε —оптимальный рюкзак, n — размерность задачи	$0(\log n \log (n/e))$	$0(n^3 / \varepsilon^2)$
2	Задача о покрытии с гарантированной оценкой отклонения не более, чем в (1+ε)log d раз	$0(\log^2 n \log m)$	$0(n)$
3	Нахождение ε — хорошей раскраски в задаче о балансировке множеств	$0(\log^3 n)$	полиномиальное
	Теория графов		
1	Ранжирование списка	$0(\log n)$	$0(n/\log n)$
2	Эйлеров путь в дереве	$0(\log n)$	$0(n/\log n)$
3	Отыскание дерева минимального веса	$0(\log^2 n)$	
	Транзитивное замыкание	$0(\log^2 n)$	
5	Раскраска вершины в Δ + 1 и Δ цветов	$0(\log^3 n \log\log n)$	$0(n+m)$
6	Дерево поиска в глубину для графа	$0(\log^3 n)$	$0(n)$
	Сортировка и поиск		
1	Сортировка	$0(\log n)$	$0(n)$
2	Слияние для двух массивов размера n и m, $N = m+m$	$0(\frac{N}{P} + \log N)$	$P = 0(N / \log N)$

Скрытый параллелизм. Необходимое условие параллельного выполнения i-й и j-й итераций цикла записывается как и в случае арифметических выражение в виде правила Рассела [10]для циклов:

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

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)

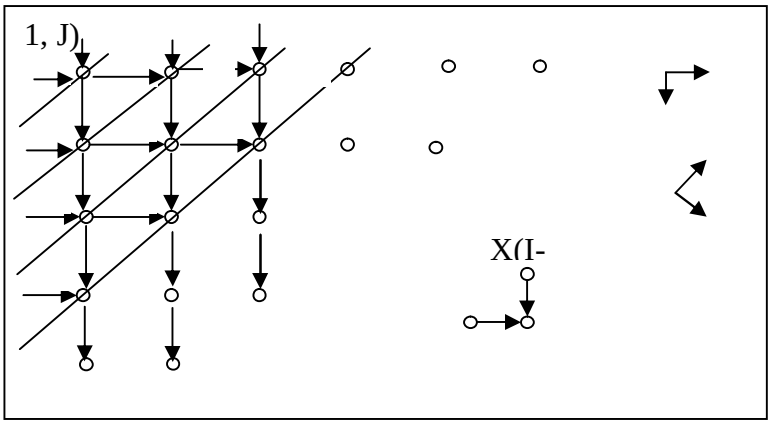
в) Итерация i s =

Итерация i + 1 s =

Знание оценок из таблицы не дает еще возможности построить практический параллельный алгоритм. Во многих случаях он не очевиден и его еще нужно различными приемами проявить в форме, доступной для программирования на некотором параллельном языке. Пример такого проявления приводится ниже.

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

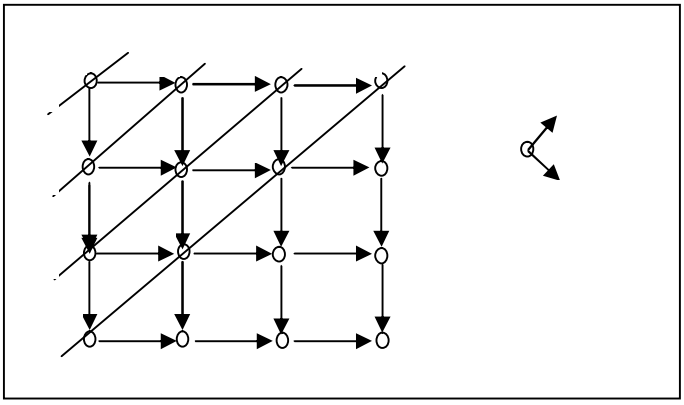
```
DO 1  I = 1,N
DO 1  J =  1,N
1   X (I, J) = X (I-1, J) + Y (I, J –1) + C
```



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)

Рисунок показывает, что между итерациями существует прямая зависимость. 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, 42, 43, 44 можно выполнять парал лельно.



Если повернуть оси I, J на 45 градусов и переименовать операции внутри каждой диагонали,как на следующем рисунке, то можно использовать этот параллелизм. Соответствующая программа для верхних диагоналей (включая главную) и нижних диагоналей приведена ниже:

```
  DO 1 I = 1,N                    Пусть I =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
```

Здесь K = 1,2 обозначает левый - верхний или правый - нижний треугольник пространства итераций

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

Недостаток метода гиперплоскостей состоит в том, что ширина параллелизма в каждой итерации параллельной программы неодинакова. Это исключено в методе параллелепипедов и ряде других методов. Достаточно полное описание методов практической разработки параллельных алгоритмов для начального изучения представлено в известной книге Ортеги [3].

Для параллельного программирования существует ряд языков. Основные из них:

- OpenMP – для многопроцессорных систем с общей памятью
- MPI - для многопроцессорных систем с индивидуальной памятью
- CUDA- для неграфических вычислений на графических процессорах

Существует часто некоторая путаница между OpenMP и MPI. Эта путаница вполне понятно, поскольку есть еще версия MPI называется "Open MPI".

Ниже кратко описываются все эти три системы программирования. В приложении 3 приводится информация по компиляции и запуску многопоточных программ, которая позволяет на персональном компьютере реально познакомится с параллельным программированием.

4.2. 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 [8] 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. Принципиально 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 никакие MPI – операторы больше выполняться не будут. Перемсенная COM_WORLD определяет перечень процессов, назначенных для выполнения программы.

MPI программа для вычисления числа π на языке C.

Для первой параллельной программы удобна программа вычисления числа π, поскольку в ней нет загрузки данных и легко проверить ответ. Вычисления сводятся к вычислению интеграла по следующей 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 $x_i = (i\text{-}1/2) / n$. Программа представлена рис.4.1.

```
#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 = 3.14159265358979238462643;  /* 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. 4.1 Программа вычисления числа p **на языке C**

Программа умножения матрицы на вектор

Результатом умножения матрицы на вектор является вектор результата. 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.4.2), код главного процесса (рис.4.3) и код подчиненного процесса (рис.4.4).

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.4. 2. Программа умножения матрицы на вектор: общая часть

Код главного процесса представлен на рис. 4.3. 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.4.3. 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.4.4.

```
!     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.4.4. 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.

Примеры задач на MPI представлены в [6,8,11]. Там же приводится полное описание библиотеки MPI, методики использования MPI в среде языков C, C++ и Fortran. Описаны методы построения кластеров на ПЭВМ.

4.3. OpenMP

Интерфейс OpenMP [13] задуман как стандарт для программирования на масштабируемых SMP-системах (модель общей памяти). В стандарт OpenMP входят спецификации набора директив компилятора, процедур и переменных среды. До появления OpenMP не было подходящего стандарта для эффективного программирования на SMP-системах.

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

- недостаточно эффективна на SMP-системах;
- относительно сложна в освоении, так как требует мышления в "невычислительных" терминах. POSIX-интерфейс для организации нитей (**Pthreads**) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического параллельного программирования: слишком низкий уровень, нет поддержки параллелизма по данным.

OpenMP можно рассматривать как высокоуровневую надстройку над Pthreads (или аналогичными библиотеками нитей). За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто последовательно добавляет в текст последовательной программы OpenMP-директивы. При этом, OpenMP - достаточно гибкий механизм, предоставляющий разработчику большие возможности контроля над поведением параллельного приложения. 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.

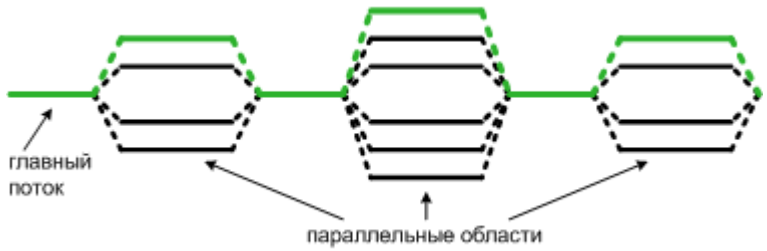
Спецификация OpenMP для C/C++, содержит следующую функциональность:

- Директивы OpenMP начинаются с комбинации символов "**#pragma omp**". Директивы можно разделить на 3 категории: определение парал-

лельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных.

- Компилятор с поддержкой OpenMP определяет макрос "**_OPENMP**", который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы.
- Распараллеливание применяется к for-циклам, для этого используется директива "**#pragma omp for**". В параллельных циклах запрещается использовать оператор break.
- Статические (static) переменные, определенные в параллельной области программы, являются общими (shared).
- Память, выделенная с помощью malloc(), является общей (однако указатель на нее может быть как общим, так и приватным).
- Типы и функции OpenMP определены во включаемом файле <**omp.h**>.
- Кроме обычных, возможны также "вложенные" (nested) мьютексы - вместо логических переменных используются целые числа, и нить, уже захватившая мьютекс, при повторном захвате может увеличить это число.

Программная модель OpenMP представляет собой fork-join параллелизм, в котором главный поток по необходимости порождает группы потоков, при вхождении программы в параллельные области приложения.



В случае симметричного мультипроцессинга SMP на всех процессорах процессорной системы исполняется один экземпляр операционной системы, которая отвечает за распределение прикладных процессов (задач, потоков) между отдельными процессорами.

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 :

$$\int_0^1 \frac{4}{1+x^2} dx = \frac{1}{n}\sum_{i=1}^{n} \frac{4}{1+x_i^{\,2}}, \quad \text{где: } x_i = \frac{1}{n}\cdot\left(i-\frac{1}{2}\right)$$

В последовательную программу вставлены две строки (директивы), и она становится параллельной (рис.4.5)..

```
#include <stdio.h>
double f(double y) {return(4.0/(1.0+y*y));}
int main()
{
double w, x, sum, pi;
int i;
int n = 1000000;
w = 1.0/n;
sum = 0.0;
#pragma omp parallel for private(x) shared(w)\
reduction(+:sum)
for(i=0; i < n; i++)
{
x = w*(i-0.5);
sum = sum + f(x);
}
pi = w*sum;
printf("pi = %f\n", pi);
}
```

Рис.4.5. Вычисление числа Пи на языке Си.

Программа начинается как единственный процесс на головном процессоре. Он исполняет все операторы вплоть до первой конструкции типа #pragma omp. В рассматриваемом примере это оператор parallel for, при исполнении которого порождается множество процессов с соответствующим каждому процессу окружением.

В случае симметричного мультипроцессинга SMP на всех процессорах процессорной системы исполняется один экземпляр операционной системы, которая отвечает за распределение прикладных процессов (задач, потоков) между отдельными процессорами. Распараллеливание применяется к for-циклам, для этого используется директива "**#pragma omp for**", по которой ОС раздает процессорам (ядрам) личный экземпляр программы, как в SPMD, попросту передает один и тотже отрезок программы на заданное число процессоров. Распараллеливание применяется к for-циклам, для этого используется директива "**#pragma omp for**", по котоой ОС раздает ппроцессорам (ядрам) личный экземпляр программы, как в SPMD.

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 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 C являются комментариями и начинаются с комбинации символов #pragma, поэтому приведенная выше программа может без изменений выполняться на последовательной ЭВМ в обычном режиме.

Распараллеливание в OpenMP выполняется явно при помощи вставки в текст программы специальных директив, а также вызова вспомогательных функций. При использовании OpenMP предполагается SPMD-модель (Single Program Multiple Data) параллельного программирования, в рамках которой для всех параллельных нитей используется один и тот же код.

Программа начинается с последовательной области – сначала работает один процесс (нить), при входе в параллельную область порождается (компилятором) ещё некоторое число процессов, между которыми в дальнейшем распределяются части кода. По завершении параллельной области все нити, кроме одной (нити мастера), завершаются, и начинается последовательная область. В программе может быть любое количество параллельных и последовательных областей.

Кроме того, параллельные области могут быть также вложенными друг в друга. В отличие от полноценных процессов, порождение нитей является отно-

сительно быстрой операцией, поэтому частые порождения и завершения нитей не так сильно влияют на время выполнения программы.

После получения исполняемого файла необходимо запустить его на требуемом количестве процессоров. Для этого обычно нужно задать количество нитей, выполняющих параллельные области программы, определив значение переменной среды MP_NUM_THREADS. После запуска начинает работать одна нить, а внутри параллельных областей одна и та же программа будет выполняться всем набором нитей. При выходе из параллельной области производится неявная синхронизация и уничтожаются все нити, кроме породившей. Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным процессорам, однако это, как правило, находится в ведении операционной системы.

Перед запуском программы количество нитей, выполняющих параллельную область, можно задать, с помощью переменной среды MP_NUM_THREADS. Например, в Linux это можно сделать при помощи следующей команды: export OMP_NUM_THREADS=n. Функция omp_get_num_procs () возвращает количество процессоров, доступных для использования программе пользователя на момент вызова. Директивы master (master ... end master) выделяют участок кода, который будет выполнен только нитью-мастером. Остальные нити просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает.

Модель данных в OpenMP предполагает наличие как общей для всех нитей области памяти, так и локальной области памяти для каждой нити. В OpenMP переменные в параллельных областях программы разделяются на два основных класса:

- shared (общие; все нити видят одну и ту же переменную);
- private (локальные, каждая нить видит свой экземпляр переменной).

Общая переменная всегда существует лишь в одном экземпляре для всей области действия. Объявление локальной переменной вызывает порождение своего экземпляра данной переменной (того же типа и размера) для каждой нити. Изменение нитью значения своей локальной переменной никак не влияет на изменение значения этой же локальной переменной в других нитях. Если несколько переменных одновременно записывают значение общей переменной без выполнения синхронизации или если как минимум одна нить читает значение общей переменной и как минимум одна нить записывает значение этой переменной без выполнения синхронизации, то возникает ситуация так называемой «гонки данных». Для синхронизации используется оператор barrier:

```
#pragma omp barrier
```

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, ждут, пока все нити не дойдут до этой точки прграммы, после чего разблокируются и продолжают работать дальше.

Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова функций OpenMP могут быть подставлены специальные «заглушки» (stub), текст которых приведен в описании стандарта. Они гарантируют корректную работу программы в последовательном случае – нужно только перекомпилировать программу и подключить другую библиотеку.

OpenMP может использоваться совместно с другими технологиями параллельного программирования, например, с MPI. Обычно в этом случае MPI используется для распределения работы между несколькими вычислительными узлами, а OpenMP затем используется для распараллеливания на одном узле.

Простейшая программа, реализующая перемножение двух квадратных матриц, представлена на рис.4.6.

```
#include <stdio.h>
#include <omp.h>
#define N 4096
double a[N][N], b[N][N], c[N][N];
int main()
{
  int i, j, k;
  double t1, t2;
// инициализация матриц
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j]=b[i][j]=i*j;
  t1=omp_get_wtime();
// основной вычислительный блок
#pragma omp parallel for shared(a, b, c) private(i, j, k)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
      c[i][j] = 0.0;
      for(k=0; k<N; k++) c[i][j]+=a[i][k]*b[k][j];
    }
  }
  t2=omp_get_wtime();
  printf("Time=%lf\n", t2-t1);
}
```

Рис.4.6. Перемножение матриц на языке Си.

В программе замеряется время на основной вычислительный блок, не включающий начальную инициализацию. В основном вычислительном блоке программы на языке Фортран изменён порядок циклов с параметрами i и j для лучшего соответствия правилам размещения элементов массивов.

4.4. Система программирования CUDA

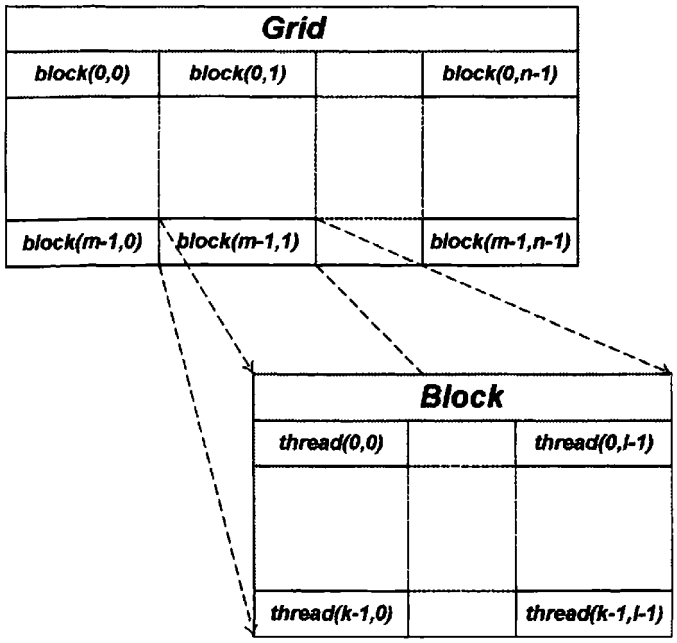
Технология multi – SIMD стала широко использоваться в графических процессорах видеокарт для выполнения объемных расчётов. Затем ее стали использовать и для нечисловых расчетов. Компания NVIDIA разработала программ-

ную реализацию этой технологии под названием CUDA (Compute Unified Device Architecture).

CUDA [14] позволяет использовать мощности графических процессоров (GPU) для вычислений общего назначения. Оказалось, что для большого класса задач производительность, которую они получают, используя GPU, значительно превосходила CPU. Поэтому было решено создать программную модель, которая позволяла бы запускать произвольный код на GPU. Эта идея и лежит в основе CUDA. Сама CUDA на рынке появилась в конце 2006 года с приходом архитектуры G80.

CUDA это C-подобный язык программирования со своим компилятором и библиотеками для вычислений на GPU. CUDA строится на концепции, что GPU (Graphics Processing Unit) выступает в роли сопроцессора к CPU (General-Purpose Unit). Программа на CUDA задействует как CPU, так и GPU. При этом обычный (последовательный, то есть непараллельный) код выполняется на CPU, а для параллельных вычислений соответствующий код выполняется на GPU как набор одновременно выполняющихся нитей (потоков, threads). Все запущенные на выполнение нити организованы в следующую иерархию.

Верхний уровень иерархии сетка (grid) соответствует всем нитям, выполняющих данное ядро. Верхний уровень представляет из себя одномерный или двухмерный массив блоков (block). Каждый блок это одномерный, двухмеый, трехмерный массив нитей (thread). При этом все блоки, образующие сетку, имеют одинаковую размерность и размер.



Каждый блок в сетке имеет свой адрес, состоящий из одного или двух неотрицательных целых чисел (индекс блока в сетке). Аналогично каждая нить внутри блока также имеет свой адрес одно, два или три неотрицательных целых числа, задающих индекс нити внутри блока.

Блоки потоков выполняются в виде небольших групп, называемых warp, размер которых - 32 потока. Это минимальный объём данных, которые могут обрабатываться в мультипроцессорах. И так как это не всегда удобно, CUDA позволяет работать и с блоками, содержащими от 64 до 512 потоков.

Группировка блоков в сетки позволяет уйти от ограничений и применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у GPU недостаточно ресурсов, он будет выполнять блоки последовательно. В обратном случае, блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеочипах разного уровня, начиная от мобильных и интегрированных.

Следует уточнить некоторые понятия в применении к GPU:

- Поток (stream) представляет собой поток элементов одного типа. В классическом программировании есть такой аналог, как массив.
- Ядро (kernel) - функция, которая будет применяться независимо к каждому элементу потока. В классическом программировании можно привести аналогию цикла - он применяется к большому числу элементов. **Ядро в CUDA не связано с понятием ядра в многоядерных процессорах.**

Ядро может получить размеры сетки и блока через встроенные переменные gridDim и blockDim.

Поскольку одно и то же ядро выполняется одновременно очень большим числом нитей, то для того, чтобы ядро могло однозначно определить номер нити (а значит, и элемент данных, который нужно обрабатывать), используются встроенные переменные threadIdx и blockIdx. Каждая из этих переменных является трехмерным целочисленным вектором. Обратите внимание, что они доступны только для функций, выполняемых на GPU, для функций, выполняющихся на CPU, они не имеют смысла.

Подобное разделение всех нитей является еще одним общим приемом использования CUDA: исходная задача разбивается на набор отдельных подзадач, решаемых независимо друг от друга. Каждой такой подзадаче соответствует свой блок нитей. При этом каждая подзадача совместно решается всеми нитями своего блока.

Разбиение нитей на warp'ы происходит отдельно для каждого блока. Таким образом, все нити одного warp'а всегда принадлежат одному блоку. При этом нити могут взаимодействовать между собой только в пределах блока. **Нити разных блоков взаимодействовать между собой не могут.**

Одним из серьезных отличий между GPU и CPU являются организация памяти и работа с ней. Обычно большую часть CPU занимают кеши различных уровней. Основная же часть GPU отведена на вычисления. Как следствие, в отличие от CPU, где есть всего один тип памяти с несколькими уровнями кеширования в самом CPU, GPU обладает более сложной структурой памяти.

Чисто физически память GPU можно разделить на DRAM (на плате видеокарты) и на память, размещенную непосредственно на GPU (точнее, в потоковых мультипроцессорах). Однако классификация памяти в CUDA ограничива-

ется ее чисто физическим расположением. В таблице приводятся доступные виды памяти в CUDA и их основные характеристики.

Тип памяти	Расположение	Кеши-руется	Доступ	Уровень доступа	Время жизни
Регистры	Мультипроцессор	Нет	R/w	Per-thread	Нить
Локальная	DRAM	Нет	R/w	Per-thread	Нить
Разделяемая	Мультипроцессор	Нет	R/w	Все нити блока	Блок
Глобальная	DRAM	Нет	R/w	Все нити и CPU	Выделяется CPU
Константная	DRAM	Да	R/o	Все нити и CPU	Выделяется CPU
Текстурная	DRAM	Да	R/o	Все нити и CPU	Выделяется CPU

Глобальная память — самый большой объём памяти, доступный для всех мультипроцессоров на видеочипе, размер составляет от 256 мегабайт до 1.5 гигабайт на текущих решениях (и до 4 Гбайт на Tesla). Обладает высокой пропускной способностью, более 100 гигабайт/с для топовых решений NVIDIA, но очень большими задержками в несколько сот тактов. Не кэшируется, поддерживает обобщённые инструкции load и store, и обычные указатели на память.

Регистровая память - наиболее простым видом памяти. Каждый потоковый мультипроцессор содержит 8192 или 16 384 32-битовых регистров (для обозначения всех регистров потокового мультипроцессора используется термин register file). Имеющиеся регистры распределяются между нитями блока на этапе компиляции (и, соответственно, влияют на количество блоков, которые может выполнять один мультипроцессор).

Каждая нить получает в свое монопольное пользование некоторое количество регистров, которые доступны как на чтение, так и на запись (readjwrite). Нить не имеет доступа к регистрам других нитей, но свои регистры доступны ей на протяжении выполнения данного ядра. Поскольку регистры расположены непосредственно в потоковом мультипроцессоре, то они обладают максимальной скоростью доступа.

Если имеющихся регистров не хватает, то для размещения локальных данных (переменных) нити используется так называемая локальная память, размещенная в DRAM. Поэтому доступ к локальной памяти характеризуется очень высокой латентностью - от 400 до 600 тактов.

Локальная память — это небольшой объём памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная — такая же, как и глобальная.

Разделяемая память — это 16-килобайтный блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре. Эта память быстрая, как регистры. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую. Преимущества разделяемой памяти: использование в виде кэша первого уровня, снижение задержек при доступе исполнительных блоков (ALU) к данным, сокращение количества обращений к глобальной памяти.

В графических API **полностью отсутствует возможность какого - либо взаимодействия между параллельно обрабатываемыми потоковыми процессорами**, что в графике действительно не нужно, но для вычислительных задач оказывается довольно желательным. Такой обмен осуществляется через память, чаще всего – через локальную.

Память констант — область памяти объемом 64 килобайта (то же — для нынешних GPU), доступная только для чтения всеми мультипроцессорами. Она кэшируется по 8 килобайт на каждый мультипроцессор. Довольно медленная - задержка в несколько сот тактов при отсутствии нужных данных в кэше.

Текстурная память — блок памяти, доступный для чтения всеми мультипроцессорами. Выборка данных осуществляется при помощи текстурных блоков видеочипа, поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кэшируется по 8 килобайт на каждый мультипроцессор. Медленная, как глобальная — сотни тактов задержки при отсутствии данных в кэше.

Естественно, что глобальная, локальная, текстурная и память констант - это физически одна и та же память, известная как локальная видеопамять видекарты. Их отличия в различных алгоритмах кэширования и моделях доступа. Центральный процессор может обновлять и запрашивать только внешнюю память: глобальную, константную и текстурную.

Программы для CUDA (соответствующие файлы имеют расширение .си) пишутся на «расширенном» С и компилируются при помощи команды nvcc.

Вводимые в CUDA расширения языка С состоят из:

- спецификаторов функций, показывающих, где будет выполняться функция и откуда она может быть вызвана;
- спецификаторов переменных, задающих тип памяти, используемый для данных переменных;
- директивы для запуска ядра, задающей как данные, так и иерархию нитей;
- встроенных переменных, содержащих информацию о текущей нити;
- runtite, включающей в себя дополнительные типы данных.

Спецификаторы функций :

Спецификатор	Функция выполняется на	Функция может вызываться из
__device__	device (GPU)	device (GPU)
__global__	device (GPU)	host (CPU)
__host__	host (CPU)	host (CPU)

Спецификаторы *host* и *device* могут быть использованы вместе (это значит, что соответствующая функция может выполняться как на GPU, так и на CPU код для обеих платформ будет автоматически crene рирован компилятором). Спецификаторы *global* и *host* не могут быть использованы вместе.

Спецификатор global обозначает ядро, и соответствующая функция должна возвращать значение типа void.

На функции, выполняемые на GPU (device и global), накладываются следующие ограничения:

- нельзя брать их адрес (за исключением global функций);
- не поддерживается рекурсия;
- не поддерживаются static переменные внутри функции;
- не поддерживается переменное число входных аргументов.

Для задания размещения в памяти GPU переменных используются следующие спецификаторы device, constant и shared. На их использовани е также накладывается ряд ограничений:

- эти спецификаторы не могут быть применены к полям структуры (struct или union);
- соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern;
- запись в переменные типа constant может осуществляться только CPU при помощи специальных функций;
- shared переменые не могут инициализироваться при объявлении.

Добавленные переменные:

В язык добавлены следующие специальные переменные:

- gridDim размер сетки (имеет тип dimз);
- blockDim размер блока (имеет тип dim3);
- blockldx индекс текущего блока в сетке (имеет тип uint3);
- threadldx индекс текущей нити в блоке (имеет тип uint3);
- warpSize размер warp'a (имеет тип int).

Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

```
kernelName <<<Dg,Db,Ns,S>>> ( args );
```

Здесь *kernelName* это имя (адрес) соответствующей global функции. Через Dg обозначена переменная (или значение) типа dim 3, задающая размерность размер сетки (в блоках). Переменная (или значение) Db типа dim3, задает размерность и размер блока (в нитях).

Необязательная переменная (или значение) *Ns* типа size_t задает дополнительный объем разделяемой памяти в байтах, которая должна быть динамически выделена каждому блоку (к уже статически выделенной разделяемой памяти), если не задано, то используется значение 0.

Переменная (или значение) S типа cudaStream_t задает поток (*CUDA stream*), в котором должен произойти вызов, по умолчанию используется поток 0.
Через *args* обозначены аргументы вызова функции ketelNate (их может быть несколько).

Следующий пример запускает ядро с именем myKernel параллельно на *n* нитях, используя одномерный массив из двумерных (16x16) блоков нитей, и передает на вход ядру два параметра *a* и *n*. При этом каждому блоку дополнительно

выделяет ся 512 байт разделяемой памяти и запуск, производится на потоке myStream.

```
myKernel<<<dim3(n/256),dim3(16,16),512,myStream>>> ( a, n );
```

Рассмотрим простой пример сложения двух векторов на классическом и графическом процессорах:

Standart C Code

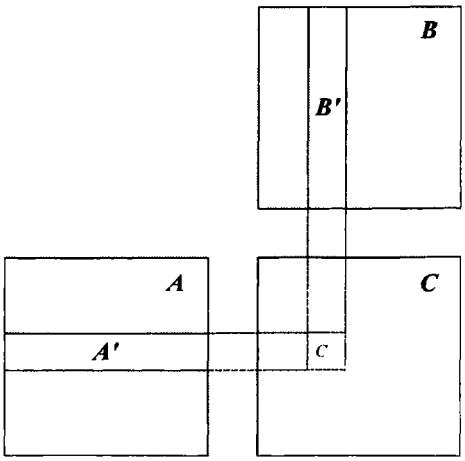
```
void saxpy_serial (int n,  float a,  float *x,  float *y )
{
    for (int i  =  0;  i  <  n;  ++i )
   y[i]  = a*x[i]  + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial (n,  2.0,  x,  y );
```

Parallel C Code

```
_global_ void saxpy parallel(int n,  float a,  float *x,  float*y)
{
    int i  =  blockIdx.x  *  blockDim.x  +  threadIdx.x;
    int (i < n )  y[i]  =  a*x[i]  +  y[i];
}
// Invoke parallel SAXPY kernel with 256 thread/block
int nblocks  =  ( n*255)/256
SAXPY_parallel <<<nblocks, 256>>>(n, 2.0,  x,  y);
```

Теперь рассмотрим более сложный пример - использование разделяемой памяти при умножения двух квадратных матриц. Будем использовать двухмерные блоки размера 16 x 16 и будем считать, что размер матриц N кратен 16. Каждый блок будет вычислять одну 16x16 подматрицу C' искомого произведения.

Как видно по рисунку, для вычисления подматрицы C' произведения A x B нам приходится постоянно обращаться к двух полосам (подматрицам) исходных матриц A’ и B’ . Обе эти полосы имеют размер Nx16, и их элементы многократно используются в расчетах.



Идеальным вариантом было разместить копии этих полос в разделяемой памяти, однако для реальных задач это неприемлемо из-за небольшого объема имеющейся разделяемой памяти (так, если N равно 1024, то одна полоса будет занимать в памяти 1024 x 16 x 4 == 64 Кб). Однако, если каждую из этих полос разбить на квадратные подматрицы 16x16, то становится видно, что результирующая матрица C просто является суммой попарных произведений подматриц из этих двух полос:

$$C' == A1' \times B1' + A2' \times B2' + ... + AN'/16 \times BN'/16.$$

За счет этого можно выполнить вычисление подматрицы C' Всего за N/16 шагов. На каждом таком шаге в разделяемую память загружаются одна 16 x 16 подматрица A и одна подматрица B (при этом нам потребуется 16 x 16 x 4 x 2 = 2 Кбайта разделяемой памяти на блок), при этом каждая нить блока загружает ровно по одному элементу из каждой из этих подматриц, то есть каждая нить делает всего два обращения к глобальной памяти на один шаг. После этого считается произведение подматриц, загруженных в разделяему память, и суммируется нужный элемент произведения, и идет переход к следующей паре подматриц.

После загрузки необходимо поставить синхронизацию, чтобы убедиться, что обе подматрицы загружены полностью (а не только 32 элемента, загруженных данным warp'ом). Точно так же синхронизацию необходимо поставить и после вычисления произведения загруженных подматриц до загрузки следующей пары (чтобы убедиться, что текущие подматрицы уже не нужны никакой нити).

Таким образом, при вычислении произведения матриц нам на каждый элемент произведения C нужно выполнить всего 2xN/16 чтений из глобальной памяти, в отличие от варианта без использования глобальной памяти, где нам требовалось на каждый элемент 2xN чтений. Количество арифметических операций не изменилось и осталось равным 2xN - 1.

Программа представлена ниже (взята из [5]).

```
#define BLOCK_SIZE  16  // Размер блока.

__global__ void matMult ( float * a, float * b, int n, float * c )
{
  int bx = blockIdx.x;
  int by = blockIdx.y;

  int tx = threadIdx.x;
  int ty = threadIdx.y;

                        // Индекс начала первой подматрицы A, обрабатываемой блоком.
  int aBegin = n * BLOCK_SIZE * by;
  int aEnd   = aBegin + n - 1;

                        // Шаг перебора подматриц A.
  int aStep = BLOCK_SIZE;

                        // Индекс первой подматрицы B обрабатываемой блоком.
  int bBegin = BLOCK_SIZE * bx;

                        // Шаг перебора подматриц B.
  int bStep = BLOCK_SIZE * n;

  float sum = 0.0f;     // Вычисляемый элемент C'.

                        // Цикл по 16*16 подматрицам
  for ( int ia = aBegin, ib = bBegin; ia &lt;= aEnd; ia += aStep, ib += bStep )
  {
                        // Очередная подматрица A в разделяемой памяти.
    __shared__ float as [BLOCK_SIZE][BLOCK_SIZE];

                        // Очередная подматрица B в разделяемой памяти.
    __shared__ float bs [BLOCK_SIZE][BLOCK_SIZE];

                        // Загрузить по одному элементу из A и B в разделяемую память.
    as [ty][tx] = a [ia + n * ty + tx];
    bs [ty][tx] = b [ib + n * ty + tx];

                        // Дождаться, когда обе подматрицы будут полностью загружены.
    __syncthreads();

                        // Вычисляем нужный элемент произведения загруженных подматриц.
    for ( int k = 0; k &lt; BLOCK_SIZE; k++ )
      sum += as [ty][k] * bs [k][tx];

                        // Дождаться, пока все остальные нити блока закончат вычислять
                        // свои элементы.
    __syncthreads();
  }

                        // Записать результат.
  int ic = n * BLOCK_SIZE * by + BLOCK_SIZE * bx;

  c [ic + n * ty + tx] = sum;
}
```

Основной процесс приложения CUDA работает на универсальном процессоре (host), он запускает несколько копий процессов kernel на видеокарте. Код для CPU делает следующее:

- инициализирует GPU,
- распределяет память на видеокарте и системе,
- копирует константы в память видеокарты,
- запускает несколько копий процессов kernel на видеокарте,
- копирует полученный результат из видеопамяти, освобождает память, завершает работу.

Из написанного выше понятно, что CUDA предполагает специальный подход к разработке, не совсем такой, как принят в программах для CPU. Нужно помнить о разных типах памяти, о том, что локальная и глобальная память не кэшируется и задержки при доступе к ней гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах, то есть важно найти оптимальное место для хранения данных, минимизировать передачу данных между CPU и GPU, использовать буферизацию.

На практике, принципиальная схема создания CUDA-программы такова: поставляемый NVIDIA компилятор встраивается в среду разработки, он компилирует исходный файл с кодом функции, которая должна исполняться на устройстве и превращает его в ассемблерный код для CUDA-устройства, который присоединяется к программе, как ресурс данных. При запуске программы на конкретной системе, этот код, с помощью библиотечной функции, передается видеодрайверу, который компилирует его для имеющегося CUDA-устройства в машинный код, специфичный для данного устройства. Так обеспечивается совместимость между редакциями архитектур. И далее, с помощью вызова библиотечных функций из кода основной программы, данные загружаются в GPU и для исполнения на устройстве вызывается скомпилированная драйвером функция.

Р А З Д Е Л 2. **РЕАЛИЗАЦИЯ**

В этом разделе рассмотрены реализации, которые позволяют увеличить количество одновременно работающих АЛУ, процессоров, что увеличивает быстродействие ЭВМ.

Глава 5. ВЫЧИСЛИТЕЛЬНЫЕ КЛАСТЕРЫ.

5.1. Вычислительные кластеры.

Кластер. Магистральным направлением развития параллельных ЭВМ для крупноформатного параллелизма является построение таких систем на базе средств массового выпуска: микропроцессоров, каналов обмена данными, системного программного обеспечения, языков программирования, конструктивов.

Системы с индивидуальной памятью идеально подходят подходят для реализации на основе электронной и программной продукции массового производства. При этом не требуется разработки никакой аппаратуры. Такие системы получили название *кластеры рабочих станций* или просто «кластеры». Кластеры также используются и в системах для распределенных вычислений Грид.

В общем случае, вычислительный кластер - это набор компьютеров (вычислительных узлов), объединенных некоторой коммуникационной сетью. Каждый вычислительный узел имеет свою оперативную память и работает под управлением своей операционной системы. Наиболее распространенным является использование однородных кластеров, то есть таких, где все узлы абсолютно одинаковы по своей архитектуре и производительности.

Первым в мире кластеромявляется кластер Beowulf, созданный в научно-космическом центре NASA – Goddard Space Flight Center летом 1994 года. Названный в честь героя скандинавской саги, кластер состоял из 16 компьютеров. Особенностью такого кластера является масштабируемость, то есть возможность увеличения количества узлов системы с пропорциональным увеличением производительности. Узлами в кластере могут служить любые серийно выпускаемые автономные компьютеры, количество которых может быть от 2 до 1024 и более.

В СНГ развитие кластеров получило развитие в рамках программы Союзного государства России и Беларуси «СКИФ», начатой в 2000 году. В настоящее время построены кластеры с быстродействием в десятки терафлопс.

Последующие годы вместили в себя гигантский скачок в развитии аппаратной базы, накопление идей и методов параллельного программирования, что сделало появление кластеров было неизбежным. В настоящее время в списке Top 500 самых высокопроизводительных систем именно кластеры составляют большую часть списка.

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

- Кластеры типа Beowulf, которые строятся на базе обычной локальной сети ПЭВМ. Используются в вузах для учебной работы и небольших организациях для выполнениях проектов.
- Монолитные кластеры, все оборудование которых компактно размещено в специализированных стойках массового производства. Это очень быстрые машины, число процессоров в которых может достигать сотен и тысяч. Процессоры в монолитных кластерах не могут использоваться в персональном режиме.

Однако, в обоих случаях, кластер строится на продукции массового производства, в частности, на локальных сетях. Основное отличие кластеров от локальных сетей заключается в системном программном обеспечении (СПО). Таким пакетом в частности является широко распространенная библиотеке функций обмена для кластеров MPI (Message Passsing Interfase), описанную выше.

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.

Организация кластера. Пример структуры кластера на базе локальной сети представлен на рис. 5.1. Кластерная система состоит из:

- стандартных вычислительных узлов (процессоры);
- высокоскоростной сети передачи данных SCI;
- управляющей сети Fast Ethernet/Gigabyte Ethernet;
- управляющей ПЭВМ.
- сетевой операционной системы LINUX или WINDOWS;
- специализированныз программных средсв для поддержки обменов данными (MPICH).

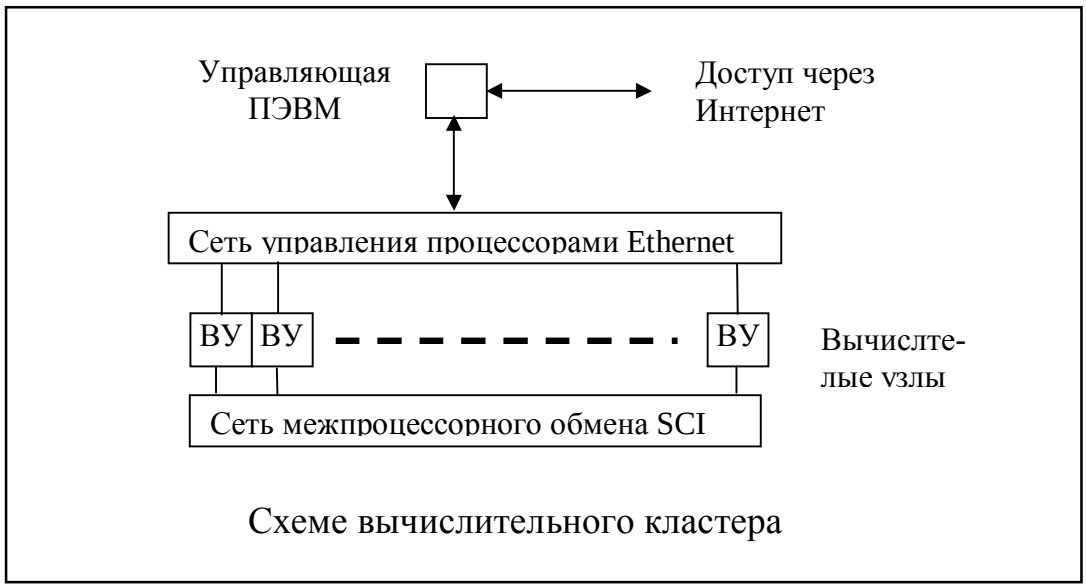


Рис.5.1. Сзема вычислительного кластера.

Управляющая ПЭВМ кластера является администратором кластера, то есть определяет конфигурацию кластера, его секционирование, подключение и отключение узлов, контроль работоспособности, обеспечивает прием заданий на выполнение вычислительных работ и контроль процесса их выполнения, планирует выполнение заданий на кластере. В качестве планировщиков обычно используются широко распрастраненные пакеты PBS, Condor, Maui и др.

Для организации взаимодействия вычислительных узлов суперкомпьютера в его составе используются различные сетевые (аппаратные и программные) средства, в совокупности образующие две системы передачи данных:

Сеть межпроцессорного обмена объединяет узлы кластерного уровня в кластер. Эта сеть поддерживает масштабируемость кластерного уровня суперкомпьютера, а также пересылку и когерентность данных во всех вычислительных узлах кластерного уровня суперкомпьютера в соответствии с программой на языке MPI.

Сеть управления предназначена для управления системой, подключения рабочих мест пользователей, интеграции суперкомпьютера в локальную сеть предприятия и/или в глобальные сети.

В качестве вычислительных узлов обычно используются однопроцессорные компьютеры, двух- или четырехпроцессорные SMP-серверы или их многоядерные реализации.

Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаще всего используются стандартные операционные системы: Linux, Windows и др. Состав и мощность узлов может меняться даже в рамках одного кластера, давая возможность создавать неоднородные системы.

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

MPI [8] 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.

5.2. Коммуникационные системы вычислительных кластеров

Коммуникационные сети [12]. Соединительная сеть, реализующая одновременно (за один такт) все виды парных и коллективных соединений, изображается в виде двудольного графа (рис. 5.2,а). Она представляет собой решетку, на пересечениях которой осуществляются необходимые замыкания (рис. 5.2,б). Такая решетка называется координатным переключателем и имеет существенный недостаток: объем оборудования в ней пропорционален $N \times M$, поэтому ко-

ординатные переключатели используются для ПП размером не более 16...32 процессора.

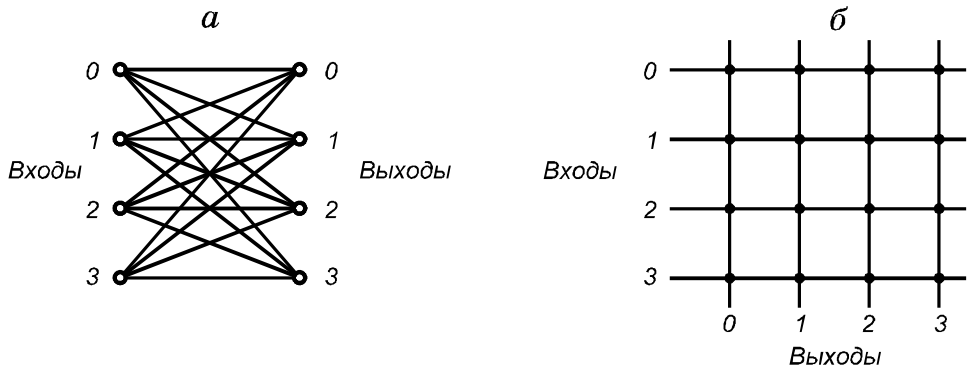


Рис. 5.2. Два представления координатного переключателя размером 4 ´ 4:

***а* — в виде двудольного графа; *б* — в виде решетки связей**

Коммутатором, противоположным по своим свойствам полному координатному соединителю, является общая шина UNIBUS, в которой осуществлятся только один обмен в единицу времени. Между этими крайними случаями имеется множество сетей, отличающихся назначением, быстродействием и стоимостью. Рассмотрим некоторые реальные системы обмена.

Шинные технологии – Ethernet. Самая простая форма топологии (**bus**) физической шины представляет собой один основной кабель, оконцованный с обеих сторон специальными типами разъемов – терминаторами. При создании такой сети основной кабель прокладывают последовательно от одного сетевого устройства к другому. Сами устройства подключаются к основному кабелю с использованием подводящих кабелей и Т-образных разъемов. Пример такой топологии приведен на рис.5.3.

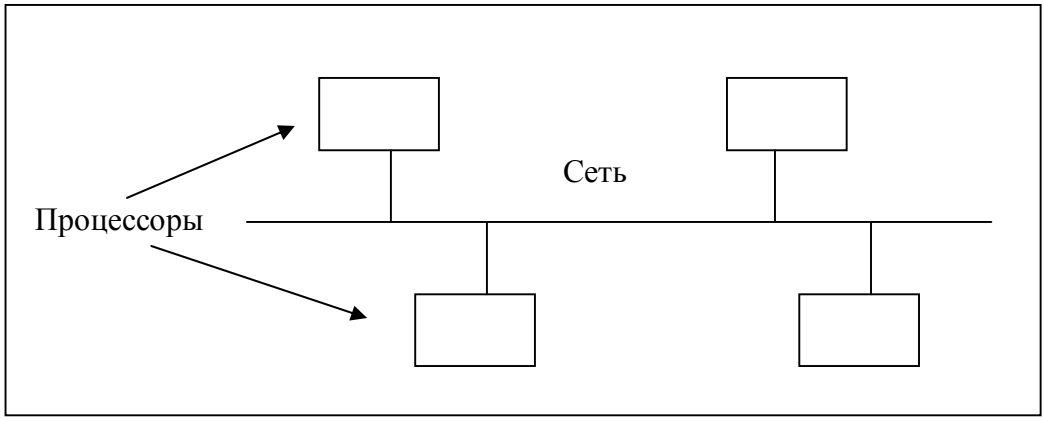


Рис. 5.3. Коммутатор с общей шиной.

Наиболее распространенной коммуникационной технологией для локальных сетей является технология Ethernet, которая имеет несколько технических реа

лизаций. 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.5.4 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.

На рис.5.4 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.

Рис.5.4. Метод случайного доступа CSMA/CD

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 Ethernet предусмотрен специальный механизм для разрешения коллизий.

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. Для исключения задержек такого рода используются промежуточные коммутаторы – свичи (switch), то есть описанные выше координатные переключатели.

Кольцевые коммутаторы. Для кластеров боьшого размера (несколько десятков или сотен узлов) Ethernet оказывается медленным, поэтому используют более быстрые технологии, например, кольцевые. Наиболее известным представителем кольцевых структур является сеть SCI (Scalable Coherent Interface). Структура сети представлена на рис.5.5.

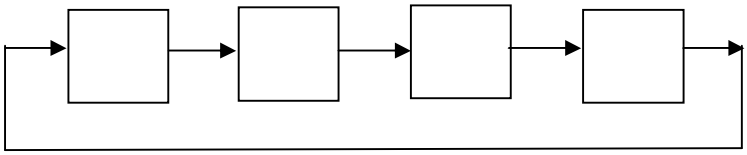


Рис.5.5. Кольцевая структура SCI

Каждый узел имеет входной и выходной каналы. Узлы связаны **однонаправленными** каналами «точка – точка». При объединении узлов должна обязательно формироваться **циклическая магистраль (кольцо)** из соединяемых узлов. Один узел в кольце, называемый «scrubber» (очиститель), выполняет функции уничтожения пакетов, не нашедших адресата. Этот узел помечает проходящие через него пакеты и уничтожает уже помеченные пакеты. В кольце может быть только один scrubber.

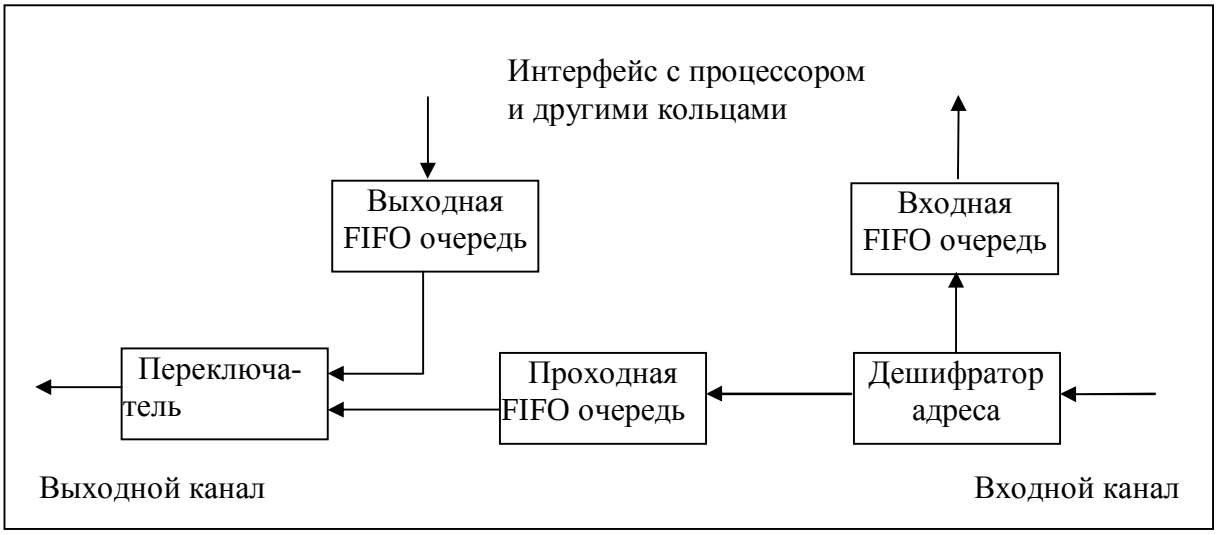


Рис.5.6. Структура узла SCI

Узлы SCI (рис.5.6) должны отсылать сформированные в них пакеты, возможно с **одновременным** приемом других пакетов, адресованных узлу, и пропуском через узел транзитных пакетов.

Для транзитных пакетов, прибывающих во время передачи узлом собственного пакета, предусмотрена проходная FIFO очередь. Размер проходной очере-

ди должен быть достаточен для приема пакетов без переполнения. В узле также вводятся входная и выходная FIFO очереди для пакетов, принимаемых и передаваемых узлом соответственно.

Узел SCI принимает поток данных и передает другой поток данных.Эти потоки состоят из SCI пакетов и свободных (пустых – iddle) символов. Через каналы непрерывно передаются либо символы пакетов, либо свободные символы, которые заполняют интервалы между пакетами.

Узел может передавать пакеты, если его проходная FIFO очередь пуста. Передача пакета инициируется его перемещением в выходную FIFO очередь. Если в течение заданного времени не поступает ответный пакет, то выполняется повтор передачи пакета.

Узел может послать много пакетов (вплоть до 64) прежде, чем будет получен ответный эхо-пакет. Эхо-пакеты могут приходить не в том порядке, в котором были посланы инициировавшие их пакеты, поэтому необходимы номера пакетов для установления соответствия между пакетами и эхо-пакетами.

Для предотвращения блокировок всем узлам предоставляется право доступа к SCI. **Это означает, что все возможные 64К устройств могут начать передачу одновременно. Это и есть масштабируемость**.

Главный недостаток сети SCI - физические ограничения на общую протяженность сети. Поэтому SCI применяется только для кластеров, поскольку они расположены на ограниченной территории. SCI не подходит для Грид.

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 представлены характеристики некоторых технологий для обмена данными.

Название технологии	Пиковая пропускная способность	Архитектура реализации	Латентность на уровне MPI	Стоимость
Ethernet	12 MB/ sec	Смешанная	50 мкс	Низкая
SCI	10 GB/ sec	Кольцо, тор	4 мкс	Высокая
InfiniBand	120 GB/ sec	Любая	5 мкс	Средняя

5.3. Метод Гаусса решения СЛАУ на кластере.

Метод решения СЛАУ с постолбцовым выбором главного элемента. Этот материал основан на статье [15]. Рассматриваются системы вида:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1, \\ &a_{21}x_{11} + a_{22}x_2 + ... + a_{2n}x_n = b_2, \\ &.. \\ &a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n, \end{aligned} \qquad (5.1)$$

или иначе, системы векторно-матричных уравнений

Ax=b.

Наиболее известным методом решения систем вида (5.1) является метод Гаусса, состоящий в последовательном исключении неизвестных. Будем поэтапно приводить систему (5.1) 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 (5.1) система:

$$\begin{aligned} a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + ... + a_{1n}x_{1n} &= b_1 \\ a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + ... + a_{2n}^{(1)}x_n &= b_2^{(1)} \\ a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 + ... + a_{3n}^{(1)}x_n &= b_3^{(1)} \qquad (5.2) \\ .... \\ a_{n2}^{(1)}x_2 + a_{n3}^{(1)}x_3 + ... + a_{nn}^{(1)}x_n &= b_n^{(1)} \end{aligned}$$

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 (5.2), получающейся исключением первого уравнения. Эквивалентный (5.2) результат второго этапа будет иметь вид:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\
a_{33}^{(2)}x_3 + \ldots + a_{3n}^{(2)}x_n &= b_3^{(2)} \qquad (5.2\text{ а}) \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_{n3}^{(2)}x_3 + \ldots + a_{nn}^{(2)}x_n &= b_n^{(2)}
\end{aligned}
$$

где

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

Продолжая этот процесс, на (n -1)–ом этапе так называемого прямого хода метода Гаусса данную систему (5.1) приведем к треугольному виду:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \qquad (5.3) \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_{nn}^{(n-1)}x_n &= b_n^{(n-1)}
\end{aligned}
$$

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

$$
a_{ij}^{(k)} = a_{ij_{ij}}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}a_{kj}^{(k-1)}, \quad b_i^{(k)} = b_i^{(k)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}b_k^{(k-1)}, \quad (5.4)
$$

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 $a_{ij}^{(0)} := a_{ij}, \quad b_i^{(0)} := b_i$.

Треугольная структура системы (5.3) 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* $k = 1, 2, \ldots, n-1, 2$
2. *Найти* $m \geq k$, *что* $|a_{mk}| = \max\{|a_{ik}|\}$, *обменять строки* m *и* k
3. *Для* $i = k+1, \ldots, n$:
4. $t_{ik} := a_{ik} / a_{kk}$,
5. $b_i := b_i - t_{ik} b_{k,}$;
6. *Для* $j = k+1, \ldots, n$: (5.4)
7. $a_{ij} := a_{ij} - t_{ik} a_{kj}$.
8. $x_n := b_n / a_{nn}$;
9. *Для* $k = n-1, \ldots, 2, 1$:
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 неизвестных системы (5.1) и вектор свободных членов, и выполнив три вложенных цикла прямого хода и один цикл вычислений обратного хода, 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 $j = k, \ldots, n$.

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:

$$\begin{array}{l} y_1 \qquad\qquad\qquad\qquad\qquad = b_1 \\ l_{21}y_1 + \quad y_2 \qquad\qquad\qquad = b_2 \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ l_{n1}y_1 + l_{n2}y_2 + \dots + l_{n,n-1}y_{n-1} + y_n = b_n \end{array}$$

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

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k \qquad (5.5)$$

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

$$\begin{array}{r} u_{11}x_1 + u_{12}x_2 + \dots\dots + u_{1n}x_n = y_1 \\ + u_{22}x_2 + \dots\dots + u_{2n}x_n = y_2 \\ \dots\dots\dots\dots\dots\dots\dots \\ u_{nn}x_n = y_n \end{array} \qquad (5.6)$$

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

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{k=i+1}^{n} u_{ik} x_k \right) \qquad (5.7)$$

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 (5.3) для МГ. Переход от МГ к КСГ рассмотрим на примере разложения в МГ (5.2): 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.5.2 а) 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 (5.3) для МГ. Переход от МГ к КСГ рассмотрим на примере разложения в МГ (5.2): 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 (5.2 а) ситуация повторяется.

Методы блочного размещения данных в кластере. Теперь будет рассмотрено размещение матрицы по слоям в машинах с распределенной памятью с целью наиболее эффективного выполнения гауссового исключения [3]. Будет обсужден порядок слоев данных, начиная с самого простого, но не эффективного варианта, и более эффективные варианты. Система обозначений показана на следующем рис.5.7.

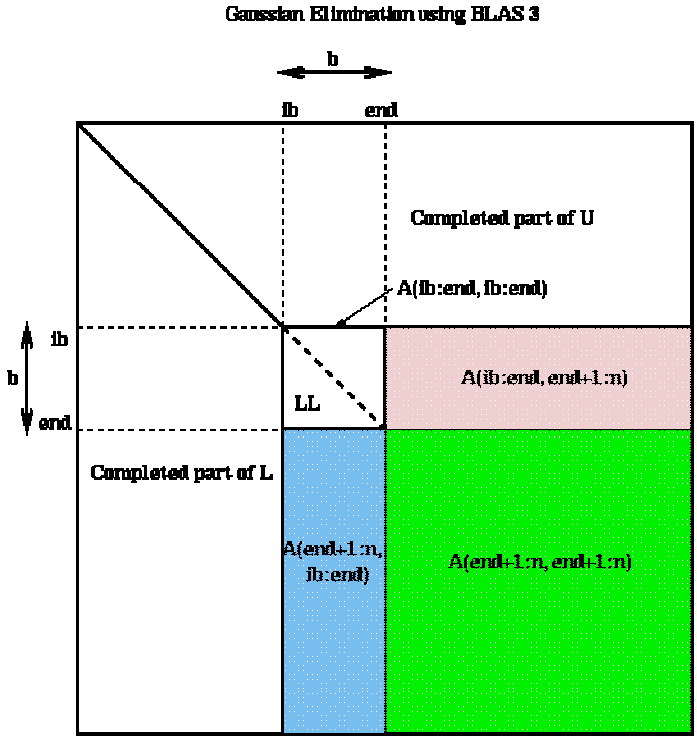


Рис. 5.7. Методы блочного размещения данных в кластере

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.

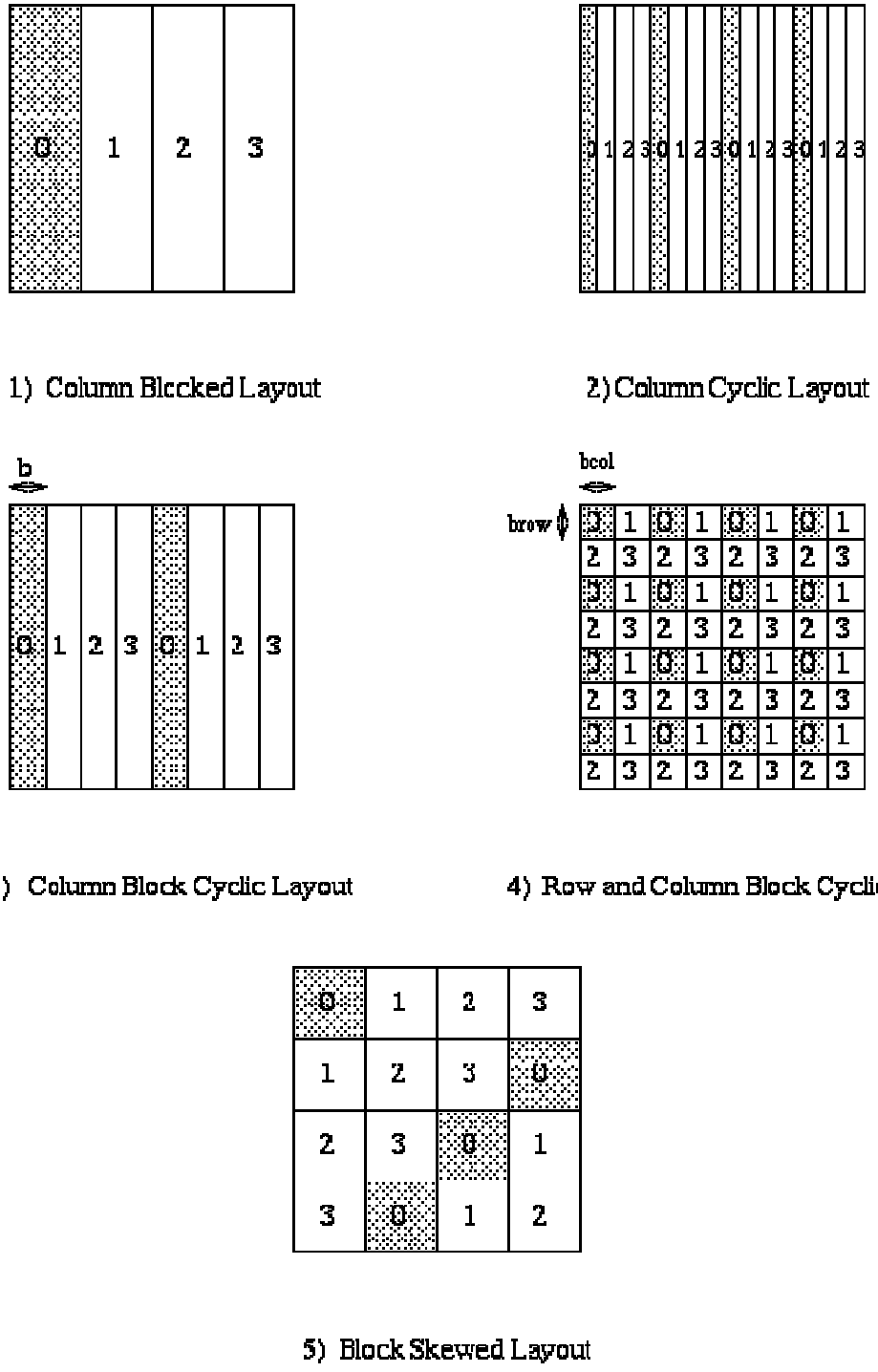


Рис.5.8. Варианты секционирования матриц

Понять эти проблемы помогает понять это рис.5.8. 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.

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 обновления 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.

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 блочно-циклического размещения:

pi = последний (i/brow) mod prow, где
brow = размер блока в направлении строк
pj = последний (j/bcol) mod pcol, где
bcol = размер блока в направлении столбцов

Поэтому, это размещение включает все предыдущие и их транспозиции как специальные случаи. На рис.5.8 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.

Есть еще одно размещение – блочное смещенное размещение (*Block Skewed 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.

Варианты LU Decomposition. Возможны три естественных варианта для LU decomposition: левосторонний поиск, правосторонний поиск и метод Краута.

- 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 последовательно ниже.

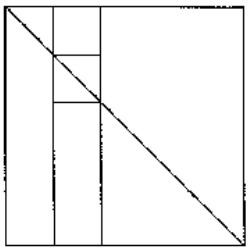


Figure 3.6 Left-Looking LU Algorithm

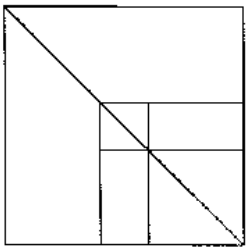


Figure 3.7 Right-Looking LU Algorithm

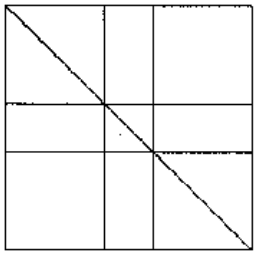


Figure 3.8 Crout LU Algorithm

Блочные методы (BLAS-3) представлены на рис. 5.10.

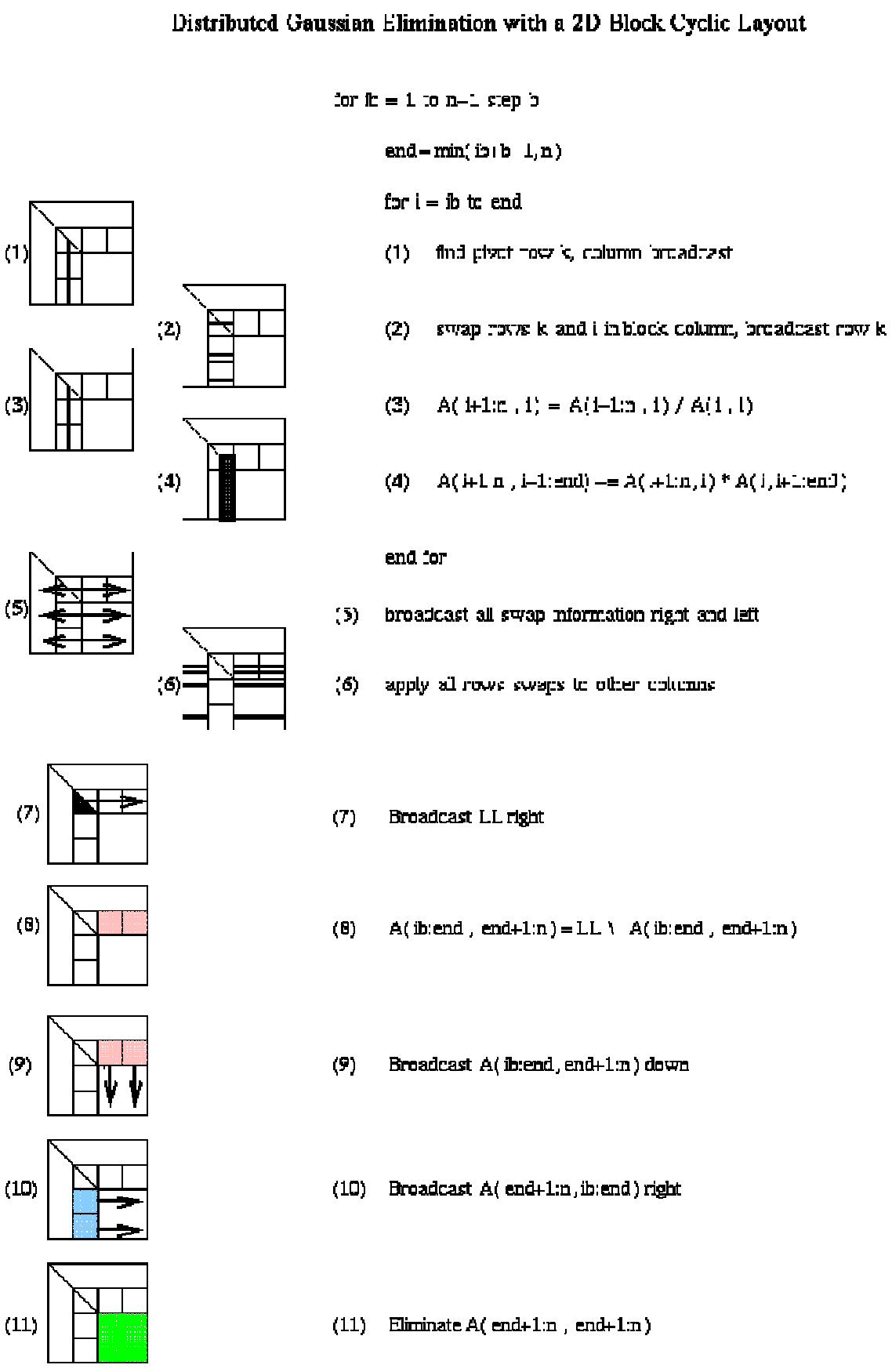


Рис.5.10. Алгоритм BLAS3 на блочно-циклическом размещении матрицы.

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 (5.4).

```
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)                    (5.8)
       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.

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.

Э**ффективность вычислений.** Для дальнейшего расчета предположим, что:

- Одна плавающая операция 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, посылаю-

щим 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 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                                          (5.9)
+ ( 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                                                (5.10)
  + (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, поэтому 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 размеру проблемы n^2 расти пропорционально числу процессоров, эффективность будет 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.

7.7. Неоднородные многоядерные процессоры

На начальном этапе создание МЯП шло по принципу: каждое ядро есть полноценный процессор. Такой процессор был многофункциональным и включал:

- АЛУ с регистрами,
- оперативную память и кэши,
- устройство выборки команд и управления ветвлениями,
- управление периферией и сетевыми устройствами.

Естественно, для создания такого процессора требовалось много транзисторов и места на кристалле, увеличивалось энергопотребление. Это ограничивало количество ядер, а значит, и быстродействие.

Есть два факта, которые составляют основу использования высокопараллельных вычислительных архитектур:

- Потребляемая мощность процессора пропорциональна примерно квадрату тактовой частоты. Например, процессор с тактовой частотой 3 ГГц потребляет больше, чем 9 процессоров частотой 1 ГГц. Значит, чтобы не перегреть кристалл МЯП, предпочтительно повышать быстродействие МЯП за счет увеличения числа ядер, а не частоты.
- Если задача уже распараллелена на большое количество потоков, то тем больше вероятность, что задача может быть распараллелена на ещё большее количество потоков. Это задачи с массовым параллелизмом. К ним относятся вычислительные задачи. Это не обязательно решение математических уравнений, это может быть обработка изображений, проверка ключей, анализ строк. Но суть в том, что надо считать, считать и считать, а не обрабатывать ветвления, когда текущая ширина параллелизма резко сужается. Но, именно для вычислительных задач, большой объем ветвлений не характерен. Это задачи с массовым параллелизмом. Это позволяет использовать МЯП с большим и очень большим количеством ядер.

В вычислительных задачах обычно используется SIMD организация, позволяющая производить параллельную обработку вектора данных. При этом возможны две многоядерные аппаратные реализации этого процесса:

- Имеется n_1 процесоров, каждый из которых обрабатывает свою часть вектора. Здесь требуется n_1 ядер.
- Имеется один мультипроцессор, в состав которого входят n_2 АЛУ, каждое обрабатывает свою часть вектора под управлением мультипроцессра. Эти АЛУ также являются ядрами.

Известно, что для построения одного АЛУ требуется в несколько раз меньше транзисторов, чем для построения процессора. Поэтому на одинаковой площади кристалла можно разместить в несколько раз больше ядер, то есть $n_2 >> n_1$.

Если на кристалле имеется несколько мультипроцессоров, этот соответствует архитектуре multi-SIMD.

Наиболее ярким примером неоднородных МЯП являются графические процессоры компаний NVIDIA и AMD, которые содержат много ядер. После появления технологии CUDA такие ядра стали называться CUDA ядрами.
Рассмотрим пример. На примерно равных по размеру кристаллах размещены:

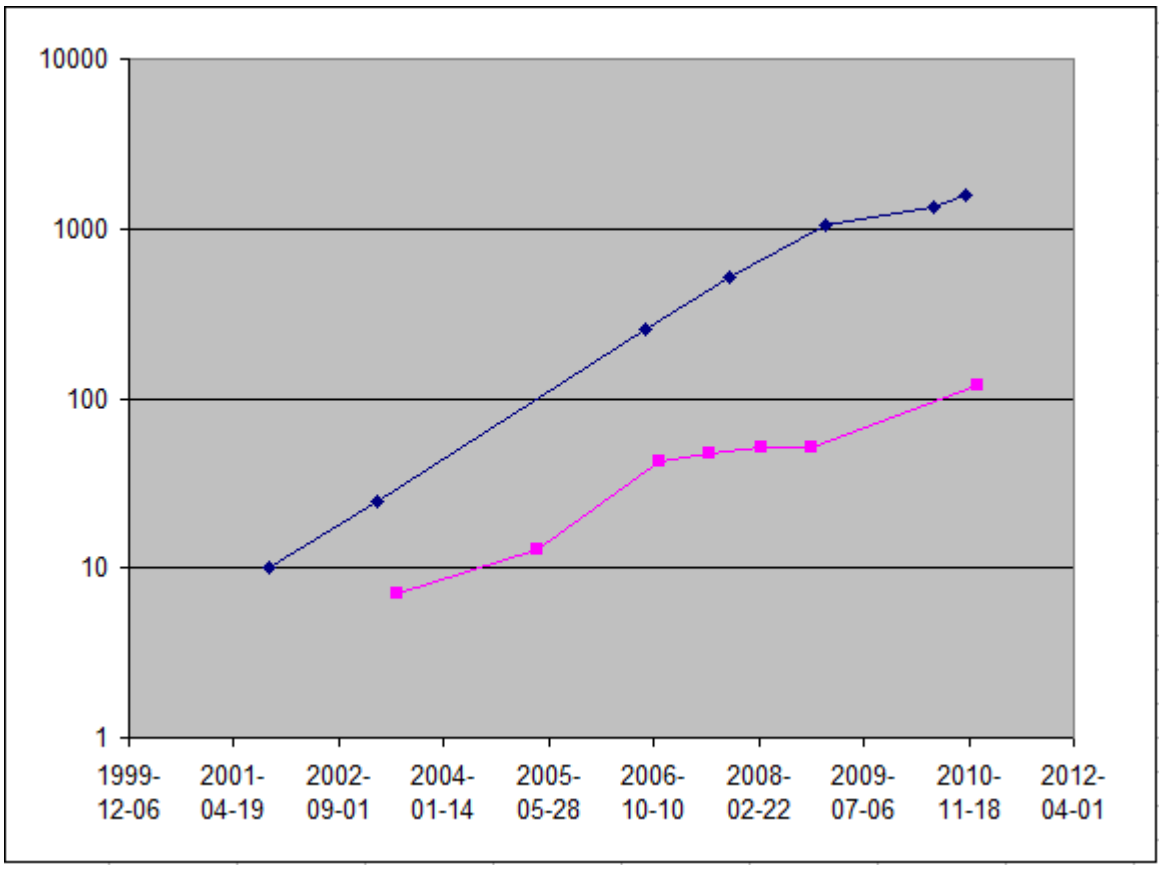
- Слева – кристалл CPU, на котором находятся 4 ядра (классические процессоры). Размеры полей рисунка пропорционально соответствуют их площади на кристалле.
- Справа – кристалл GPU, на котором находится 8 процессоров, под управление кождого из них находится 16 АЛУ. Значит, на кристалле находится 128 CUDA ядер.
-

Очевидно, что при равной частоте синхронизации на GPU будет выполнено значительно больше вычислительной работы.

Ниже предсавлена сравнительнаа таблица некоторых графических процессоров компании NVIDIA и AMD. Число АЛУ достигает нескольки тысяч.

	NVIDIA GTX 680	NVIDIA GTX 580	AMD Radeon HD 7970
Ядро	GK104	GF110	Tahiti
Количество транзисторов, млрд. шт	3.5	3.0	4.3
Техпроцесс, нм	28	40	28
Количество потоковых процессоров (АЛУ)	1536	512	2048
Частота ядра, МГц	1006	772	925
Шина памяти, бит	256	384	384
Объем памяти, Мбайт	2048	1536	3072
Частота памяти, МГц	6000	4008	5500
Мощность блока питания, вт	550	600	550

Увеличение быстродействия графических процессоров продолжится. Это видно из графика ниже.



На рисунке представлен сравнительный график роста быстродействия графических (синяя линия) и универсальных процессров (красная линия) по годам. По вертикали отложено быстродействие в Гфлопс/сек в логарифмическом масштабе. Для 2010 года быстродействие этих процессоров отличалось примерно на десятичный порядок.

На сегодня в списке TOP 500 в первой пятерке находится 3 компъютера, построенных на графических процессорах. Сейчас ведутся разработки с 1000 ядер на одном кристалле. Где предел? На этот вопросдан такой ответ:

Ключевым моментом роста является то, что наращивать количество «маленьких вычислителей» значительно проще, чем увеличивать тактовую частоту. Центральные процессоры не могут позволить себе такой рост именно из-за того, что не могут наращивать параллелизм такими же темпами. Поскольку они вынуждены хорошо исполнять традиционные приложения — OS, прикладные программы (Word, Excel к примеру) и они просто не могут себе позволить уменьшить вычислительное ядро. Они не могут деградировать производительность всех этих приложений.

Но не следует забывать, что графические процессоры в определенной степени все же являются специализированными процессорами, хотя область их применения довольно широка. Это - задачи с массовым параллелизмом, то есть задачи с большим объемом данных и прямолинейным без переходов исполнением.

Глава 6. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ В ГРИД

6.1. Некоторые этапы развития IT технологий

Некоторые этапы развития параллельных технологий. 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 на базе интернета:

- 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. Эта технология позволяет объдинить для решения одной задачи множество кластеров и отдельных процессоров. Грид в той или иной мере использует достижения упомянутых выше технологий, в особенности Web службы, многие элементы которых вошли в Грид.
- Облачные технологии. Это система доступа к ресурсам, в рамках которой информация постоянно хранится на серверах в интернете и временно кэшируется на клиентской стороне, например, на персональных компьютерах, игровых приставках, ноутбуках, смартфонах и т. д..

Интернет

Интернет есть единое информационное пространство, в котором можно строить различные сооружения - сайты, хосты, серверы и т. д. История Интернет началась с 1958 года, когда США создали организацию под названием DARPA. В 1969-м году была построена первая Сеть, основанная на современных принципах Интернет. К 1978-му году были выработаны все базовые протоколы, которые и сейчас используются в Интернет, в чстности:

- Адресный протокол IP (Internet Protocol address)
- Протокол HTTP (Hyper Text Transfer Protocol), в котором есть ссылки на другие гипертексты

WWW

WWW, World Wide Web, Всемирная паутина, Web, Веб, - это все названия одного и того же сервиса, который появился в 1991 году и использует протокол HTTP для передачи гипертекстовых документов и других файлов от Веб сервера к клиентам. Все перемещения по сети от одного документа к другому происходят по ссылкам. Эти документы написаны на языке HTML (HyperText Markup

Language). Этот язык позволяет работать практически со всеми доступными сейчас на компьютере видами документов: это могут быть текстовые файлы, иллюстрации, звуковые и видео ролики, и т.д. Программа просмотра HTML текстов называется browser (браузер).

Таким образом, 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 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 перевести, как "единый указатель на ресурс". Практически, это адрес документа.

Web services (Web службы)

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

Традиционно, используя Internet, клиениту придется посетить сервер авиакомпании, сервер гостиниц, сервер компании по аренде автомобилей и так далее. Более удобно было бы запустить приложение, которое бы приняло от клиента необходимую информацию и выполнило все эти рутинные действия Чтобы это стало возможным, следует использовать Web-сервисы.

Архитектура сети Web Services и взаимодействие между клиентами и службами представлены на рис.6.1.

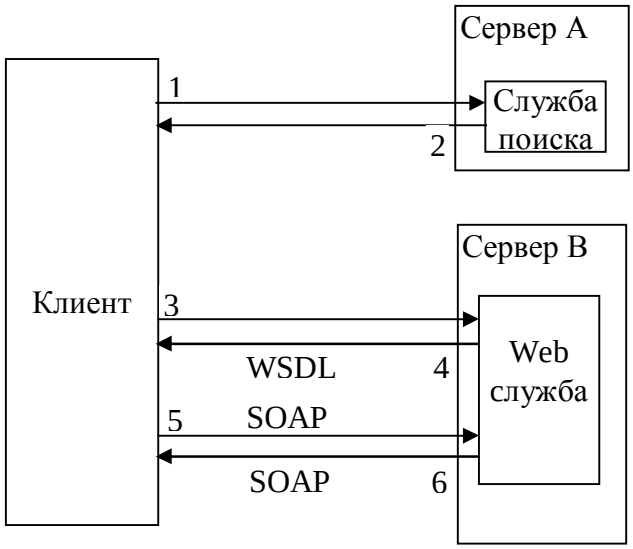


Рис. 6.1. – Схема взаимодействия клиентов и служб

Архитектура Web-служб предполагает слабую связность между компонентами сети, которая означает, что компонентам системы не обязательно знать, как устроены взаимодействующие с ними подсистемы, а для взаимодействия нет необходимости в создании специального программного обеспечения.

Web Services базируется на применении открытых стандартов и протоколов, 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.

Рассмотрим, как выполняется обращение к Web-службе. Этапы таковы:

- Сначала на языке UDDI производится обращение к сетевой справочной системе. В ответ предоставляется набор интернет адресов, содержащих требуемую службу (стрелки 1, 2).
- Затем производится обращение к службе по одному из адресов, представленных UDDI. Служба отвечает на языке WSDL, который предоставляет подробное описание возможностей службы и правил обращения к ней (стрелки 3, 4).
- Зная эти правила, пользователь на языке SOAP передает службе свое задание и получает ответ (стрелки 4, 5).

UDDI, WSDL, SOAP – это основные протоколы Web-служб, и реализованы они на системе адресации HTTP (TCP/IP) и языке XML (EXtensible Markup Language). XML - это в переводе "расширяемый язык разметки", предназначенный для описания данных и их типов. Для адресации в Web-службах используется простой URI (Uniform Resource Identifiers), подобный URL (Uniform Resource Location).

Клиентское приложение обращается к Web-службам на обычном для клиентов языке. Для того, чтобы перейти от этого языка к WSDL, необходим stub – специальный переходник (рис. 6.2). Рассмотрим типичный вызов WS.

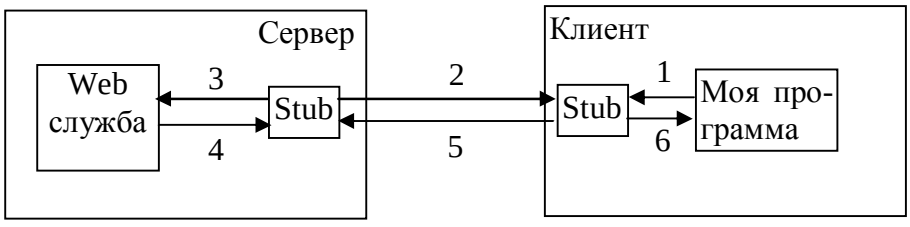


Рис. 6.2. – Обмен через переходник

1. При всяком обращении к Web-службе вызывается *stub*, который преобразует клиентское обращение в правильный запрос на языке SOAP.
2. SOAP запрос пересылается через сеть с помощью протокола HTTP. Сервер принимает SOAP запрос и передает его серверному *stub*, который преобразует его из SOAP в понятное серверу.
3. Серверный *stub* вызывает нужную Web-службу, которая выполняет работу.
4. Затем результат обрабатывается серверным *stub* в SOAP ответ.
5. Ответ пересылается через сеть на основе протокола HTTP, клиентский *stub* получает SOAP ответ и преобразует его в форму, понятную клиенту.

Язык XML

Основой для реализации всех этих протоколов является язык XML (EXtensible Markup Language - 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. 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,например 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.

Посмотрим 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, и не 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.

Грид

Под английским термином GRID понимается совокупность пространственно распределенных вычислительных узлов, связанных некоторой сетью для обмена данными. В дальнейшем вместо GRID будет использоваться слово Грид. Это слово 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 том, кем и где она произведена.

Различие между Web Service и Грид состоит в следующем:

- Web Service позволяет клиенту выполнить на оборудовании владельца ресурса некоторую функцию из списка, составленного владельцем этого оборудования.
- Грид - метод использования глобально процессорных мощностей и систем хранения информации (дисковые системы большой емкости) на основе повременной аренды без их физического перемещения в пространстве. Элементами Грид в основном являются кластеры, а не отдельные компьютеры.

Естественно, Грид включает все, наработанное в WebServices. Более того, протоколы WebServices (WSDL, SOAP, UDDI), средства адресации в расширенном варианте являются основными протоколами. Основная информация по Globus GRID представлена в [16].

Одной из причин создания европйского Грид явилась необходимость обработки громадного объема информации, которая поступает с Большого адронного коллайдера (БАК), созданного ЦЕРНЕ исследований. Для него использован 27-километровый подземный тоннель, проложенный на глубине около 100 метров на границе Швейцарии и Франции. БАК предназначен для разгона протонов и тяжёлых ионов, которые при столкновении на встречных пучках порождают новые частицы, изучение этих частиц будет способствовать изучению основ мироздания. С БАК за год будет поступать 10 Петабайт данных. Для обработки этого гигантского объема данных будет используется технология распределенных вычислений Грид.

Общая структура Грид не примере одного узла представлена на рис.6.4.

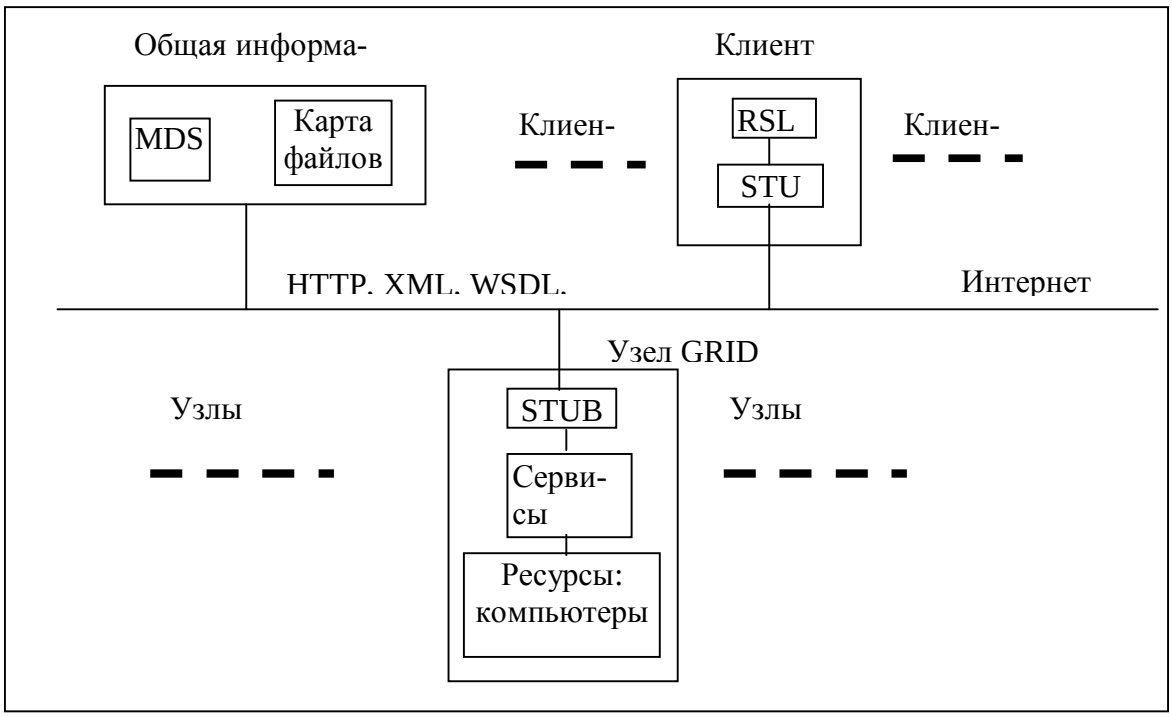


Рис.6.4 Организация ГРИД

Клиент обращается в реестр ресурсов MDS (Monitoring and Discovery Service), чтобы получить сведения о наличии и месте расположения нужного ресурса. Затем клиент обращается к ресурсу, чтобы получить интерфейс ресурса, в котором указывается способ задания ресурсу требуемой работы. Наконец, клиент передает ресурсу задание на языке RSL (Resource Specification Language) и получает результат.

В глобальной сети находися десятки тысяч (и более) узлов, каждый из них может быть как ресурсом, так и клиентом. Кроме клиентов и ресурсов имеются общие службы, например, справочная система и карта расположения файлов. Главное состоит в том, что на каналах обмена используется стандарт Web-служб: XML, WSDL, SOAP и др.

В среде Грид явным образом присутствуют следующие элементы:

- Программы пользователя
- Ресурсы (аппаратура, ОС, кластерное ПО и т.п.)
- Промежуточное программное обеспечение (**Middleware**) , выступающее в роли посредника между 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:

- GT4 - разработан в США по проекту Globus. Он будет рассмотрен далее.
- gLITE. Его условно можно назвать европейским проектом, поскольку его разработка курируется ЦЕРН.

Облачные вычисления

Появление Грид является следствием непрерывного развития различных реализаций на базе Грид. Нетрудно понять, что Грид не может быть конечной точкой развития интернет.

Термин «Облако» используется как метафора, основанная на изображении Интернета на диаграмме компьютерной сети, или как образ сложной инфраструктуры, за которой скрываются все технические детали [17].

Согласно документу IEEE (2008 год) «Облачная обработка данных — это парадигма, в рамках которой информация постоянно хранится на серверах в интернете и временно кэшируется на клиентской стороне, например, на персональных компьютерах, игровых приставках, ноутбуках, смартфонах и т. д.».

Облачная обработка данных как концепция включает в себя понятия:

- Всё как услуга .
- Инфраструктура как услуга.
- Платформа как услуга.
- Программное обеспечение как услуга.
- Данные как услуга.
- Рабочее место как услуга.

Облако для пользователя – это некоторый набор услуг (soft и hard), которые потребляются и оплачиваются, порой без малейшего представления, что же там используется внутри.

Варианты предоставления ресурсов сильно отличаются. Все, что касается Cloud Computing, обычно принято называть словом aaS (aaS – расшифровывается как "as a Service"). Существуют много видов aaS, например:

- SaaS (Software-aaS), или приложения в виде сервисов - вариант, при котором тебе предлагают использовать какое-то конкретное ПО, например, корпоративные системы, в виде сервиса по подписке
- PaaS (Platform-aaS) - в отличие от SaaS, предназначенного больше для конечного пользователя, вариант для разработчиков. В облаке функционирует некоторый набор программ, основных сервисов и библиотек, на основе которых предлагается разрабатывать свои приложения.
- HaaS (Hardware-aaS) - один из первых терминов, означающих предоставление некоторых базовых "железных" функций и ресурсов в виде сервисов.
- CaaS (Communication-aaS) - подразумевается, что в качестве сервисов предоставляются услуги связи; обычно это IP-телефония, почта и мгновенные коммуникации (чаты, IM).

Облачные технологии обеспечивают более совершенный доступ к ресурсам. Для решения решения больших задач нужно другое - увеличение числа процессоров. Поэтому облачные технологии далее рассматриваться не будут.

Что дали для быстродействия:

Интернет	IP (Internet Protocol address) HTTP (Hyper Text Transfer Protocol)
WWW	Язык гипертекстовой разметки HTML Системы адресации (URL, URN, URI etc.)
WebServise	SOAP (Simple Object Access Protocol) WSDL (Web Services Description Language) UDDI (Universal Description, Discovery and Integration) Язык XML
Грид	Пакеты GT4, G2
Облачная технология	

6.2. Пакет Globus Toolkit.

Состав пакета GT4. 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 службы, которые позволяют построить полнофункциональную Грид систему. Пакет G T4 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.

Далее будут рассмотрены только два пункта из приведенного перечня: обеспечение безопасности и управление исполнением заданий.

6.2.1. Security - Обеспечение безопасности

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 %.

Криптографические системы с открытым ключом используют [18] так называемые однонаправленные функции, которые обладают следующим свойством:

- Если x известно, то y = f (x) вычислить относительно просто.
- Если известно y, то для f (y) нет простого пути вычисления x.

Под однонаправленностью понимается не теоретическая однонаправленость, а практическая невозможность вычислить обратное значение, используя современные вычислительные средства, за обозримый интервал времени. Такими функциями являются, например, дискретное логарифмирование, разложение большого числа на простые множители и др.

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:

$$y=a^x \pmod p. \qquad (6.1)$$

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.6.1 – Открытые ключи

Имя	Открытый ключ
A	$У_A$
B	$У_B$
C	$У_C$
…..	…..

Пользователь А, посылая сообщение В, формирует закрытый ключ вида

$$Z_{ab} = (y_B)^{x_A} = (a^{x_B})^{x_A} = a^{x_B x_A} \qquad (6.2)$$

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

$$Z_{ba} = (y_A)^{x_B} = (a^{x_A})^{x_B} = a^{x_A x_B} \qquad (6.3)$$

Поскольку $Z_{ab} = Z_{ba}$, то оба абонента имеют общий закрытый ключ, полученный без его передачи по открытой сети. Таким образом, выражения (6.1), (6.2), (6.3) 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 криптографической системе RSA каждый ключ состоит из пары целых чисел.

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

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. При внедрении и развертывании Грид систем инфраструктура безопасности является наиболее важным звеном т.к. от нее напрямую зависит безопасность всей системы.

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.

Иногда к аутентификации предъявляются строгие требования. Это достигается на основе цифровых сертификатов. Цифровой сертификат (рис.6.5) есть цифровой документ, который свидетельствует, что определенный открытый ключ принадлежит владельцу закрытого ключа. Этот документ подписывается третьей стороной, называемой certificate authority (CA).

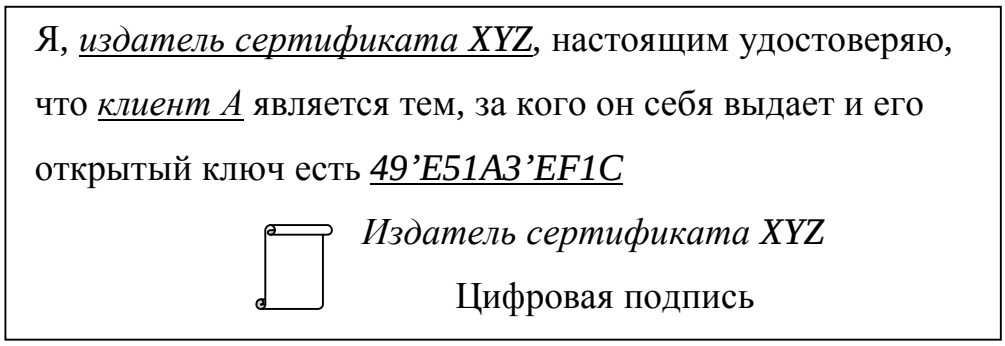


Рис.6.5. Формат цифрового сертификата.

Подпись издателя есть в действительности цифровая подпись, сгенерированная с помощью закрытого ключа CA, поэтому мы можем проверить целостность сертификата, используя открытый ключ CA. Если вы подписываете сообщение вашим закрытым ключом и посылаете получателю копию 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. Иерархия включает список всех CA, которым можно доверять, включая и ваш CA. 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.

6.2.2. Execution Management - Управление исполнением заданий

Система управления исполнением заданий имеет многоуровневую структуру. Задание представляется на языке RSL, который предназначен для обмена информацией о ресурсах между всеми компонентами метакомпьютерной среды Globus. RSL также описывает размещение программы пользователя и данных, требуемые системные программные и аппаратные средства, средства синхронизации процессов. Приведем основные стандартные параметры:

(executable = value) – определяет выполняемый файл задания;
(directory = value) – задает рабочую директорию для задания;
(arguments = value ...) – задает параметры командной строки ;
(environment = (var value)...) – задает среду выполнения задания;
(count = value) – задает число процессов (экземпляров выполнения executable);

Следующий пример илюстрирует RSL строки, илюстрирующие использование и семантику переменных:

```
& (rsl_substitution  = (TOPDIR  "/home/nobody")
            (DATADIR "/home/nobody/data")
            (EXECDIR "/home/nobody/bin") )
  (executable = "/home/nobody/bin/a.out" )
  (directory  = "/home/nobody" )
  (arguments  = "/home/nobody/data/file1"
            "/home/nobody/data/file2"
            "$(FOO)" )
  (environment = (DATADIR "/home/nobody/data"))
  (count = 1)
```

Здесь (count = 1)определяет аппаратные средства, указывая, что в задаче используется один процесс (процессор). Описания требуемого оборудования могут включать логические связки, задающие более сложную структуру аппаратных средсв. Например, текст:

```
& (count>=5) (count<10)
   (max_time=240) (memory>=64)
    (executable=myprog)
```

означает, что будет использоваться 5-10 процессоров с памятью не менее 64 MB для выполнения программы myprog в течение 4 часов.

Система управление исполнением заданий имеет многоуровневую структуру (рис.6.6).

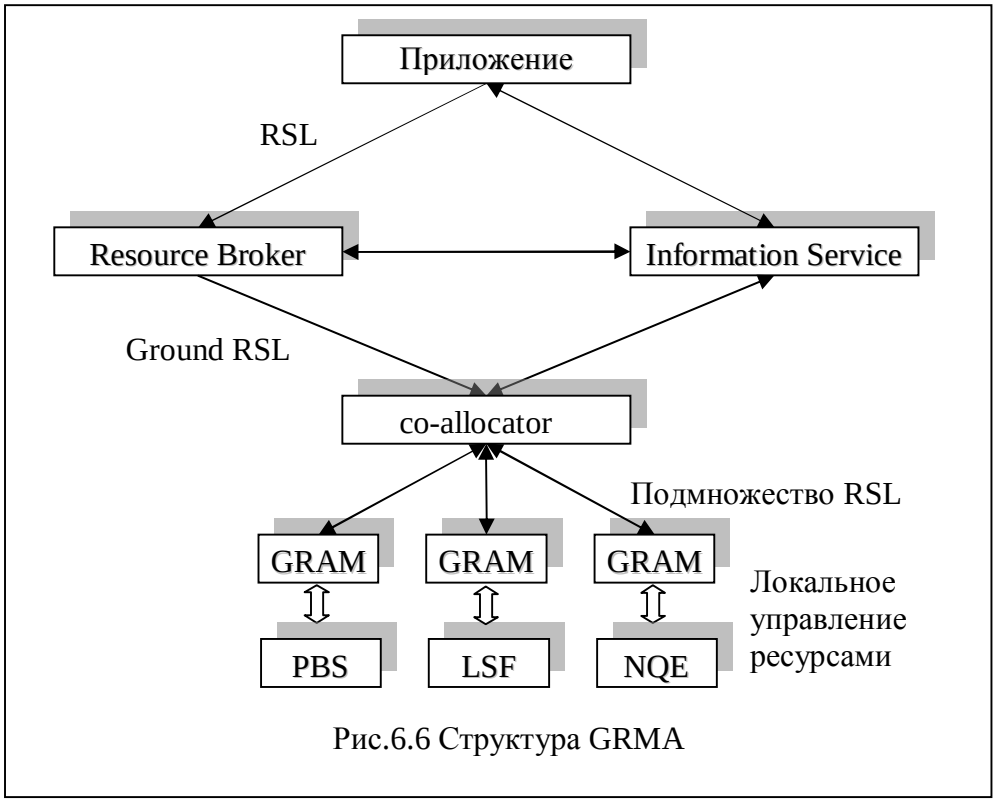


Рис.6.6 Структура GRMA

Система информационного сервиса Information Service. Основным элементом системы является (рис.6.7) подсистема **MDS** в которой зарегистрированы все ресурсы Грид. Информация о ресурсах может содержать 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). MDS состоит из двух 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.

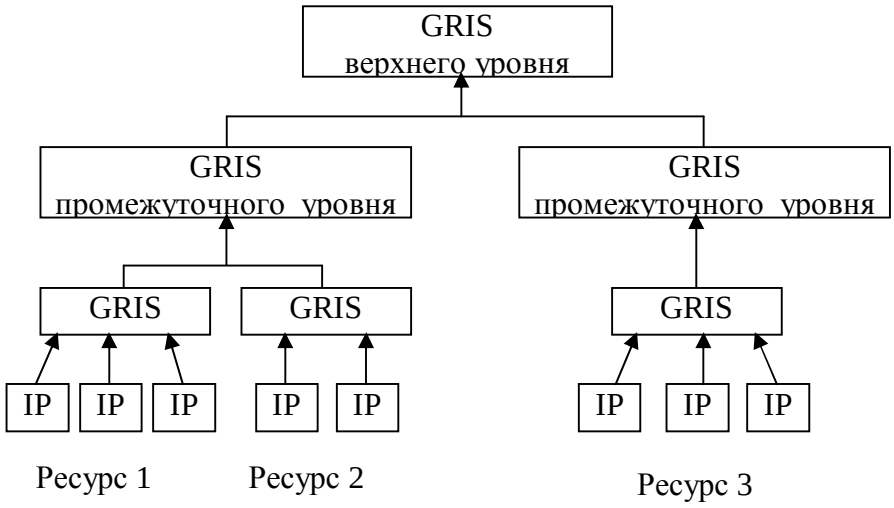


Рис. 6.7. Подсистема обновления и отыскания сервисов.

Resource Broker - высокоуровневый менеджер ресурсов на основе информации из Information Service должен произвести коллективное выделение ресурсов, то есть определить, какие узлы доступны в текущий момент, их загрузку, производительность и другие параметры, указанные в RSL-запросе, выбрать наиболее оптимальный вариант, сгенерировать новый RSL-запрос (ground RSL) с описанием предложенных ресурсов и передать его планировщику co-allocator. Этот запрос будет содержать уже более конкретные данные, такие, как имена конкретных узлов, требуемое количество памяти и др.

Co-allocator - производит детальное распределение работ по подмножествам процессоров и понижает описание работ до уровня заданий, понятных менеджеру GRAM (Globus Resource Allocation Manager).

GRAM (рис.6.8) – является интерфейсом между высокоуровневым менеджером ресурсов и локальной системой управления ресурсами узла.

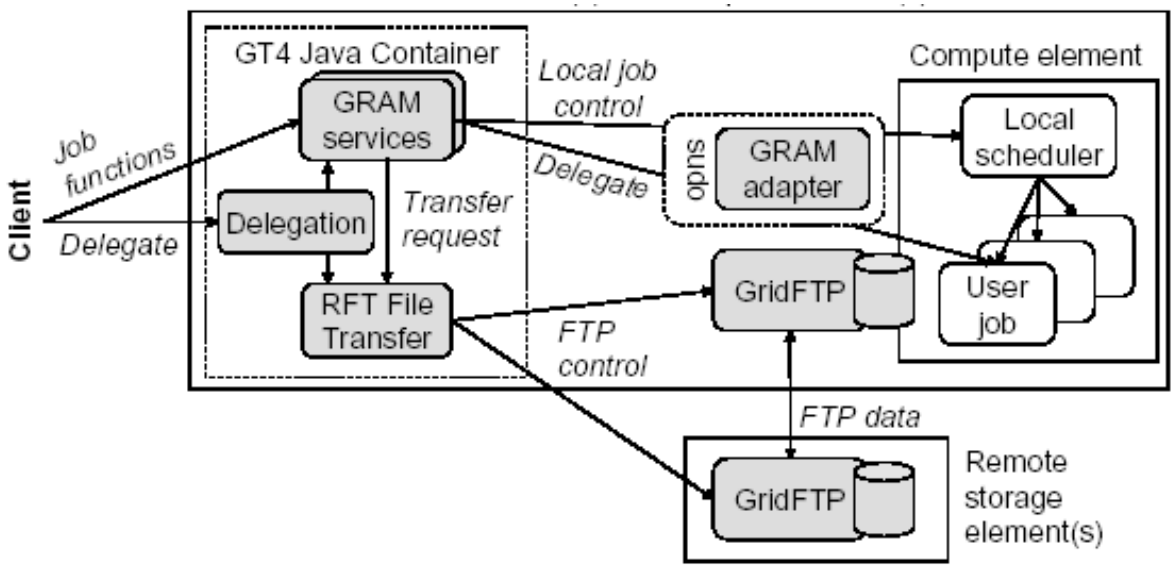


Рис.6.8. Схема GRAM

Этот интерфейс может взаимодействовать с различными локальными системами управления кластерами: PBS, LSF, NQE и др. GRAM выполняет следующие функции

- **Delegation** - производит взаимную аутентификацию с клиентом.
- **GRAM services** - запускает процесс вычислений на кластере, передает его планировщику список требуемых аппаратных и программных ресурсов, обеспечивает мониторинг этапов выполнения работы, сообщая клиенту об ошибках и других событиях.
- **RFT** (Reliable File Transfer) - управляет передачей данных между данным узлом и другими, а также обеспечивает транзитную передачу.

Пакет GT4 имеет модульную структуру и устанавливается и настраивается в каждом узле индивидуально. Resource Broker и co-allocator работают на узле пользователя, а сервис GRAM работает на вычислительном ресурсе. GRAM принимает на вход описание задачи пользователя, связывается с локальной системой управления и передает ей инструкции для запуска задачи. Дополнительно поддерживается проверка состояния выполнения задачи, ее удаление и сбор информации о состоянии вычислений в целом. Сервер RFT реализует передачу файлов между различными узлами Грид. Клиент RFT формирует заявку на передачу файла, которая состоит из в указании двух URL на файлы: откуда и куда копировать.

Программы командной строки пользователя (таб. 6.2) позволяют вызвать компоненты непосредственно из программы пользователя.

Таб. 6.2 - 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

Для задания работы в GRAM испльзуется команда **globusrun-ws**. В качестве 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.

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.

Существует много форматов, созданных различными разработчиками, поэтому за структурой RSL надо обращаться к документации конкретного разработчика.

Планирование MPI вычислений в системе Condor. Condor – одна из многих 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 через RSL, Condor ставит работу в очередь, запускает ее и затем информирует пользователя о результатах.

6.3. Параллельные вычислени в ГРИД. Пакет G2.

Для выполнения параллельных вычислений на кластерах используется библиотека MPI. Пакет MPICH-G2 [19] есть Грид ориентированная полная реализация стандарта MPI-1, которая использует службы Globus Toolkit для прозрачной работы в среде Грид, в том числе и гетерогенной.

MPICH-G2:

- позволяет программисту соединять компьютеров различной архитектуры;
- планирует распределение ресурсов;
- автоматически преобразовывает данные в сообщениях между компьютерами различной архитектуры;
- освобождает пользователя от работы по изучению специфики конкретных машин и позволяет пользователю запускать многопроцессорное приложение одной командой mpirun.

Схема выполнения вычислений на MPI соответствует рис.6.6, но в качестве планировщика co-allocator используется программа DUROC которая:

- Из всех возможных ресурсов выбирает только такие, которые могут работать с MPI.
- Выбранные ресурсы разнесены в пространстве, различаются по характеристикам (быстродействие процессоров, объем памяти, скорость передачи данных по каналам связи и др.), поэтому времена выполнения локальных вычислений могут оказаться разными. Следовательно, неоходимо установить факт окончания всех локальных работ и обменов данными, прежде чем запускать следующие отрезки локальных вычислений. Это достигается с помощью операторов синхронизации, например, MPI_Barrier.
- Оптимизирует выбор путей обмена данными с учетом их пропускной способности и последовательности обменов, чтобы сократить общее время передачи данных. Каналы отличаются по пропускной способности: intra-machine messaging, short (LAN) and long (WAN). По этой стратегии MPICH-G2 упорядочивает различные коммуникационные методы на следующих допущениях:

WAN-TCP < LAN-TCP < intra-machine TCP < vendor-supplied MPI.

Порядок запуска проложения таков:

- Чтобы запустить MPICH-G2 приложение, пользователю нужно получить открытый ключ, который используется для аутентификации пользователя на каждом удаленном сайте.
- Когда идентификация произведена, пользователь использует стандартную **mpirun** команду, чтобы запросить создание MPI вычислений. Реализация **mpirun** использует скрипты RSL, которые пишут пользователи. В них идентифицируются ресурсы (то есть компьютеры), описывают требования (количество CPU, памяти, требуемого времени и т. д.) и параметры (размещение программ, аргументы командной строки, переменные среды и т.д.) для каждого ресурса.
- На основе этой информации MPICH-G2 вызывает DUROC, чтобы спланировать и запустить приложение на различных компьютерах. DUROC реализует операцию размещения через множественные RM (ресурсные менеджеры) в рамках Globus.

- Для каждого подвычисления DUROC генерирует GRAM запрос к удаленному GRAM серверу, который распознает пользователя, выполняет локальную авторизацию и затем взаимодействует с локальным планировщиком, чтобы инициировать вычисления. DUROC и связанные с MPICH-G2 библиотеки связывают различные подвычисления в единое MPI вычисление.
- DUROC и GRAM также взаимодействуют при мониторинге и выполнении приложения. Каждый GRAM сервер следит за жизненным циклом его вычислений на всех стадиях: задержка, выполнение, окончание.

6.4. 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 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").

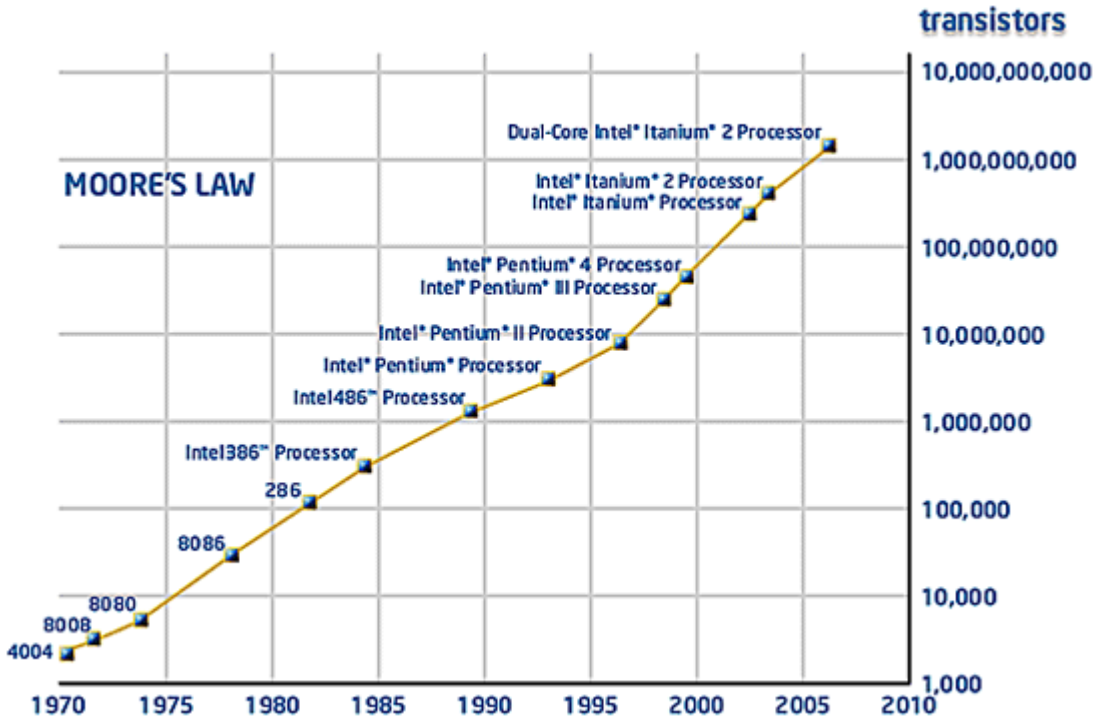
Глава 7. МНОГОЯДЕРНЫЕ ПРОЦЕССОРЫ.

7.1. Многоядерные микропроцессоры. Закон Мура для ядер.

Развитие микропроцессорной техники в области универсальных микропроцессоров идет по пути постоянного повышения их производительности. Традиционными направлениями такого развития являются:

1. Повышение тактовой частоты работы МП, которое в основном обеспечивается путем увеличения количества ступеней в конвейере, что приводит к большим потерям времени при необходимости перезагрузки конвейера вследствие зависимости команд по данным и конфликтов по управлению. Кроме того, при длинных конвейерах время передачи данных с этапа на этап становится соизмеримым со временем выполнения этапа.
2. Увеличение количества одновременно выполняемых команд за счет увеличения числа конвейеров в МП. Это слишком усложняет управление МП, да и число команд в программе, которые можно выполнить одновременно, невелико.

В связи с успехами технологии в области микроэлектроники появился третий путь. На сегодня сохраняется экспоненциальный рост числа транзисторов (рисунок ниже), который описывается экспериментальным законом Мура [9]



Это количество транзисторов на кристалле позволяет строить на одном кристалле большое количество законченных процессоров, которые называют многоядерными.

Многоядерный процессор (МЯП) – это центральный процессор, содержащий два и более вычислительных ядра на одном процессорном кристалле. Каждое ядро по функциональным и техническим возможностям возможностям мо-

жет соответствовать центральному или специализированномц процессору. Ядра одного многоядерного процессора могут быть одинаковыми или различаться по своим возможностям. Основные сведения по МЯП расположены по адресу [20].

Закон Мура, определяющий рост числа транзисторов на кристалле, формулируется следующим образом:

Количество транзисторов на кристалле удваивается каждые два года

Мур указал также, что результатом роста числа элементов на кристалле станет снижение стоимости на элемент. При этом, темпы увеличения числа элементов (и функций, соответственно) – выше, чем темпы роста стоимости производства кристалла. Это и является движущей силой развтия многоядерности.

Дъяконов [21] записал закон Мура в математической форме:

$$\hat{E} = \hat{E}_0 2^{\frac{t-t_0}{2}} = \hat{E}_0 2^{\frac{\Delta t}{2}}$$

где K, K_0, t,t_0 – количество транзисторов начальное и через врем Δt. Если взять пример микропроцессора *i*486 с параметрами t_0 – 1990, $K_0 = 10^6$, то к 2010 получаем

$$\hat{E} = 10^6 2^{\frac{20}{2}} = 10^9,$$

что примерно соответствует таблице.

Если предположить, что на одно ядро тратится 10^7 транзисторов, то получаем 100 ядер, что соответствует действительности (48, 80 ядер от Intel). Таким образом, можно считать, что закон Мура распространяется и на многоядерные процессоры:

Количество ядер на кристалле удваивается каждые два года

По этому закону следует ожидать к 2020 коду 1000 ядер на одном кристалле.
Следует уточнить понятие ядра:

- Упоминавшийся в главе 1 суперкомпьютер СКИФ-Аврора имеет 2048 ядер, но это не ядра одного кристалла, а суммарное число ядер всех 4-ядерных процессоров Intel, использованных в этом суперкомпьютере.
- Специалисты из Токийского университета создали процессор, который состоит из 512 ядер, расположенных на одном кристалле Каждое из 512 ядер отвечает за отдельную математическую операцию. Этот кристалл с частотой 500 мегагерц соединен с картой PCI-X и осуществляет поддержку центрального процессора. По существу это не ядра, а специализированные АЛУ. Ядром же следует считать полноценный процессор.

Многоядерные процессоры могут использоваться двояко:

1. **Как автономные устройства**: ПЭВМ, рабочие станции, серверы и др. В этом случае критерием использования МЯ является увеличение номинального быстродействия МЯ кристалла при резком снижении стоимости одного ядра.
2. **Как элементная база** вычислительных кластеров. Здесь критерий другой - максимальное быстродействие кристалла при жестком ограничении на потребляемую мощность и нагрев (это разные вещи). В этом случае количество ядер не ограничивается – чем больше, тем лучше.

Однако на этом пути возникает следующее препятствие – нагрев кристалла, что потребуетособых методов охлаждения кристалла, а в случае многокристаллных систем в болших кластерах приведет к мегаваттному потреблению энергии.

На рис.7.1 слева представлен обычный кремниевый транзистор.

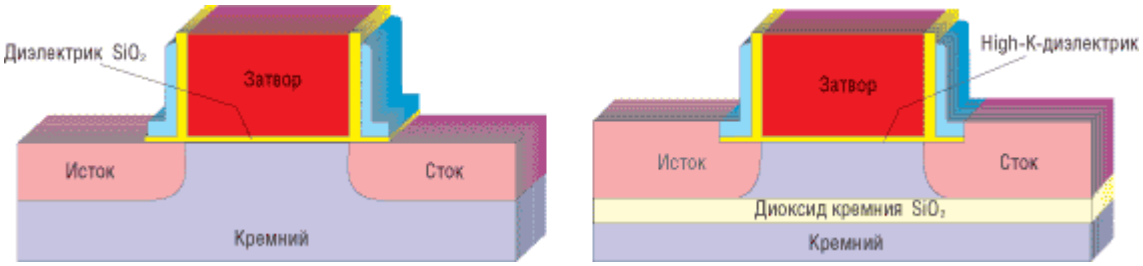


Рис.7.1. Обычный транзистор (слева) и транзистор с изоляцией против утечек (справа)

При уменьшении размеров такого транзистора в области наноразмеров (ниже 100 нм) в m раз параметры транзистора и кристалл, на котором он расположен, изменяются, как указано в таблице [22]:

Характеристика	**Коэффицтент**
Длина затвора	1/m
Напряжение	1/m
Плотность размещения	m^2
Скорость	m
Рассеиваемая мощность	$1/m^2$

- в m^2 раз возрастает количество транзисторов на кристалле, что в идеале увеличивает в m^2 раз быстродействие МЯ за счет увеличени числа ядер.
- кроме того, в m раз уменьшается рсстояние между истоком и стоком, поэтому в m раз растет скорость переключения транзистор, то есть частота. Следовательно, при увеличении m рост быстродействия кристалла имеет порядок $V_{кр} = m^2 * m = m^3$.

Из таблицы следует, что рассеиваемая кристаллом равна:

Плотность размещения * Мощность транзистора = m^2 *1/m^2 = const

Это означает, что при росте транзисторов (ядер) на кристалле нагрев кристалла не меняется . При этом кристалл рассеивает около 100 вт, для чего требуется стандартный радиатор и вентилятор. Это идеальная ситуация.

Более того, если учесть, что частота f = k1×m, а количество процессоров p = k2× m×m, то быстродействие вычислительных устройств растет так:

$$V = f \times p = k1 \times m \times k2 \times m \times m = k \times m^3(t),$$

то есть по годам скорость вычислений растет кубически, причем, в большей степени за счет числа ядер, а не их частоты.

К сожалению, уменьшение линейных размеров транзистора до 30 – 10 нм приводит к возникновению токов утечки вследствие туннельного эффекта. Токи утечки возникают (рис.6.1 слева):

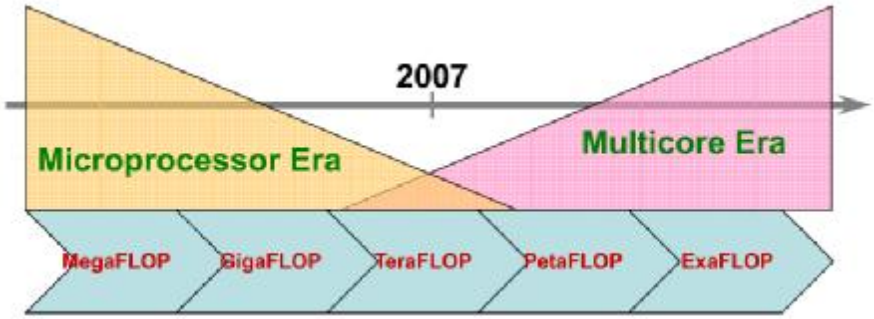
- через слой диэлектрика, отделяющего область затвора от подложки,
- между истоком и стоком при «выключенном» состоянии транзистора.

Эти токи соизмеримы с рабочими токами транзистора, и при увеличении числа транзисторов на кристалле, он бедет сильно перегреваться. Если не принимать мер, это будет существенно ограничивать рост числа транзисторов и ядер.

Использование новых изолирующих материалов (рис.6.1 справа), как считают специалисты, продлевает действие закона Мура до 20 – го года, сохраняя расеиваемую кристаллом мощность до 100 вт.

Однако, этих мер в дальнейшем может оказаться недостаточно. Тогда применяют дополнительные меры, например: межслойное водяное охлаждение в кристалла типа «сэндвич», избирательное отключение незагруженных ядер и узлов, снижение тактовой частоты.

Переломным при переходе от микропроцессоров к многоядерным кристаллам явился 2007 год [23], но происходило это не мгновенно, а было растянутого во времени.



7.2. Две архитектуры многоядерных процессоров.

В области МЯП следует различать понятия: архитектура МЯП и микроархитектура ядра. Под микроархитектурой ядра понимают состав арифметико – логических блоков (могут быть сопроцессоры VLIW, MMX), блоков обращения к памяти и реализации условных переходов. Архитектура же определяет связи ядер между собой (общая или индивидуальная память), с памятью, типы коммутаторов, связь с внешним миром.

Далее будут рассмотрены два типа многоядерных процессоров:

- **Nehalem** - 8 – ядерный процессор с общей памятью для всех ядер.
- **Polaris -** 80 – ядерный экспериментальный процессор с индивидуальной памятью для каждого ядра.

В литературу эти два типа многопроцесрных систем имеют названия **Multicore** и **Manycore**. Различие между этими двумя архитектурами довольно очевидное.

Архитектура с общей памятью:

- Облегчает программирование, поскольку не надо описывать перемещение данных между ядрами, данные находятся в общей для всех ядер памяти.
- Использование существующего программного обеспечения.
- Высокая надёжность системы, поскольку число ядер невелико.
- Главный недостаток –число ядер невелико из-за ограничений общей памяти. Система кэширования сглаживает этот фактор.

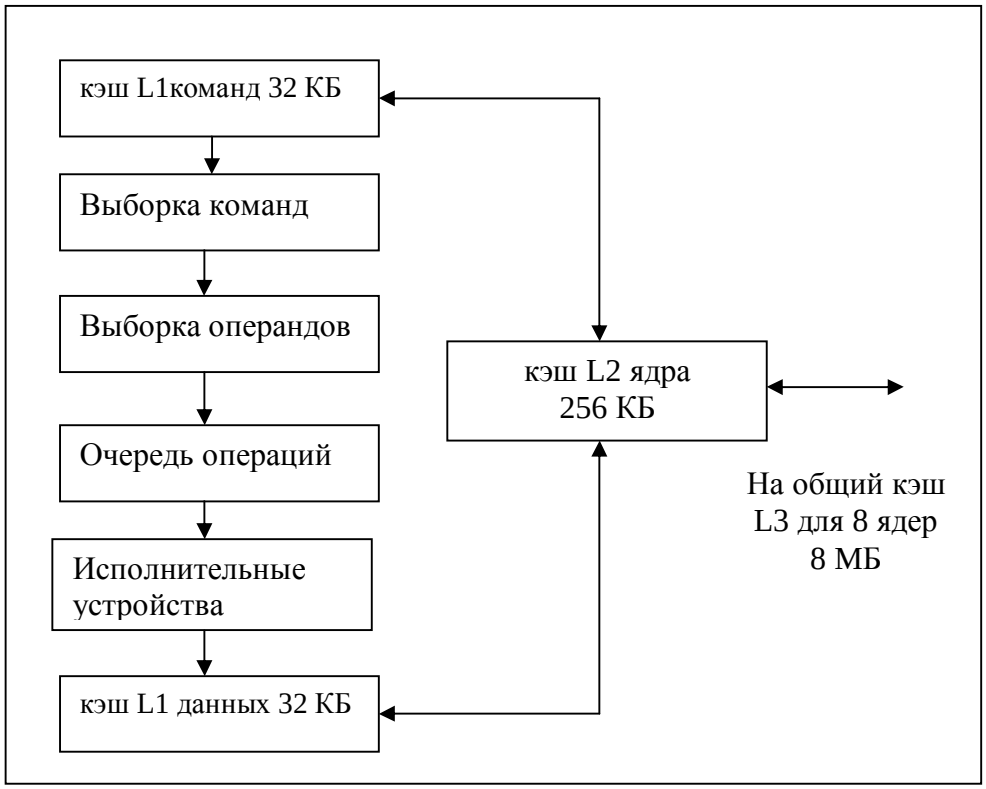
Архитектура с индивидуальной памятью для каждого ядра использует малые ядра с короткими конвейерами и небольшими частотами. Но, малые ядра имеют меньшее энергопотребление, на кристалле их может быть существенно больше, они обеспечивают наилучшую вычислительную эффективность на ватт.

7.3. Процессор Nehalem

Nehalem - однокристальный процессор с архитектурой общей (разделяемой) памяти. Он имеет следующие технические характеристики:

- Год выпуска 2008 – 2009
- Процессоры содержат до 8 ядер.
- 45-нм технологический процесс.
- Тактовая частота 3 ГГц.
- 2,3 млрд транзисторов.
- Тепловыделение 95 – 130 вт.
- Ядро одновременно может обрабатываться 2 потока.
- В Nehalem каждый процессор имеет собственную ОС. Это отличает его от общепринятого понимания SMP систем.

Микроархитектура ядра. Ядро Nehalem выполняет стандартную для микропроцессоров последовательность шагов, показанную на рисунке.



В ядре используются 3 уровня кэш – памяти:

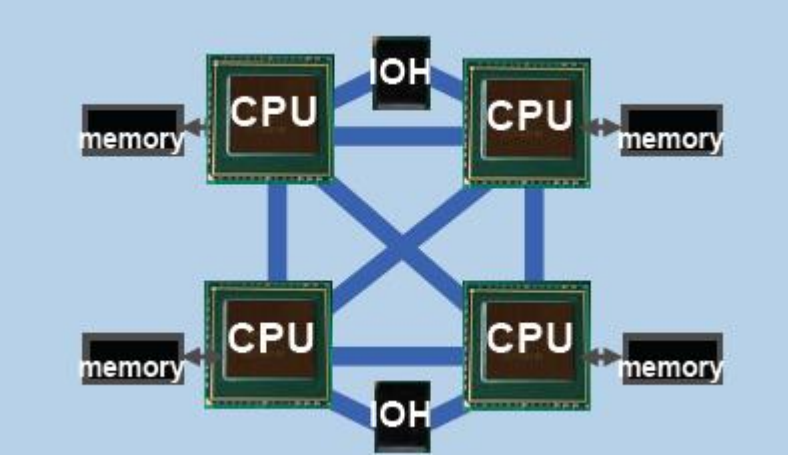
- 32 KB L1 - кэш для программ и 32 KB L1 кэш для данных на одно ядро
- 256 KB L2 - кэш на ядро
- до 24 MB L3 - кэш для всех ядер

Состав исполнительных устройств одного ядра. Существенно поработав над предварительными стадиями конвейера Nehalem, инженеры Intel оставили исполнительные устройства нового процессора без заметных изменений.

Блок исполнительных устройств содержит 6 портов. Порты 0,1 управляют работой 8 арифметических устройств для выполнения целочисленных и операций с лавающей точкой. Порты 2-4 обеспечивают работу памяти, а порт 5 выполняет некоторые арифметические операции и управляет операциями переходов. Следовательно, в Nehalem может одновременно выполняться 6 операций.

Кроме скалярных операций ядро может выполнять команды SSE (или MMX) и потоки для Hyper-Threading. Технология SSE позволяла преодолеть 2 основные проблемы MMX — при использовании MMX невозможно было одновременно использовать инструкции сопроцессора, так как его регистры использовались для MMX и работы с вещественными числами. SSE включает в архитектуру процессора восемь 128-битных регистров, каждый из которых трактуется как 4 последовательных значения с плавающей точкой одинарной точности. SSE включает в себя набор инструкций, который производит операции со скалярными и упакованными типами данных.

Архитектура кристалла. Процессор Nehalem имеет по четыре широкополосных шины QuickPath. Решение, выбранное Intel под названием QuickPath Interconnect (QPI), не является чем-то новым; оно представляет собой встроенный контроллер памяти и очень быструю последовательную шину "точка-точка".



С технической точки зрения интерфейс QPI является двунаправленным с двумя 20-битными шинами, по одной на каждое направление, из которых 16 зарезервировано под данные, а оставшиеся четыре - под функции исправления ошибок или служебную информацию протокола.

Это позволяет:

- создавать полносвязные четырёхпроцессорные системы, когда каждый процессор может получать доступ к любой области памяти через один хоп QPI, поскольку каждый процессор напрямую подключён к трём
- позволяет строить системы с несколькими процессорными сокетами, способными обрабатывать одновременно до 128 (8*8*2) процессов без использования дополнительных устройств.

IOH – хаб – для связи с внешним миром: центральным процессором или другими кристаллами Nehalem

Программирование. Система программирования Posex для потоков является низкоуровневым инструментом программирования, поэтому требует больших знаний и усилий. Интерфейс **OpenMP** является стандартом для программирования на масштабируемых SMP-системах с разделяемой памятью, в частности, для многоядерных процессоров с разделяемой памятью. OpenMP идеально подходит для распараллеливания программ 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.

Что же касается Hyper-Threading (HT), то тут дело обстоит следующим образом. Один процесс изначально рассчитан на исполнение только одного по

тока команд (один СчАК). У него есть только один регистр-указатель выборки команд и один набор регистров общего назначения. Процессор не может физически выполнять несколько потоков одновременно.

Процесс (process) — это некоторая часть (единица) работы, создаваемая операционной системой. Программа может состоять из нескольких процессов. Процесс задается адресным пространством и идентификатором. Адресное пространство процесса делится на три логических раздела: текстовый (код программы), информационный (данные) и стековый (для стеков программы).

Процесс может содержать несколько потоков. Различие между потоками и процессами состоит в том, что каждый процесс имеет собственное адресное пространство, а потоки — нет. Ресурсы, открытые родительским процессом, немедленно становятся доступными всем потокам.

Использование потоков имеет ряд преимуществ:

- Для переключения контекста требуется меньше системных ресурсов.
- Достигается более высокая производительность приложения.
- Для обеспечения взаимодействия между задачами не требуется никакого специального механизма.
- Программа имеет более простую структуру.
- Переключение процессора на поток минимизировано вплоть до операций сохранения/восстановления этих указателей.

К сожалению, многопоточное параллельное программирование требует повышенных усилий от программиста, касающихся обеспечения корректности параллельной программы (однозначности результата её выполнения независимо от временных характеристик индуцируемых потоков и использования общих переменных), её эффективности (частое "столкновение" потоков при обращении к общим программным и аппаратным ресурсам).

Потоки живут только из-за простоев памяти , блоков АЛУ и др. Рссмотрим пример. Если оба потока содержат мало вычислений и активно обращаются к памяти, из-за единого интерфейса к памяти выполнение потоков превращается практически в последовательное (невозможно ускорить операцию memcpy(), разделив ее на потоки).

Сфера применения многоядерных процессоров с общей памятью:
Как элементная база всех современных кластеров.

- Как автономное средство в качестве ПЭВМ, серверов, рабочих станций.

В последнем случае сфера их приложения разнообразна:

- 2D/3D САПР.
- Системы моделирования, средства работы с анимацией;
- Средства обработки цифровых изображений.
- Электронные издательские системы.
- Средства видеомонтажа/рендеринга.
- Компьютерные игры (на клиентских компьютерах и серверах).
- Финансовое моделирование.
- Научные и технические расчеты.

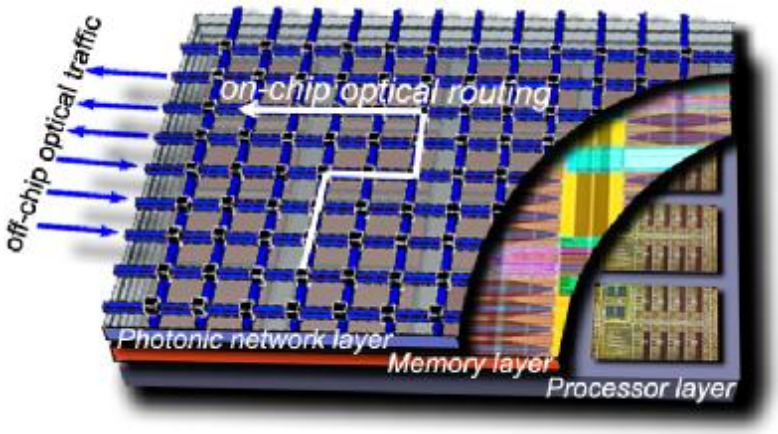
7.4. Процессоры с индивидуальной памятью.

Для решения проблемы низкой пропускной способности памяти идут по пути построения на кристалле многопроцессорной системы с индивидуальной памятью для каждого ядра. Такая система широко применяется в вычислительных кластерах. Это дает следующие преимущества:

При проектировании использовались следующие прогрессивные технологии:

1. Использовались малые ядра. Такие ядра используют, если хи число превышает 100. Преимущества малых ядер описаны выше.
2. Применение объемного (трехслойного) кристалла. Уменьшение размеров кристалла увеличивает выход годных пластин.
3. Для межядерных обменов использована матричная схема соединений, использующая кремниевые лазеры и оптические линии передачи данных, обеспечивающие существенно большую скорость при меньших энергозатратах, чем электрические проводники. Для обмена используются протоколы интернет.
4. Одним из наиболее существенных преимуществ платформы является ее "сборная конструкция". По сути Intel удалось создать процессор, для которого **не важно какой вычислительный движок заключен в каждом из ядер**. Это позволит в будущем, используя единую логическую систему, создавать сложнейшие вычислительные системы, которые будут собираться как "конструктор", для решения самых различных задач на базе одной платформы.

Трехслойный кристалл может выглядеть следующим образом (рисунок взят из (www.research.ibm.com/photonics):



Структура кристалла представляет собой своеобразный "сэндвич". Память размещена в нижней части чипа и вертикально связанна с находящимся сверху ядром. Каждому ядру полагается по 64 Мбайт ОЗУ. В прототипе общий объем памяти составил 5 Гбайт, но Intel была ограничена техническими требованиями и определенным количеством транзисторов. Частично эта структура реализована в процессоре Polaris.

7.5. Процессор Polaris на 80 ядер.

Экспериментальный микропроцессор Polaris является результатом выполнения компанией Intel программы Tera-Scale, в которой ставились следующие задачи:

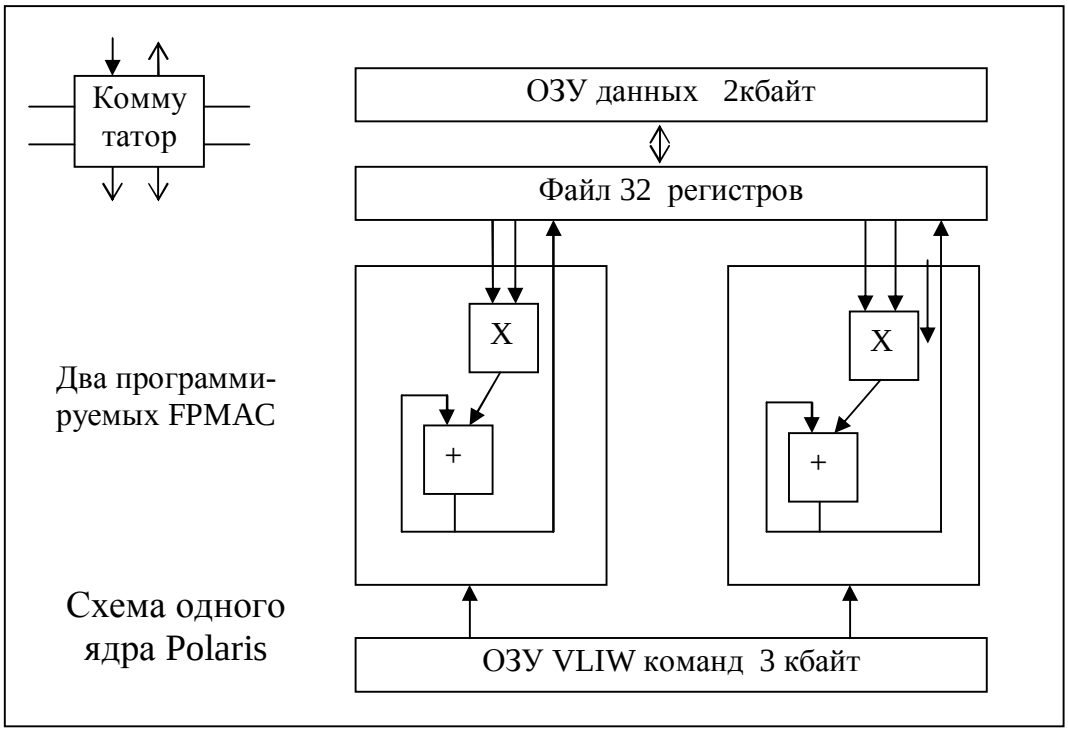
- TERA Operation Per Secord (10^{12} операций в секунду).
- Terabit Data Transfer (Обмен данными при терабитовых скоростях).
- TERA I/O (Терабитовая скорость операций ввода/вывода).

Polaris является шагом в направлении создания микропроцессоров для реализации «облачных» вычислений.

Общие характеристики Polaris:

- 80 однопоточных ядер, любое ядро можно отключать при разгрузке.
- Технологии с типоразмером 65 нм.
- Производительность – 1.01 Tflops.
- 100 млн. транзисторов (Nehalem – 2.3 млрд).
- Рассеиваемая мощность 62 Вт.
- Матрица процессоров 8 на 10 – 275, площадь ядра - 3 кв.мм
- Частота – 3.16 ГГц
- Напряжении ядер 0.95 В
- Матрица обменов данными использует сетевые протоколы, поэтому отказы отдельных узлов не выключают весь кристалл.

Микроархитектура ядра. Ядро содержит вычислительный блок и и 5-портовый коммутатор, связывающий его с четырьмя соседями.

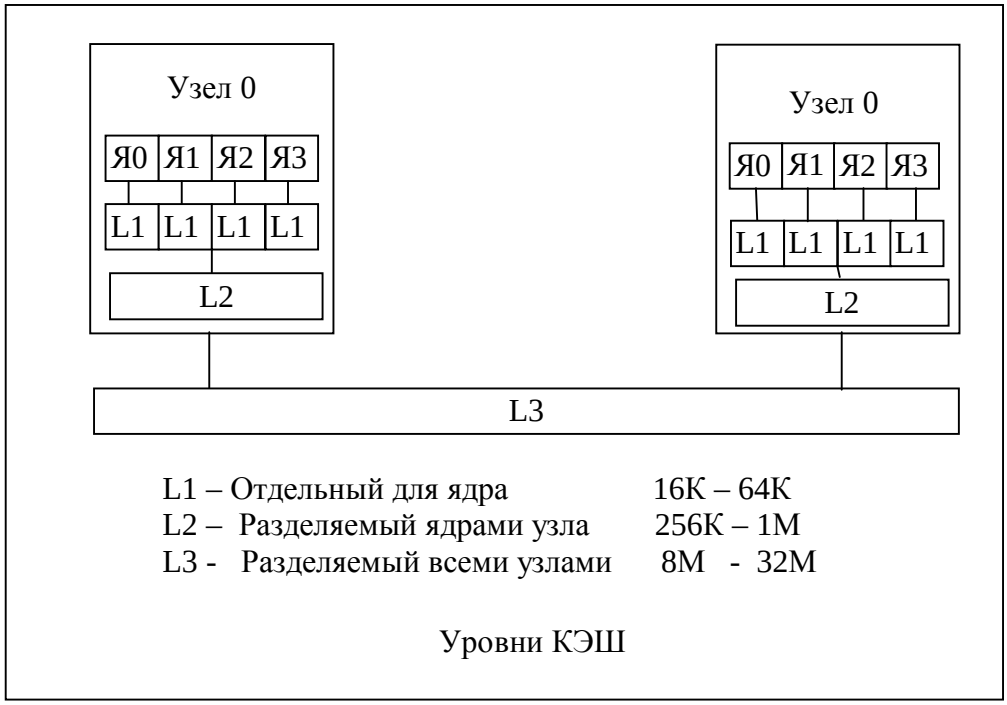


Ядро содержит однотактные память данных (2 Кбайт) и команд (3 Кбайт), файл регистров емкостью 32 строки (десять портов чтения плюс четыре порта записи) и два 32-разрядных FPMAC-устройства с плавающей запятой. 9-стадийные конвейеры могут выполнять команды «умножить-и-сложить», это дает четыре результата с плавающей запятой за такт на процессорный элемент, каждый из которых использует VLIW длиной 96 бит, что обеспечивает возможность выполнения до восьми операций за такт.

В целях максимального упрощения ядер они построены на основе 96-разрядной архитектуры VLIW (Very Long Instruction Word - "очень длинного командного слова "). При этом в одной команде VLIW может содержаться до восьми простых операций.

Архитектура кристалла.

Память. Каждое ядро имеет отдельную память. Для каждого узла (4 ядра) имеется отдельная память. Наконец, не менее значимой инновацией является микросхема статической памяти L3 (SRAM) объемом 20 Мбайт, пакетированная с процессором и размещенная на одном кристалле с ним.



Пакетирование в процессоре позволяет создать тысячи межкомпонентных соединений и обеспечивает полосу пропускания канала между памятью и ядрами шириной более 1 Тбайт/с.

В процессорах на платформе Tera-scale предусмотрено использование трехуровневой системы кэширования. Кэши L1 и L2 будут непосредственно связаны с каждым потоком и объем L1 составит от 16 до 64 Кбайт. Кэш второго уровня будет размером 256 - 1024 Кбайт. Общий для всего узла станет кэш третьего уровня объемом 8 - 32 Мбайт. Модули кэш-памяти будут размещаться

на той же подложке, что и вычислительные ядра, маршрутизаторы и прочие элементы.

Также инженеры Intel продемонстрировали модель работы **нового L4-кэша высокой ёмкости, доступного для всех узлов процессора**, который будет размещаться между процессором и памятью (схема "бутерброд"). Есть также вариант размещения кэша четвертого уровня напротив процессора. У обеих схем есть свои преимущества и свои недостатки. В настоящее время эксперты компании тестируют оба варианта.

Коммутация. Основной целью проекта Intel Terascale было исследование возможности создания накристальных сетей и управление энергопотреблением. Кристалл содержит решетку 8x10 процессорных ядер. Каналы между ядрами устойчивы к фазовым сдвигам их тактовых сигналов.

Канал между двумя ядрами поддерживает функционирование двух логических каналов (канал 0 и канал 1). Канал 0 обычно используется для передачи коротких сообщений, содержащих команды, а канал 1— для передачи длинных сообщений с данными. Это предотвращает возможность задержки передачи команд.

Встроенный в каждое из ядер пятипортовый маршрутизатор используется для обмена данными между ядрами, которые, тем самым, объединяются в сеть. При этом любое ядро может использоваться исключительно для передачи данных, что позволяет динамически отключать питание вычислительных модулей таких ядер и экономить электроэнергию. Как видно на слайде, четыре из пяти портов используются для связи с другими ядрами, а пятый - для подключения ко встроенной в "слоёный" чип памяти. В целях оптимизации охлаждения процессора сам многоядерный чип будет находиться сверху "бутерброда", а слои с памятью - под ним.

Программирование.

Intel® и другие производители процессоров в течение нескольких лет неустанно пропагандируют распространение параллельных вычислений.Согласно закону Мура, через 10 лет будут производиться процессоры с 128 ядрами, через 12 лет – с 256 ядрами, через 14 лет - с 512 ядрами, а к 2023 году число ядер в процессорах может превысить 1000.

К счастью, для создания качественно новых приложений для многоядерных систем вполне сгодятся проверенные временем и хорошо знакомые программистам средства разработки многопроцессорного ПО: OpenMP, MPI и Pthreads. Разработчики Intel также трудятся над подготовкой новых моделей программирования. Подробную информацию по разработке приложений для многоядерных систем можно найти на web-сайте сети Intel® Software Network.

Разработка ПО для терафлопных процессоров сопряжена и с другими трудностями. Прирост производительности приложения в системе на базе терафлопного процессора будет определяться долей последовательного программного кода (закон Амдала). Если программный код на 25% состоит из последова-

тельных сегментов, то, теоретически, на неограниченном числе ядер такое приложение будет работать в четыре раза быстрее.

Закон работает и в обратную сторону: для достижения максимальной производительности в четырёхъядерной системе достаточно распараллелить код на 75%. Чтобы достичь максимального прироста производительности в системе с 64 и более ядрами, необходимо распараллелить код на 99% и выше, что весьма затруднительно. Такого параллелизма нельзя достичь, распараллелив последовательный алгоритм овтоматически с помощью логических структур Pthreads или OpenMP. Здесь важно переосмыслить поставленную задачу и творчески подвести её решение под новый, эффективный параллельный алгоритм.

7.6. Процессор SCC на 48 ядер.

Перед экспериментальным процессором Polaris ставилась задача проверить возможность построения процессора с бальшим количеством ядер. Ядра Polaris были просто-напросто вычислители плавающей точки, не поддерживающие набор инструкций IA и, соответственно, не могущие работать как полноценный центральный процессор.

Intel представила свою экспериментальную разработку — 48−ядерный «одночиповый облачный компьютер» (Single-chip Cloud Computer, SCC) с архитектурой IA-32. Если его вместе со специализированной материнской платой установить в компьютер и поставить на него Windows, то система заработает. В этом и состоит его отличие от процессор Polaris.

Микрочип SCC, изготовленный по 45−нм-технологическому процессу, содержит 1,3 млрд транзисторов, частота пределах 1,6-1,8 ГГц, а его энергопотребление при максимальной нагрузке составляет всего 125 Вт — столько же потребляют две обыкновенных лампы накаливания.

SCC является наследником процессора Polaris, разработанного в рамках исследовательской программы Intel Tera-Scale, и предшественником 100 - ядерного коммерческого процессора. В отличие от своего прародителя SCC поддерживает стандартное программное обеспечение, созданное для архитектуры x86 — так, на демонстрации «облачного» компьютера были успешно запущены операционные системы семейств Windows и Linux. При этом каждое из 48 ядер является независимо программируемым, а каждое ядро может работать на своем напряжении и на своей частоте или быть полностью выключенным при отсутствии нагрузки.

«Облачность» *процессора* состоит в том, что все 48 ядер сообщаются между собой как сетевые узлы, поэтому отказ некоторых ядер не вызывает отказа всего кристалла.

Теперь демонстрируется уже не прототип 48-ядерного процессора, а реально работающий компьютер на его основе. Впрочем, экспериментальным компьютером дело явно не ограничится. В Intel говорят, что многоядерные концепции, которые заложены в 48-ядерном процессоре, позволяют использовать производные технологии в устройствах от серверов до мобильных телефонов.

7.7. Неоднородные многоядерные процессоры

На начальном этапе создание МЯП шло по принципу: каждое ядро есть полноценный процессор. Такой процессор был многофункциональным и включал:

- АЛУ с регистрами,
- оперативную память и кэши,
- устройство выборки команд и управления ветвлениями,
- управление периферией и сетевыми устройствами.

Естественно, для создания такого процессора требовалось много транзисторов и места на кристалле, увеличивалось энергопотребление. Это ограничивало количество ядер, а значит, и быстродействие.

Есть два факта, которые составляют основу использования высокопараллельных вычислительных архитектур:

- Потребляемая мощность процессора пропорциональна примерно квадрату тактовой частоты. Например, процессор с тактовой частотой 3 ГГц потребляет больше, чем 9 процессоров частотой 1 ГГц. Значит, чтобы не перегреть кристалл МЯП, предпочтительно повышать быстродействие МЯП за счет увеличения числа ядер, а не частоты.
- Если задача уже распараллелена на большое количество потоков, то тем больше вероятность, что задача может быть распараллелена на ещё большее количество потоков. Это задачи с массовым параллелизмом. К ним относятся вычислительные задачи. Это не обязательно решение математических уравнений, это может быть обработка изображений, проверка ключей, анализ строк. Но суть в том, что надо считать, считать и считать, а не обрабатывать ветвления, когда текущая ширина параллелизма резко сужается. Но, именно для вычислительных задач, большой объем ветвлений не характерен. Это задачи с массовым параллелизмом. Это позволяет использовать МЯП с большим и очень большим количеством ядер.

В вычислительных задачах обычно используется SIMD организация, позволяющая производить параллельную обработку вектора данных. При этом возможны две многоядерные аппаратные реализации этого процесса:

- Имеется n_1 процесоров, каждый из которых обрабатывает свою часть вектора. Здесь требуется n_1 ядер.
- Имеется один мультипроцессор, в состав которого входят n_2 АЛУ, каждое обрабатывает свою часть вектора под управлением мультипроцессра. Эти АЛУ также являются ядрами.

Известно, что для построения одного АЛУ требуется в несколько раз меньше транзисторов, чем для построения процессора. Поэтому на одинаковой площади кристалла можно разместить в несколько раз больше ядер, то есть $n_2 >> n_1$.

Если на кристалле имеется несколько мультипроцессоров, этот соответствует архитектуре multi-SIMD.

Наиболее ярким примером неоднородных МЯП являются графические процессоры компаний NVIDIA и AMD, которые содержат много ядер. После появления технологии CUDA такие ядра стали называться CUDA ядрами.
Рассмотрим пример. На примерно равных по размеру кристаллах размещены:

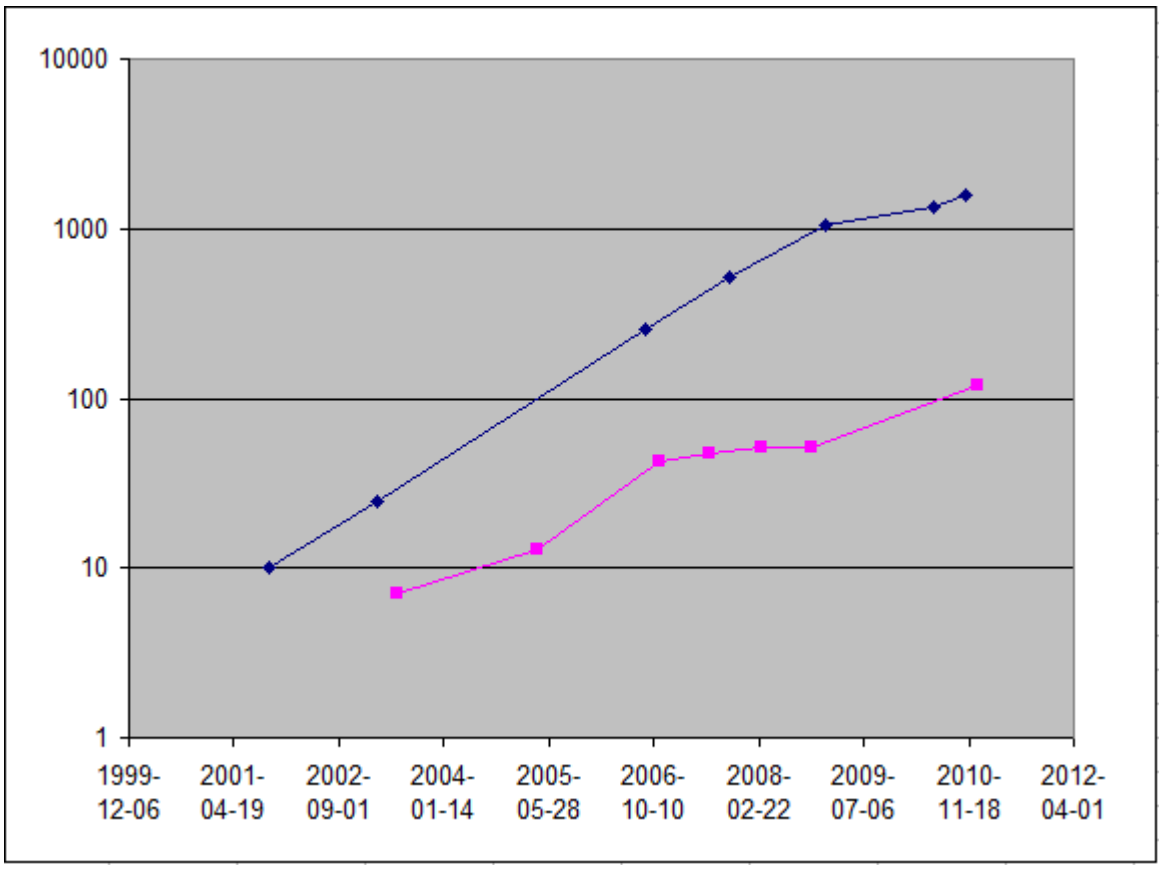
- Слева – кристалл CPU, на котором находятся 4 ядра (классические процессоры). Размеры полей рисунка пропорционально соответствуют их площади на кристалле.
- Справа – кристалл GPU, на котором находится 8 процессоров, под управление кождого из них находится 16 АЛУ. Значит, на кристалле находится 128 CUDA ядер.
-

Очевидно, что при равной частоте синхронизации на GPU будет выполнено значительно больше вычислительной работы.

Ниже предсавлена сравнительнаа таблица некоторых графических процессоров компании NVIDIA и AMD. Число АЛУ достигает нескольки тысяч.

	NVIDIA GTX 680	NVIDIA GTX 580	AMD Radeon HD 7970
Ядро	GK104	GF110	Tahiti
Количество транзисторов, млрд. шт	3.5	3.0	4.3
Техпроцесс, нм	28	40	28
Количество потоковых процессоров (АЛУ)	1536	512	2048
Частота ядра, МГц	1006	772	925
Шина памяти, бит	256	384	384
Объем памяти, Мбайт	2048	1536	3072
Частота памяти, МГц	6000	4008	5500
Мощность блока питания, вт	550	600	550

Увеличение быстродействия графических процессоров продолжится. Это видно из графика ниже.



На рисунке представлен сравнительный график роста быстродействия графических (синяя линия) и универсальных процессров (красная линия) по годам. По вертикали отложено быстродействие в Гфлопс/сек в логарифмическом масштабе. Для 2010 года быстродействие этих процессоров отличалось примерно на десятичный порядок.

На сегодня в списке TOP 500 в первой пятерке находится 3 компъютера, построенных на графических процессорах. Сейчас ведутся разработки с 1000 ядер на одном кристалле. Где предел? На этот вопросдан такой ответ:

Ключевым моментом роста является то, что наращивать количество «маленьких вычислителей» значительно проще, чем увеличивать тактовую частоту. Центральные процессоры не могут позволить себе такой рост именно из-за того, что не могут наращивать параллелизм такими же темпами. Поскольку они вынуждены хорошо исполнять традиционные приложения — OS, прикладные программы (Word, Excel к примеру) и они просто не могут себе позволить уменьшить вычислительное ядро. Они не могут деградировать производительность всех этих приложений.

Но не следует забывать, что графические процессоры в определенной степени все же являются специализированными процессорами, хотя область их применения довольно широка. Это - задачи с массовым параллелизмом, то есть задачи с большим объемом данных и прямолинейным, без условных переходов исполнением.

Глава 8. КВАНТОВЫЕ КОМПЬЮТЕРЫ.

8.1. Классические и квантовые компьютеры.

Как показано в главе 7 (Дъяконов), число R арифметико-логических операций, выполняемых за один такт, в классических ЭВМ имеет порядок

$$R = O(2^{t/2}),$$

где t – отрезок времени (в годах) от некоторого базового до настоящего времени. Для сравнения, как будет показано далее, в квантовом компьютере (КК) за один такт выполняется (предположительно):

$$R_{кк} = O\,(2^{\,n}) \text{ операций,}$$

где n – разрядность квантового регистра. Возьмем отрезок времени t = 60 лет. Предположим, что за это время будет создан КК на 1000 разрядов, при этом, пусть длительность такта КК составляет 1000 тактов классического процессора, тогда получаем:

$$R_{мяп}\,(t) = 2^{30} = 10^{10}$$
$$R_{кк}\,(n) = 2^{1000} = 10^{300}$$

Подсчитано, что для взлома ключа в системе шифрования RCA размерностью 320 бит надо выполнить 3.10^{94} операций. Следовательно, что доступно КК, то недоступно классическому компьютеру.

В главе 1 сказано, что до минимального размера порядка 10 нм транзистор сохраняет свои переключательные и усилительные свойства. В диапазоне размеров 5-0.5 нм наступает зона **мезоскопических структур** и приборов. В этой области разработан материал графен - плоский лист из атомов углерода толщиной в 1 атом. Из этого материала даже создан транзистор. Графен открвывает большие возможности для микроэлектроники, как с точки зрения быстродействия, так и нагрева. Но отдельный транзистор – это не процессор.

При размерах ниже 0.5 нм начинается зона **квантовых чипов.** Возможно, в этой области успех будет достигнут быстрее, чем в мезоскопической зоне.

Идея квантового компьютера была предложена в 1980 году советским ученым Ю.И.Маниным и в 1982 году - американским ученым Фейнманом. Законы квантовой механики, определяющие поведение таких квантовых битов (quantum bit) – **кубитов**, обеспечивают огромные преимущества квантового компьютера (скорость и параллелизм вычислений) по сравнению с классическим компьютером.

Переход от теории к практик начался с 1994 года, когда американский математик П.Шор предложил знаменитый квантовый алгоритм, который так и называется – «алгоритм Шора». Это алгоритм факторизации и дискретного логарифмирования. Эти задачи относятся к классу NP и алгоритм Шора позволяет

выполнять их за полиномиальное время, то есть появилась возможность взламывать ключи распространенной системы шифрования RSA, что и послужило сильным толчком для развития работ по квантовым компьютерам.

Работа квантового компьютера основана на краеугольном принципе квантовой механики – принципе суперпозиции Шредингера (1926 год):

> Если квантовая система может находиться в состояниях $|y_1\rangle, |y_2\rangle, \ldots, |y_n\rangle$, то она может находиться и в суперпозиции этих состояний.

Квантовым компьютерам посвящена монография Нильсена и Чанга [4].

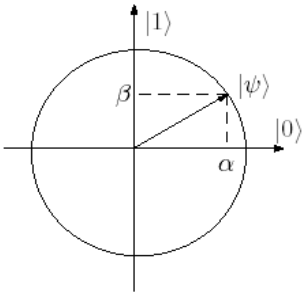
8.2. Квантовые биты.

На квантовом уровне существует много объектов, способных принимать состояния 0 и 1, которые используются в вычислительной технике для построения логических схем и блоков компьютеров. Например, электрон может вращаться в одну или другую сторону и его спин будет направлен вверх (1) или вниз (0). На самом деле положение спина электрона зависит от его магнитного момента, но приведенный пример двоичного элемента обычно используется из-за своей наглядности. Двум значениям кубита могут соответствовать, например, основное и возбужденное состояния атома, направление тока в сверхпроводящем кольце, два возможных положения электрона в полупроводнике и т.п.

Применительно системам с двумя базисными состояниями $|0\rangle$ и $|1\rangle$ (обозначения Дирака) это означает, что произвольное квантовое состояние $|y\rangle$ есть когерентная линейная суперпозиция базисных состояний:

$$|y\rangle = a|0\rangle + b|1\rangle$$

где комплексные числа a и b удовлетворяют условию нормировки $|a|^2 + |b|^2 = 1$. При этом система в таком суперпозиционном состоянии не находится ни в состоянии $|0\rangle$, ни в состоянии $|1\rangle$, а находится *одновременно* в этих *двух* состояниях с вероятностями согласно условию нормировки, что и показано на рисунке ниже:



Квантовый элемент с двумя состояниями называется **кубит** (квантовый бит), но в отличие от классического бита состояние кубита есть непрерывная величина, определяемая двумя числами и, в принципе, способная хранить бесконечное количество информации, как показано на рисунке ниже.

Фейнман по этому поводу кубита говорил «*Что в действительности означает* $a|0\rangle + b|1\rangle$*? Это Тайна. Мы не понимаем этого. Но мы можем сказать вам, как это работает*».

Состояние **кубита** можно узнать спомощью операции измерения, но при этом выполняетсяя следующая особенность.

> При измерении (классическая операция) кубит переходит в одно из классических состояний 0 или 1 в зависимости от значений a и b, предыдущее состояние теряется. Поэтому состояние кубита нельзя скопировать.

Обычно в векторно-матричной форме состояние кубита $|y\rangle = a|0\rangle + b|1\rangle$ представляют так:

$$a|1\rangle + b|0\rangle \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad a|0\rangle + b|1\rangle \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

8.3. Квантовый регистр.

Квантовым регистром (КР) назвается схема из кубитов, которые могут находится в тензорном (разомкнутом) или запутанном (зацепленном) состоянии. Рассмотрим квантовый выриант умножения векторов на примере двух кубитов. Если два кубита находятся в состояниях $|y_1\rangle$ и $|y_2\rangle$, то состояние двухразрядного регистра будет их произведением:

$$\mathbf{a} \otimes \mathbf{b} \rightarrow \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_1 b_1 & a_1 b_2 & a_1 b_3 \\ a_2 b_1 & a_2 b_2 & a_2 b_3 \\ a_3 b_1 & a_3 b_2 & a_3 b_3 \\ a_4 b_1 & a_4 b_2 & a_4 b_3 \end{bmatrix}.$$

В форме квантовых обозначений это будет выглядеть следующим образом:

$$|y_{12}\rangle = |y_1\rangle \otimes |y_2\rangle$$

$$(a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) = a_1a_2|00\rangle + a_1b_2|01\rangle + b_1a_2|10\rangle + b_1b_2|11\rangle \ ,$$

которое можно обозначить следующим образом:

$$|y_{12}\rangle = A|00\rangle + B|01\rangle + C|10\rangle + D|11\rangle .$$

Значит, при двух кубитах система имеет 4 состояния. Следовательно, система из N кубитов будет иметь 2^N состояний, например, при 100 кубитах число состояний равно $2^{100} \approx 10^{30}$.

Условием выполнения $|y_{12}\rangle$ является равенство $AD - BC = 0$. Когда это равенство выполняется, система находится в тензорном состоянии, в протином случае при $AD - BC \neq 0$ - кубиты регистра находятся в запутанном состоянии.

Тензорное состояние означает, что кубиты регистра не связаны: при изменении состояния одного кубита остальные не изменяются

В запутанном состоянии изменение состояния какого-либа кубита регистра изменяет состояние всех остальных кубитов, но при этом должно выполняться условие $\sum_{i=0}^{N-1} a_i^2 = 1$.

Запутаность между кубитами — это необходимое условие для работы квантового компьютера, это ключевой фактор, отвечающий за квантовый параллелизм и определяющий преимущество квантового компьютера над обычным. Состояния регистра никак не связано с состоянием отдельных кубитов. Вероятности a_i возникают и изменяются в процессе выполнения квантовых алгоритмов. Таким образом, в запутанном состоянии хранит 2^N n – разрядных слов. Изменение состояния одного кубита меняет состояние всех $2^N \times n$ битов. Это и есть *квантовый параллелизм* и прямо соответствует векторной архитектуре ОКМД.

Состоянии регистра можно измерить. Измерение - вероятностный процесс, который выполняется следующим образом:

1. Пусть $|y\rangle = (z_0, z_1, ..., z_{2^{n-1}})$ - квантовое состояние и $B = \{|0\rangle, ..., |2^n - 1\rangle\}$ - стандартный базис системы из n кубитов устойчивых состояний.
2. При измерении с вероятностью наибольшего $|z_i|^2$ состояние регистра становится $|y'\rangle = |i\rangle$. После измерения амплитуды всех остальных состояний становятся равными нулю.

Таким образом, прямой связи состояния регистра с состояниями входящих в нее кубитов нет. Вопрос о том, кто и как меняет состояние амплитуд регистра, чтобы увеличить значение $|Z_i|^2$ для некоторых из них, является сердцем любого квантового алгоритма. На сегодня известны следующие алгоритмы:

- **Алгоритме Гровера** – состоит в усилении нужного выходного значения путем применения диффузии и преобразования инверсии относительно среднего.
- **Алгоритм Шора** – состоит в нахождении общих свойств всех значений $f(x)$ таких, например, как периодичность функции.

Для таких систем флуктуации отдельных частей взаимосвязаны, но не посредством обычных классических взаимодействий, ограниченных, например, скоростью света, а посредством нелокальных квантовых корреляций, когда изменение одной части системы в тот же самый момент времени сказывается на остальных ее частях, даже разделенных в пространстве, что подтверждено экспериментами.

Организация квантового регистра. Квантовый регистр – это упорядоченное множество конечного числа кубитов (рис.8.1). На квантовом регистре и производятся вычисления в квантовом компьютере. Именно квантовый регистр, находящийся в запутанном состоянии, и создает экспоненциальное ускорение вычислений.

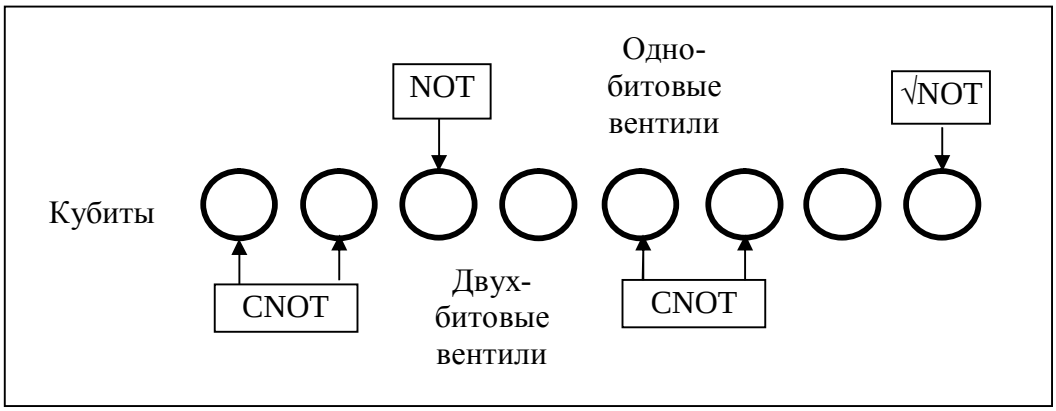


Рис. 8.1. Квантовый регистр. Одно- или двухкубитовые гейты осуществляют логические операции над кубитами или парами кубитов.

Физическая реализация Кане квантового регистра. В 1998 г. австралийским физиком Б.Кейном [24] было предложено использовать в качестве кубитов обладающие ядерным спином 1/2 донорные атомы с изотопами 31P, которые имплантируются в кремниевую структуру (рис.8.2). Это предложение, которое пока остается нереализованным, открывает потенциальную возможность создания квантовых вычислительных устройств с практически неограниченным числом кубитов.

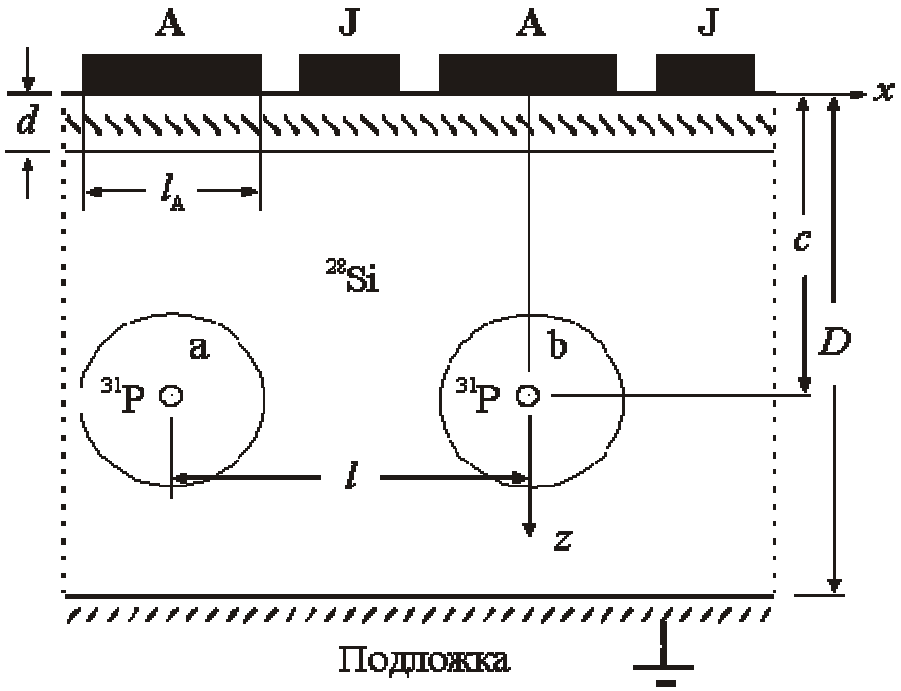


Рис.8.2. Схема двух ячеек регистра Кейна.

В рассматриваемом варианте предполагается использовать **температуры достаточно низкие** для того, чтобы электроны донорных атомов занимали только нижнее спиновое состояние в магнитном поле. Каждый донорный атом с ядерным спином - кубит в полупроводниковой структуре предполагается расположить регулярным образом с достаточной точностью под "своим" управляющим металлическим затвором (затвор A), отделенным от поверхности кремния тонким диэлектриком (например, окисью кремния толщиной порядка нескольких нанометров). Эти затворы образуют линейную периодическую решетку произвольной длины.

На схеме Кейна: l_A=10 нм, l =20 нм, c = 20 нм. Под наноэлектродами A в бесспиновом кремнии находятся одиночные неионизованные атомы 31P. Ядерные спины a, b выступают в качестве кубитов. Напряжения на электродах A управляют частотой магнитного резонанса ядерных спинов; с помощью напряжения на электроде J "включается" взаимодействие спинов, необходимое для выполнения операции CNOT.

С помощью электрического поля, создаваемого потенциалом затворов A, можно осуществлять индивидуальное управление квантовыми операциями путем селективного воздействия резонансных радиочастотных импульсов на ядерные спины определенных доноров. Величиной взаимодействия между ядерными спинами соседних доноров для выполнение двухкубитовых операций, предлагается управлять с помощью затворов J.

8.4.Квантовые гейты.

Квантовый гейт c n входами и n выходами – это преобразование, заданное на n кубитах и определяемое матрицей U размерности $2^n \times 2^n$, реализуемое внешними классическими средствами. При подаче радиочастотного импульса определенной величины и направления можно повернуть спин кубита на заданную величину, то есть изменить соотношение коэффициентов в выражении $|y\rangle = a|0\rangle + b|1\rangle$. Однокубитовые гейты во многом повторяют классическую систему логических элементов: AND, OR, XOR, NOT и др., но есть и специфические квантовые логические элементы.

$$\text{I:}\quad \begin{array}{l} |0\rangle \to |0\rangle \\ |1\rangle \to |1\rangle \end{array} \qquad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\text{X:}\quad \begin{array}{l} |0\rangle \to |1\rangle \\ |1\rangle \to |0\rangle \end{array} \qquad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\text{Y:}\quad \begin{array}{l} |0\rangle \to |1\rangle \\ |1\rangle \to -|0\rangle \end{array} \qquad \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

$$\text{Z:}\quad \begin{array}{l} |0\rangle \to |0\rangle \\ |1\rangle \to -|1\rangle \end{array} \qquad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Список наиболее распространенных элементов представлен выше в матричной форме. Они называются:

I – тождественное преобразование.

X – операцияи NOT.

Y – Y=ZX – комбинация последних двух.

Z – операция сдвига по фазе.

Тогда, например, операция NOT над 0 – ым состоянием кубита будет выполнена так:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$ то есть результат получен.

Преобразоване, которое применяет H к регистру из n кубит, называется преобразование Уолша – Адамара. Оно может быть определено рекурсивно следующим образом.

$$W_1 = H, W_{n+1} = H \oplus W_n$$

Для создания функционально полной системы логических элементов необходимы гейты с условным выполнением. Таким гейтом является двухкубитовый гейт Controlled NOT (CNOT или *XOR*), который действует на двух кубитах следующим образом. Он изменяет значение второго кубита, если значение первого кубита равно 1, и оставляет второй кубит неизменным в противном случае. Графическое изображение квантового гейта CNOT приведно на рис.8.3.

$$\begin{aligned} |00\rangle &\to |00\rangle \\ |01\rangle &\to |01\rangle \\ |10\rangle &\to |11\rangle \\ |11\rangle &\to |10\rangle \end{aligned} \quad \to \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

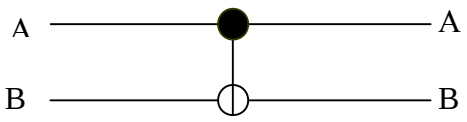
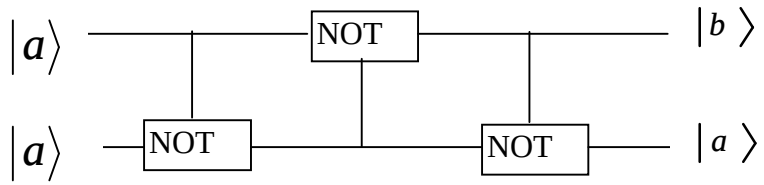


Рис.8.3. Графическое изображение гейта CNOT

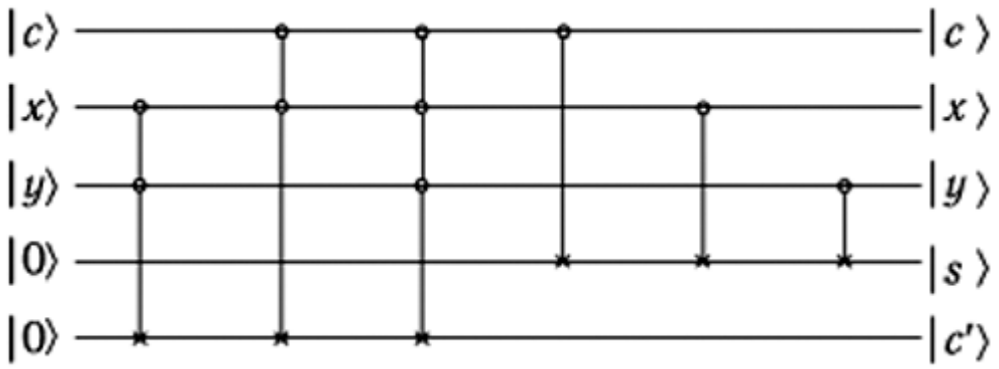
Квантовые схемы. Из многочисленных связок гейтов могут быть образованы произвольные квантовые цепи, например, представленная ниже:



Особенности квантовой схемы:

- Схему следует читать слева направо
- Горизонтальные линии представляют кубиты (состояния)
- Вертикальные линии обозначают суперпозицию
- В схеме нет обратных связей

Используя вентили Тоффоли можно построить любую классическую логическую схему. Например, квантовая схема, показанная ниже, позволяет производить одноразрядное сложение, используя гейты Тоффоли и CNOT.



На рисунке x и y обозначают биты данных, s - их сумму по модулю 2, c - входной разряд переноса, а c' - выходной разряд переноса.

8.5. Квантовый компьютер.

Принципиальная схема работы любого квантового компьютера (КК) представлена ниже [25].

Основной его частью является квантовый регистр - совокупность некоторого числа L кубитов. Этапы работы КК следующие:

- **Инициализация**. Все кубиты регистра должны быть приведены в базисные состояния.

- **Ввод данных.** Каждый кубит подвергается селективному воздействию, например, с помощью импульсов внешнего электромагнитного поля, управляемых классическим компьютером, которое переведет основные базисные состояния определенных кубитов в не основные состояния. При этом состояние всего регистра перейдет в суперпозицию базисных состояний вида $|n\rangle = |n1,n2,n3,...nL\rangle$, где ni = 0,1.
- **Эволюция**. В таком виде информация далее подвергается воздействию квантового процессора, выполняющего последовательность квантовых логических операций, включая изменение состояния запутанности, определяемую унитарным преобразованием, действующим на состояние всего регистра.
- **Измерение.** К моменту времени t в результате преобразований исходное квантовое состояние становится новой суперпозицией, которая и определяет результат преобразования информации на выходе компьютера. Чтобы произвести измерение, **вычисление должно приводить к результату, факторизованному по отдельным кубитам.** Только в этом случае можно проводить измерения (после вычисления) последовательно над каждым кубитом в отдельности, не портя квантовые состояния остальных. Причем в идеале каждый кубит должен находиться в одном из двух заранее заданных состояний (то есть фактически к концу вычисления должно появиться вполне определенное число в двоичной записи на кубитах, и никаких квантовых суперпозиций!). Именно это требование задает колоссальное ограничение на теоретически возможные квантовые алгоритмы. Рассмотрим порядок вычислений (рис.8.4).

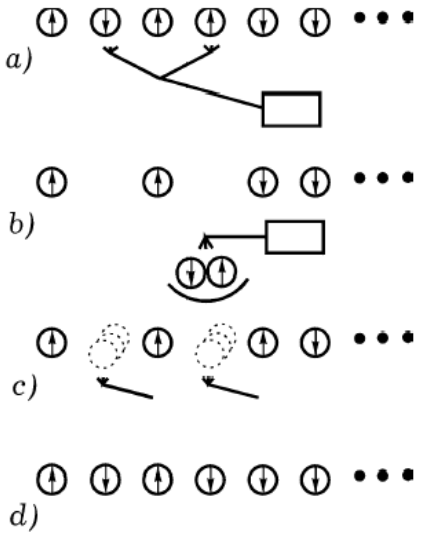


Рис. 8.4. Порядок вычислений

Первоначально все частицы имеют спины вниз. Затем выполняются следующие этапы.

1. Классическая машина (КМ) выбирает отдельные спины или пары спинов.
2. Над выбранными спинами КМ производит нужную одно или двухбитную операцию.

3. Скрещенные частицы возвращаются на свои первоначальные места. Эти три этапа повторяются многократно в соответствии с командами КМ до завершения этот цикл.
4. Производится измерении состояния частиц, поместив их предварительно в некоторую двоичную строку. Эта строка и есть результат вычислений.

Одной из самых сложных проблем построения квантовых компьютеров является проблема потери когерентности (декогернизация) во время вычислений из-зи влияния внешнего окружения. Разрабатываются специальные методы увеличения периода когерентности. Специалисты считают, что регистр Кейна может решит как проблему когерентности, так и проблему неограниченного количества кубитов.

Общие требования к квантовому компьютеру задаются “критериями ДиВинченцо”:

1. Возможность реализация кубитов и наращивания их числа. Эффективно работающий квантовый компьютер должен содержать, по крайней мере, 1000 кубитов.
2. Возможность приготовления начального состояния системы кубитов.
3. Низкая декогерентность системы кубитов в целом. По оценкам, за время декорентизации компьютер должен произвести не менее 10^4 вычислительных операций.
4. Существование квантовых гейтов, селективно воздействующих на кубиты, позволяющих, в том числе, контролировать связь (запутанность) между кубитами.
5. Возможность производить квантовые измерения состояний кубитов и получать результаты вычислений.

Совокупность всех возможных операций на входе данного компьютера, формирующих исходные состояния, а также осуществляющих унитарные локальные преобразования, соответствующие алгоритму вычисления, способы подавления потери когерентности квантовых состояний и исправления случайных ошибок, играют здесь ту же роль, что и " software " в классическом компьютере.

8.6. Алгоритм Гровера.

Ввиду запрета копирования и по другим причинам строить квантовые алгоритмы сложно, надо проявлять нестандартные приемы. Если просто повторять логические схемы классической логики (И, ИЛИ, НЕТ, сумматоры и др.) на квантовом уровне, то быстродействие КК будет лишь соизмеримым с классическим или хуже. Квантовые алгоритмы должны быть хитроумными. Их еще мало. При этом, было показано, что не для всякого алгоритма возможно «квантовое ускорение.

Алгоритм Гровера [26], предназначенный для поиска в неструктуированной базе данных, дает только квадратичное (полиномиальное) ускорение вычислений, а не экспоненциальное, как алгоритм Шора. Алгоритм Гровера выбран для

демонстрации квантовых вычислений из-за его относительной простоты по сравнению с алгоритмом Шора. Да и поиск в база данных является практически чрезвычайно важной проблемой.

Представим карту с большим количеством городов. Необходимо найти кратчайший путь через все эти города (задача коммивояжера). Простейший алгоритм заключается в переборе всех вариантов. Это потребует $O(N)$ операций. Квантовый алгоритм Гровера выполняет эту работу за $O(\sqrt{N})$ операций.

Другой пример — так называемая «универсальная задача перебора». Предположим, необходимо отыскать номер телефона, записанный произвольным образом на одном из 10 000 лежащих в аккуратной стопке листов. Чтобы найти нужный, возможно, потребуется последовательно пересмотреть всю стопку, то есть произвести 10 000 операций. Один из простейших квантовых алгоритмов — алгоритм американского математика Л.Гровера, предложенный в 1997 году, позволяет справиться с этим вопросом с гораздо меньшими затратами: нужное количество операций оказывается пропорционально всего лишь квадратному корню из числа возможных вариантов. Если вариантов 10 000, то потребуется 100 попыток.

Квантовый регистр (КР) из n кубитов может хранить $N = 2^n$ чисел. Это не зависит от того, находится ли регистр в тензорном или запутанном состоянии. Каждое число при чтении (измерении) может быть прочитано с вероятностью (амплитудой) a_i. При этом соблюдается $\sum_{i=1}^{N} a_i = 1$. В общем случае при большом n величина a_i очень мала, и отличить a_i от a_j не представляется возможным. Чтобы уверенно прочесть некоторое число, его амплитуда должна находиться в пределах от 1/2 до 1. Ядро алгоритма Гровера именно и составляет метод пошагового увеличения амплитуды отмеченного числа.

В дальнейшем изучении алгоритма Гровера не будут рассматриваться:

- Как устроена квантовая память.
- Как отмечаются нужные числа.
- Если рассматривать помеченное число как адрес, то где находится и как извлекается связанная с ним информация.
- Как реализуются физически гейты, используемые в алгоритме.

Эти сведения имеются у Нильсена и Чанга [4].

Основная операция для квантовых вычислений – гейт Адамара M:

$$M = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

который превращает состоянии бита 0 в суперпозицию двух состояний $(1/\sqrt{2}, 1/\sqrt{2})$, а состояние 1 трансформируется в $(1/\sqrt{2}, -1/\sqrt{2})$, то есть при равной

амплитуде фаза перевернута. В системе из ***n*** кубитов преобразование M можно осуществить покубитно, последовательно изменяя состояние системы.

Алгоритм Гровера. Пусть система имеет $N=2^n$ состояний, которые обозначаются $S_1,.....,S_N$. Эти 2^n состояний представляются как n – битные строки. Пусть существует состояние S_V, которое удовлетворяет условию $C(S_V)=1$, тогда, как для всех других состояний $C(S)=0$. Задача состоит в распознавании S_V Это поиск в базе данных, где функция $C(S)$ определена содержанием ячейки памяти, соответствующей состоянию S. Альтернативно, значение $C(S)$ может определяться компьютером. В такой форме можно представить различные важные задачи. Алгоритм состоит из следующих шагов:

A. На нулевое состояние действуем оператором Уолша-Адамара:

$$W|0,0,...,0\rangle = \frac{1}{\sqrt{n}}\sum_{i=0}^{n-1}|i\rangle;$$

В регистре из ***n*** кубитов преобразование **W** можно осуществить, покубитно применяя преобразование Адамара.

B. Этот пункт будет циклически повторяться определенное количество раз:

a) Контролируемое изменение фазы с контролем в виде *C* (имитация). В результате все состояния, кроме S_v, останутся без изменений, а у S_v фаза изменится на π, то есть поменяется знак.

b) Применим к полученному состоянию преобразование диффузии D.

C. Произвести измерение полученного состояния, которое удовлетворяет условию $C(S_V)=1$ с вероятностью, по крайней мере, не меньшей , чем 0.5.

Цикл шага **B** – суть алгоритма. Матрица преобразование диффузии D определяется следующим образом:

$$D_y = 2/N, \text{ если } i \neq j \text{ и } D_{ii} = -1 + 2/N$$

Преобразование D можно представить в форме $D=-I+2P$, где I – тождественная матрица (только 1 по диагонали), а P – проекционная матрица с $P=1/N$ для всех *i, j*.

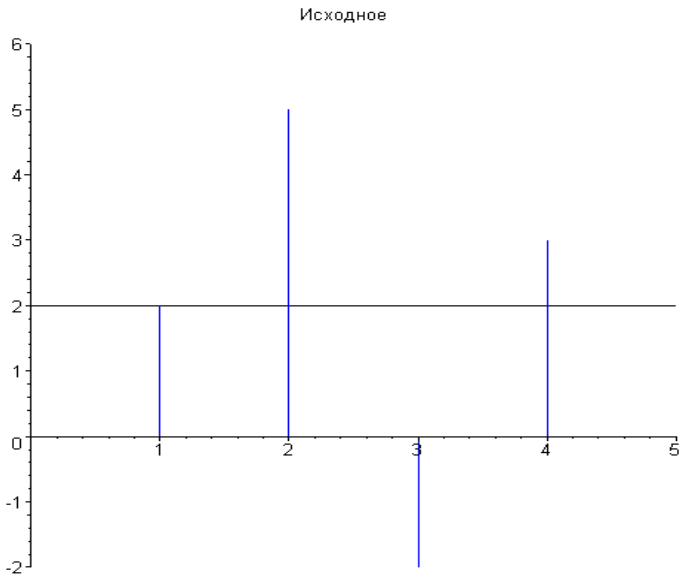
Матрица *P*, действуя на любой вектор V, дает вектор, каждая составляющая которого равна среднему по всем составляющим. Чтобы показать, что D – это инверсия относительно среднего, посмотрим, что случится, когда D действует на произвольный вектор $\bar{v}$.

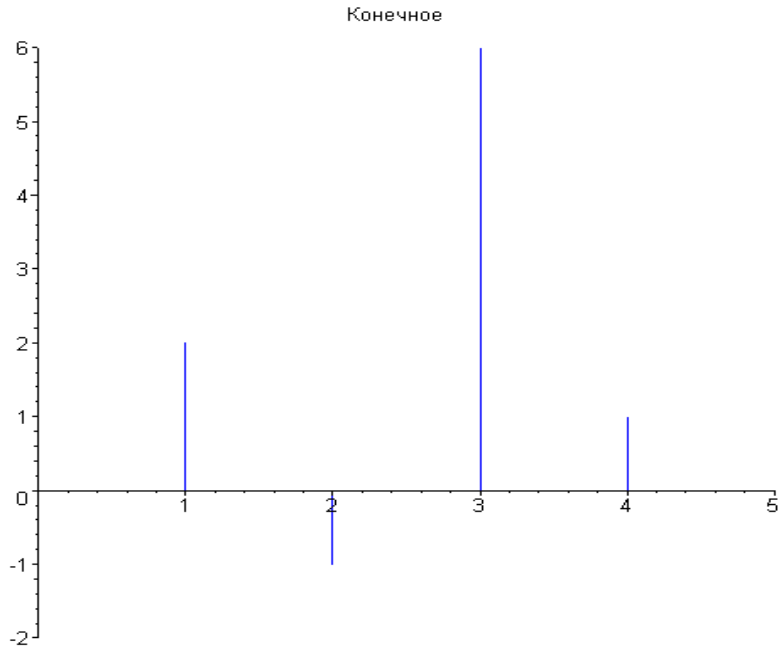
$$P=\frac{1}{8}\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Представляя $D=-I+2P$,получаем, что $D\bar{v}=(-I+2P)\bar{v}=-\bar{v}+2P\bar{v}$. Как установлено выше, каждая составляющая вектора $P\bar{v}$ есть A, где A есть среднее по всем составляющим вектора $\bar{v}$. Следовательно, *i*-ая составляющая вектора $D\bar{v}$ равна $(-\bar{v}+2A)$, что можно записать как $(A+(A-\bar{v}))$, что в точности есть инверсия относительно среднего.

Пусть есть некоторое состояние $\left|z=\sum_{|x\rangle} a_x|x\rangle\right\rangle$. Оператор P превратит его в вектор с компонентами, равными среднему значению $\left|A=\frac{1}{n}\sum_{|x\rangle} a_x|x\rangle\right\rangle$ (по одному на каждый кубит. Оператор диффузии производися над всеми кубитами одновременно и каждом кубите выполняется диффузия $D=-I+2P= P+(P-I)$.

На рисунках ниже, взятых из [27], представлен пример преобразования *D*. Амплитуды сотояний (вероятности) представлены всего для двух кубитов, при этом число состояний равно 4 ($N=2^2$). На осях координат для наглядности вместо дробных использованы целые числа, хотя сумма квадратов состояний равнв 1.





Каждый шаг (*B*) – это фазовое вращение. В его реализацию должна быть включена процедура распознавания состояния и последующего определения - осуществлять или нет поворот фазы. Это делается за один шаг и не включает классического измерения.

Каждая итерация этого цикла увеличивает амплитуду этого состояния на $O(1/\sqrt{N})$. В результате за $O(\sqrt{N})$ повторений цикла амплитуда, а, следовательно, и вероятность оказаться в желаемом состоянии b достигнет величины $O(1)$.

Заметим, что если провести большее число операций, то вероятность правильного ответа сначала будет уменьшаться, потом увеличиваться и далее таким же образом колебаться. Это означает, что в алгоритме надо правильно выбрать момент остановки. Алгоритм Гровера даёт выигрыш в числе операций при больших значениях *n*, ибо он требует только $O(\sqrt{N})$ шагов, в то время, как классический алгоритм требует *O*(*N*/2). Правда есть и проблемы. В квантовом алгоритме всегда есть ненулевая вероятность получить неверный результат. Впрочем, его легко проверить и, если требуется, запустить алгоритм ещё раз.

Квантовый алгоритм поиска требует только преобразование Уолша-Адамара и операция условного сдвига фазы, поэтому его будет проще реализовать по сравнению с другими квантовыми алгоритмами, такими, как «большое преобразование Фурье».

8.7. Алгоритм Шора.

8.7.1 Алглоритм создания открытого и секретных ключей в RSA

В криптографической системе с открытым ключом каждый участник располагает как открытым ключом (англ. *public key*), так и секретным ключом (англ. *secret key*). Каждый ключ — часть информации. В криптографической системе RSA каждый ключ состоит из пары целых чисел. Каждый участник создаёт

свой открытый и секретный ключ самостоятельно. Секретный ключ каждый из них держит в секрете, а открытые ключи можно сообщать кому угодно или даже публиковать их.

В основу криптографической системы с открытым ключом RSA положена задача умножения и разложения простых чисел на множители, которая является **вычислительно однонаправленной** задачей.

В качестве примера рассмотрим алгоритм создания открытого и закрытого ключей (автор LanG, habrahabr.ru/blogs/infosecurity/49634).

RSA – ключи генерируются следующим образом:

1. Выбираются случайные простые числа p и q, например, размером 1024 бита каждое.
2. Вычисляется их произведение $n = pq$, которое называется **модулем**.
3. Вычисляется значение функции Эйлера от числа n: $j(n) = (p-1)(q-1)$
4. Выбирается целое число e ($1 < e < j(n)$), взаимно простое со значением функции $j(n)$. В качестве e берут простые числа, содержащие небольшое количество единичных битов в двоичной записи, например, простые числа Ферма 17, 257, или 85537.
   - Число e называется **открытой экспонентой**
   - Время, необходимое для шифрования с использованием быстрого возведения в степень,пропорционально числу единичных битов в e
   - Слишком малые значения e могут ослабить безопасность RSA
5. Вычисляется число d, удовлетворяющее условию:
   $de \equiv 1 \pmod{\varphi(n)}$, или $de \equiv 1 + k\varphi(n)$, где k – некоторое число.
   - Число d называется **секретной экспонентой.**
   - Обычно, оно вычисляется при помощи расширенного алгоритма Эвклида.
6. Пара $P=(e, n)$ публикуется в **качестве открытого ключа** RSA.
7. Пара $P=(d, n)$ публикуется в **качестве секретного ключа** RSA.

Передача шифрованного сообщения от **A** к **B** производится так.

B выполняет:

- Взять открытый ключ (e,n) стороны **A**
- Взять открытый текст **M**
- Передать шифрованное сообщение $P_A(M) = M^e \bmod n$

A выполняет:

- Принять зашифрованное сообщение C
- Применить свой секретный ключ (d,n) для расшифровки сообщения: $S_A(C) = C^d \bmod n$

Пример.

Рассмотрим пример Шифрование и Дешифрование с помощью открытого и закрытого ключей на примере слова БЛОГИ. Создадим пару ключей.

1. p=3 q=11
2. N=33
3. F(p,q)=20
4. e =3
5. 7*3 mod 20 =1, d=7

Шифрование. Получаем цифровой эквивалент слова БЛОГИ с помощью вычисления их порядковых номеров в алфавите Цифровой эквивалент = 2(Б) 13(Л) 16(О) 4(Г) 10(И). Шифрование производится по формуле $y_i = x_i \bmod N$, где x_i цифровой эквивалент букве, остальные значения получены выше.

y1=27 mod 33=128 mod 33= 29
y2=137 mod 33=62748517 mod 33= 7
y3=167 mod 33=268435456 mod 33= 25
y4=47 mod 33=16384 mod 33= 16
y5=107 mod 33=10000000 mod 33= 10

Дешифрация. Расшифрование производится по формуле $x_i = y_i \, d \bmod N$. После небольших расчетов получаем:

x1=293 mod 33=24389 mod 33= 2
x2=73 mod 33=343 mod 33= 13
x3=253 mod 33=15625 mod 33= 16
x4=163 mod 33=4096 mod 33= 4
x5=103 mod 33=1000 mod 33= 10

Если сравнить x_i с порядковыми номерами букв мы получим слово БЛОГИ.
Таким образом, чтобы вычислить секретный ключ, нужно:

- Вычислить de (пункт 5), из которого легко определяется d.
- Но для этого нужно знать $j(n) = (p-1)(q-1)$, то есть знать p и q.
- Таким образом, начальной операцией определения *секретного ключа* является операция операция факторизации, то есть разложения некоторого большога числа на простые множители $n = pq$. Сам модуль n находится в открытом ключе участников шифропередачи.

8.7.2. Алгоритм факторизации Шора

Алгоритм факторизаци Шора состоит в определении простых множителей p и q для заданного числа $M = p \cdot q$ с использованием квантовой схемы для определения периода r некоторой функции вида:

$$y_M(x) = a^x \bmod M ,$$

где x = 0, 1, 2, …, $N = 2^L$, a – любое число, не имеющее общих делителей с M.Рассмотрим это на примере, взятом из [28]. Пусть M = 15. Выберем a = 2. В этом случае последовательность чисел a^x по модулю 15 представляется в следующем виде

x	0	1	2	3	4	5	6	7	8	...
a^x	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	...
Число	1	2	4	8	16	32	64	128	256	...
$a^x \, mod \, 15$	1	2	4	8	1	2	4	8	1	

Таким образом последовательность чисел $a^x \equiv 2^x$ по модулю 15 представляется в следующем виде: 1, 2, 4, 8, 1, 2, 4, 8,…, то есть имеет период по *x* равный r = 4 и удовлетворяет состояние $2^r \equiv 1 \bmod 15$. В общем случае $a^r = 1 \bmod M$, параметр *r* называется порядком функции $a^x \bmod M$, когда *a* □ *M* и не имеет общих множителей с *M*.

Если известен период *r*, множители числа *M* определяются с помощью классического алгоритма Евклида как наибольшие общие делители целых чисел $2^{r/2} \pm 1 \; и \; M$. В рассматриваемом примере $2^{4/2} \pm 1 = (5,3)$, то есть 15 = 5•3.

Таким образом, главное в алгоритме Шора - определение периода, затем факторизация производится легко.

Основной операцией по времени счета при определении секретного ключа является факторизацию больших чисел. За достаточно короткое время это можно сделать с помощью квантового компьютера на основе квантового алгоритма Шора. Разложив модуль *n* на простые множители, можно будет вычислить секретный показатель *d*.

8.7.3. Алгоритма Шора на квантовом компьютере

Разложения числа N на множители с помощью классических операций в простейшем случае можно сделать путем деления N на 2, 3, 4 и т. д. до $\sqrt{N}$. Число элементарных операций в этом случае будет $L^{L/2}$, то есть экспоненцально зависит от числа кубитов в регистре *L*. Для *L*= 1000 потребуется много лет работы суперсовременной ЭВМ.

В рассматриваемом далее алгоритме Шора, описание которого взято из [28], использующего квантовое дискретное преобразование Фурье, число операций зависит от *L* полиномиально (~ L^3), то есть имеет место экспоненциальный выигрыш по сравнению с классическими методами.

На сегодняшний день основой выполнения факторизации больших чисел является алгоритм определения периода некоторой функции. Алгоритм состоит в определении простых множителей *p* и *q* для заданного целого числа $M = p \cdot q$ путем использования квантовой схемы для определения периода *r* некоторой периодической функции вида *y*: *M (x)= ax mod M*, где *x* = =0,1,… N-1, N = 2^L, *a* – любое число, не имеющее общих делителей с рассматриваемым числом *M*.

Квантовый алгоритм Шора использует два квантовых регистра X и Y, первоначально находящихся в нулевом состоянии $|0\rangle$. В регистре X размещаются аргументы функции $y_m(x)$, то есть N состояний

$$|x\rangle = |x_{L-1},\ x_{L-2},...x_0\rangle \equiv |x_{L-1}\rangle \otimes |x_{L-2}\rangle \otimes ... \otimes |x_0\rangle .$$

Вспомогательный регистр Y используется для размещения значений самой функции $y_m(x)$ с подлежащим определению периодом r. Число состояний регистра $N = 2^L \geq M^2 \geq r^2$.

Первый этап расматриваемого алгоритма состоит в переводе начального значения $|0\rangle$ регистра X в в равновероятную суперпозицию всех состояний $N = 2^L|x\rangle = |x_{L-1}, x_{L-2}, ..., x_0\rangle$, путем применения операции Уолша_Адамара. Регистр Y не меняется. В результате для системы двух регистров X и Y получается состояние

$$|\Phi(x,0)\rangle = \sqrt{\frac{1}{N}}\sum_{x=0}^{N-1}|x\rangle \otimes |0\rangle = \sqrt{\frac{1}{N}}\sum_{x=0}^{N-1}|x,0\rangle$$

Если, например M = 15, данное состояние есть

$$|\varphi[x, y_m(x)]\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle \otimes |y_m(x)\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle \otimes |2^x mod 15\rangle =$$
$$= \frac{1}{\sqrt{N}}(|0\rangle \otimes |1\rangle + |1\rangle \otimes |2\rangle + |2\rangle \otimes |4\rangle + |3\rangle \otimes |8\rangle + |4\rangle \otimes |1\rangle + |5\rangle \otimes |2\rangle +$$
$$+ |6\rangle \otimes |4\rangle + |7\rangle \otimes |8\rangle + \ldots |N-1\rangle \otimes |2^{N-1} mod 15\rangle)$$

то есть последовательность функций $y_{15}(x)$ имеет период r = 4.

Каждому фиксированному состоянию второго регистра (Y) соответствует последовательность амплитуд, оставшихся в первом (X) регистре. Например, если зафиксировано состояние второго регистра(Y), то в первом регистре соответствующие числа отличаются на период r = 4.

$$|\varphi[x,y]\rangle = \frac{1}{\sqrt{N}}(|2\rangle + |6\rangle + |10\rangle + \cdots + |4A+\ell\rangle) \otimes |4\rangle = \frac{1}{\sqrt{A+\ell}}\sum_{j=0}^{A}|rj+\ell\rangle \otimes |4\rangle$$

где $0 \leq l \leq r < M$; A = целая часть $\left[\frac{N}{2} - 1\right]$. В рассматривемом случае l = 2 - начальное значение (определяемое выбором фиксированного значения состояния второго регистра). Таким образом, второй регистр служит для приготовления периодического состояния в первом регистре.

На втором этапе выделения периода r над состояние первого регистра производится операция преобразования Фурье. Для простоты пусть N точно делится на r, так что $A=N/2 -1$. В этом случае преобразование Фурье есть:

$$QFT_N:\ \sqrt{\frac{r}{N}}\sum_{j=0}^{A}|rj+\ell\rangle \Rightarrow \sum_{k=0}^{N-1} f_\ell(k)\,|k\rangle$$

где

$$f_\ell(k) = (\sqrt{r}/N)\sum_{j=0}^{A}\exp\left(\frac{2\pi i(jr+\ell)}{N}\,k\right)$$

Вероятность получить состояние $|k\rangle$ определяется выражением:

$$p(k) = |f_\ell(k)|^2 = \left(\frac{r}{N^2}\right)\left|\sum_{j=0}^{A}\exp(2\pi ijrk/N)\right|^2$$

которое как видно не зависит от *l*. Так как основной вклад в p (*k*) дают слагаемые, у которых *rk/N* близко к целому числу, точнее

$$-\frac{r}{2} \leqslant rk \mod N \leqslant r/2.$$

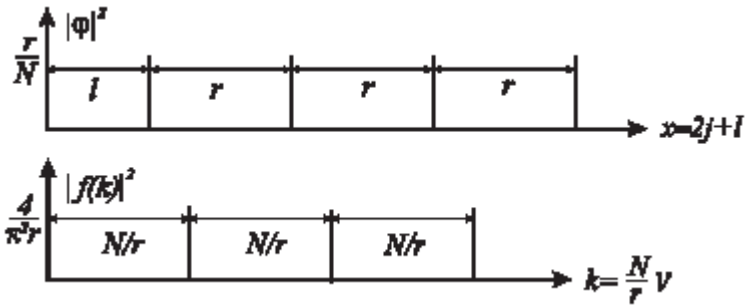
В случае малых значений *r/N* для каждого *r*, можно получить оценку для вероятности в виде

$$p(k) \geqslant 4/(\pi^2 r)$$

Отсюда следует, что по крайней мере с вероятностью $4/p^2 \approx 0{,}405$ измеренное значение принимает дискретные значения

$$k = \frac{N}{2}\nu,\ \text{где}\ \nu = 0, 1, \ldots r-1$$

То есть в результате квантового преобразования Фурье исходная суперпозиция для *M* = 15 преобразуется в равновероятную суперпозицию с периодом *N/r*.



Измерение вероятности по рисунку позволяет определить значение $k = \nu\frac{N}{r}$, имея которые при известном *k/N*, можно найти отношение *v/r*. Если *v* и *r* не имеют общих множителей, можно определить период *r* путем преобразования отношения *v/r* к виду, когда числительи знаменатель не имеют общих наибольших делителей. После этого с помощью алгоритма Эвклида легко найти и множители числа *M*.

1.8. Некоторые результаты по квантовым компьютерам

Общий уровень практических достижений представлен ниже в таблице [29]:

Тип оборудования	Число кубитов	Состояние на сегодня
Квантовый элемент И-НЕ	2	Продемонстрирован
Комплекс логических элементов	2	Продемонстрирован
Алгоритм Дойча	2	Продемонстрирован
Квантовое моделирование	Несколько	Простые эксперименты
Алгоритм Гровера	3+	Продемонстрирован с ЯРМ
Линия телепортации	3	Продемонстрирована
Алгоритм Шора	16+	Будущее
Кванновая машина факторизации	сотни	Будущее
Квантовый компьютер	Тысячи+	Будущее

На сегодня в интернет имеются сведения о разработанном компанией D-Wave Systems квантовом компьютере Orion на 16 кубитов и работы по наращиванию числа кубитов продолжаются. Однако, полных сведений по этому направлению нет.

В Республике Беларусь интенсивные работы по по квантовым вычислениям проводятся в АН РБ и некоторых вузах [30].

8.9. Словать терминов к главе 8

Вектор состояния — полное описание замкнутой системы в выбранном базисе. Задается лучом гильбертова пространства.

Волновая функция (волновой вектор) — частный случай вектора состояния, одно из координатных его представлений, когда в качестве базиса выбираются пространственно-временные координаты.

Гильбертово пространство (пространство состояний) — совокупность всех потенциально возможных состояний системы.

Декогеренция — физический процесс, при котором нарушается нелокальность и уменьшается квантовая запутанность между составными частями системы в результате ее взаимодействия с окружением. При этом подсистемы «проявляются» из нелокального состояния в виде отдельных самостоятельных элементов реальности, они обосабливаются, отделяются друг от друга, приобретая видимые локальные формы.

Запутанность — см. несепарабельность.

Квантовая система — это словосочетание указывает не на размер системы (микроуровень), а на способ описания: на то, что система описывается методами квантовой теории в терминах состояний.

Квантовая теория — это описание любой системы в терминах состояний, независимо от того, велика система или мала. Такое описание является на данный момент наиболее полным из всех других известных описаний физической

реальности, поэтому выводы, полученные квантовой теорией, имеют фундаментальное значение и формируют современную концепцию естествознания в целом.

Когерентные состояния (когерентная суперпозиция) — суперпозиция чистых состояний, то есть «наложение друг на друга» отдельных состояний, в которых может находиться замкнутая система. Когерентность означает согласованность поведения отдельных составных частей системы посредством нелокальных корреляций между ними.

Кубит (квантовый бит) — единица квантовой информации. В отличие от бита (единицы классической информации), который принимает только два возможных значения (0 и 1), квантовый бит может находиться в суперпозиции этих состояний.

Матрица плотности — матрица (таблица элементов), при помощи которой можно описывать как чистые состояния (замкнутые системы), так и смешанные, то есть открытые системы, взаимодействующие со своим окружением.

Принцип суперпозиции состояний — если система может находиться в различных состояниях, то она может находиться в состояниях, которые получаются одновременным «наложением» двух или более состояний из этого набора.

Смешанное состояние (открытая система) — такое состояние системы, которое не может быть описано одним вектором состояния, а может быть формализовано только матрицей плотности.

Состояние системы — реализация при данных условиях отдельных потенциальных возможностей системы. Характеризуется набором величин, которые могут быть измерены наблюдателем, в том числе в результате самонаблюдения (самовоздействия). Задается вектором состояния или матрицей плотности.

Спин — внутренняя характеристика частицы, не связанная с ее движением в пространстве и не имеющая классического аналога. Иногда, для наглядности, спин представляют в виде «быстро вращающегося волчка», что не совсем корректно. Для частиц со спином 1/2 пространство состояний является двумерным, и в качестве базисных состояний принято выбирать спин-вверх и спин-вниз.

Чистое состояние (замкнутая система) — такое состояние системы, которое может быть описано одним вектором состояния.

Приложение 1. СуперЭВМ семейства «СКИФ» Ряда 4.

Настоящий материал является краткой выборкой из работы: С.М. Абрамов, В.Ф. Заднепровский, А.Б. Шмелев, А.А. Московский. СуперЭВМ Ряда 4 семейства СКИФ. www.ict.edu.ru/vconf/files/11858.pdf

Опытные образцы Ряда 4 суперЭВМ семейства «СКИФ» запланированы к разработке в 2008–2012 гг. Данные суперЭВМ будут иметь производительность 500–5 000 Tflops (0.5–5 Pflops) и выше. Основываясь на прогнозах и планах ведущих компаний, мы предусматриваем выпуск четырех последовательностей моделей в рамках Ряда 4: СКИФ 4.N, СКИФ 4.W, СКИФ 4.S, СКИФ4.D.

Последовательности моделей суперЭВМ срок выпуска	Шасси	Шкаф	Система минимальная	Система средняя	Система максимальная
	Производительность, электропотребление (пиковые)		Пиковая производительность, размер системы		
СКИФ 4.N 3 кв. 2009	3 Tflops, 10.6 KW	24 Tflops 85 KW	48 Tflops, 2 шкафа	0.5 Pflops, 21 шкаф	0.77 Pflops 32 шкафа
СКИФ 4.W 3 кв. 2010	4.5 Tflops, 10.6 KW	36 Tflops, 85 KW	72 Tflops, 2 шкафа	1.0 Pflops 28 шкафов	1.1 Pflops 32 шкафа
СКИФ 4.S 1 кв. 2012	9 Tflops, 10.6 KW	72 Tflops, 85 KW	144 Tflops, 2 шкафа	2.0 Pflops 28 шкафов	2.3 Pflops 32 шкафа
СКИФ 4.D 2 кв. 2012	15 Tflops, 16.2 KW	120 Tflops, 130 KW	240 Tflops, 2 шкафа	7.7 Pflops 64 шкафа	10 Pflops 84 шкафа

Каждая последовательность моделей охватывает широкий спектр производительности от нескольких Tflops до 1000 (несколько тысяч) Tflops и предусматривает доступность для потребителя трех видов изделий:

- **Персональная суперЭВМ**. Вычислитель представляет собой одно *шасси*, которое можно расположить на рабочем месте сотрудника, тем более что это изделие бесшумное и имеет вполне приемлемое (для рабочего места) электропотребление. Пиковая производительность такого вычислителя может быть от трех до 15 Tflops. Заметим, что вся коммутация вычислительных узлов системной и вспомогательной сети уже выполнена в рамках шасси. Шасси является первым уровнем законченного изделия и строительным блоком для более крупных систем (шкаф, система из нескольких шкафов).
- **СуперЭВМ для лабораторий** (конструкторских отделов и т. п.) представляет собой один *шкаф*, содержащий от двух до восьми шасси и всю необходимую соединительную инфраструктуру для них: соединения системной сети, вспомогательной сети, сервисной сети, подсистем электропитания и охлаждения. Пиковая производительность такого вычислителя может быть от шести до 120 Tflops. Шкаф является бесшумным законченным изделием, а также строительным блоком для систем из нескольких шкафов.

- **Суперкомпьютерная система** *для крупных суперкомпьютерных центров* представляет собой несколько (2–32 и более) шкафов, объединенных общей инфраструктурой: соединения системной сети, вспомогательной сети, сервисной сети, подсистем электропитания и охлаждения. Пиковая производительность такого вычислителя может быть от 48 Tflops до 10 Pflops.

Таким образом, суперкомпьютеры ряда 4 семейства «СКИФ» охватывают большое разнообразие областей применения и широкий диапазон производительности.

Приложение 2. Графические вычисления.

2.1. Введение.

Видеокарта, графический процессор (англ. *graphics processing unit, GPU*) - отдельное устройство персонального компьютера или игровой приставки, выполняющее графический рендеринг (процесс создания изображения).

В 1961 году программист С. Рассел возглавил проект по созданию первой компьютерной игры с графикой «Космические войны». В 1968 году группой под руководством Н. Н. Константинова была создана компьютерная математическая модель движения кошки – мультфильм «Кошечка».

Разработки в области компьютерной графики сначала двигались лишь академическим интересом и шли в научных учреждениях. Постепенно компьютерная графика прочно вошла в повседневную жизнь, стало возможным вести коммерчески успешные проекты в этой области [1]. К основным сферам применения технологий компьютерной графики относятся:

- Графический интерфейс пользователя.
- Цифровая кинематография, телевидение.
- Цифровая фотография и живопись.
- Компьютерные игры.
- Тренажёры (управления самолётом, танком).
- Системы автоматизированного проектирования.
- Компьютерная томография.

Для решения этих задач видеокристалл использует множество простых вычислительных устойств – АЛУ, которые работают параллельно. Ясно, что возникает желание использовать этот параллелизм для неграфических вычислений. Тем более, что задач с большими массивами, элементы которых можно выполнять параллельно, оказалось множество кроме вышеуказанных. И главное, эти GPU есть здесь и сейчас. Теперь важно разработать удобную систему программирования.

2. 2. Создание графического объекта

GPU имеют нетрадиционную архитектуру, исторически определяемую методами обработки графической информации. Для получения трёхмерного изображения на плоскости нужны следующие шаги [1]:

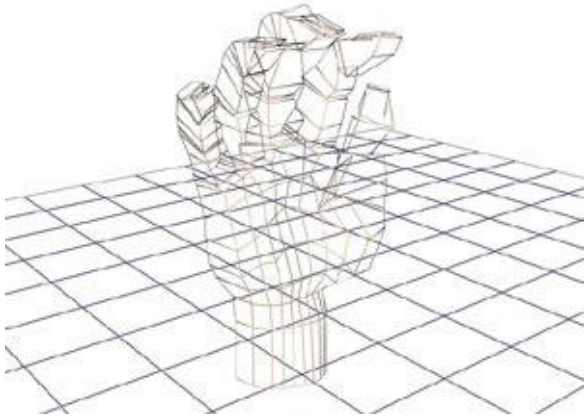
- Моделирование — создание трёхмерной математической модели сцены и объектов в ней. Выполняет CPU.

- Рендеринг (визуализация) — построение проекции в соответствии с выбранной физической моделью. Выполняет GPU.
- Вывод полученного изображения на устройство вывода - дисплей или принтер.

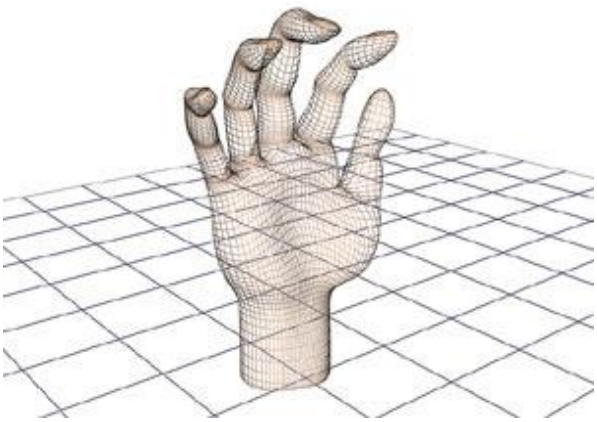
Рендеринг состоит в преобразовании 3D объекта в 2D кадр, при этом часть информации теряется, прежде всего, о глубине объекта. Чтобы сделать объект реалистичным используется ряд приемов. Для этого объекты проходят несколько стадий обработки. Самые важные стадии это создание формы (shape), обтягивание текстурами, освещение, создание перспективы, глубины резкости (depth of field) и сглаживания (anti-aliasing). Выполнение этих шагов без разделения функций между CPU и GPU приводится ниже.

Создание формы.

Для того чтобы составить достоверную картинку с кривыми линиями как в окружающем мире, приходится компоновать форму из множества мелких формочек (полигонов). Например, человеческое тело может потребовать тысячи этих формочек. Вместе они будут образовывать структуру, называемую каркасом.



На иллюстрации показан каркас руки, составленный из 862 полигонов



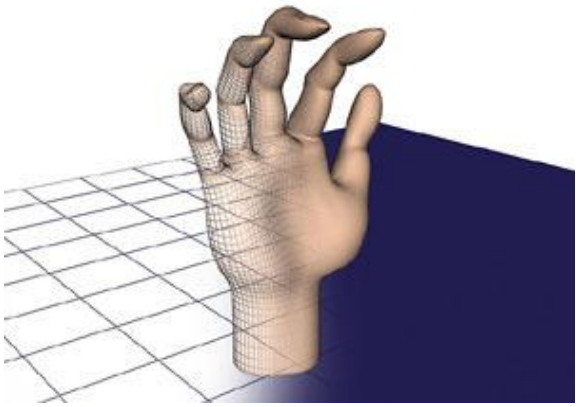
Качество выше, но это требует уже 3444 полигона

Создание поверхности каркаса

Информация о поверхности складывается из трех составляющих:

- Цвет: какого поверхность цвета? Однородно ли она окрашена?
- Текстура: ровная ли поверхность, есть вмятины, бугры, рихтовка?
- Отражающая способность: отражает ли свет? Четкость отражения ?

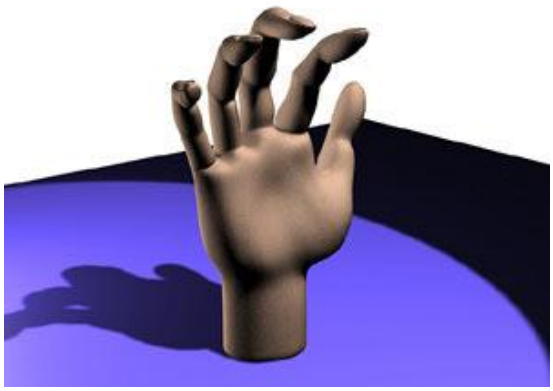
Придание "реальности" объекту состоит в подборе комбинации этих трех составляющих в различных частях изображения.



Добавление поверхности к каркасу улучшает изображение

Освещение.

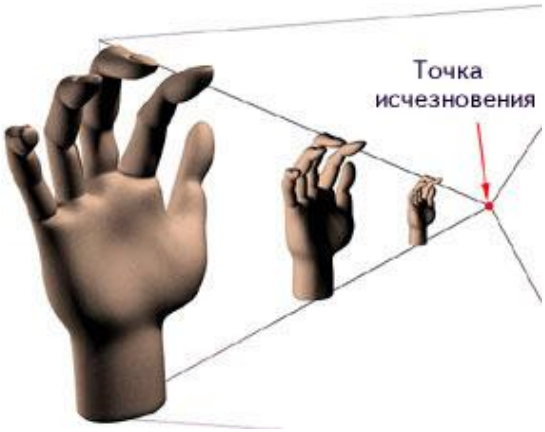
Освещение играет ключевую роль в двух эффектах, придающих ощущение веса и цельности объектам: затенения (shading) и тени (shadow). Первый эффект затенения заключается в изменении интенсивности освещения объекта от одной его стороны к другой. Благодаря затенению шар выглядит круглым, высокие скулы выпирают на лице, а одеяло кажется объемным и мягким. Эти различия в интенсивности света совместно с формой усиливают иллюзию, что объект кроме высоты и ширины имеет еще и глубину. Иллюзия веса создается вторым эффектом - тенью.



Подсветка добавляет глубину объекту через затенение, но и "привязывает" объект к земле посредством тени.

Перспектива.

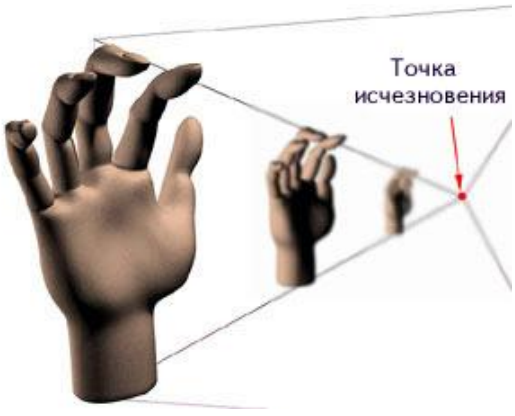
Если встать на обочину длинной прямой дороги и посмотреть вдаль, то вам покажется что правая и левая полоса дороги сходятся в точку на горизонте. Если все объекты на экране будут сходиться в одну точку, то это и будет называться перспективой.



На приведенной иллюстрации руки выглядят разделенными, но на большинстве сцен одни объекты находятся впереди и частично закрывают вид на другие объекты. Для таких сцен необходимо учитывать информацию, какие объекты закрывают другие и насколько сильно. Наиболее часто для этого используется Z-буфер (Z-Buffer). Z-буфер присваивает каждому полигону номер в зависимости от того, насколько близко к переднему краю сцены располагается объект, содержащий этот полигон. Обычно меньшие номера присваиваются ближашим к экрану полигонам. Объект с самым маленьким Z-значением будет полностью прорисовываться, другие же объекты с большими значениями будут прорисованы лишь частично.

Глубина резкости

По мере удаления объекта от наблюдателя будет потеря резкости. Это тоже на до реализовать.



Это только небольшое количество приемов, применяемых для повышения реалистичности рализуемых компьютером изображений. Все приведенные последовательно реализуемые действия над изображеним составляют графический конвейер, который реализуется программно-аппаратными средствами.

2.3. Графический конвейер

Графический конвейер представляет собой аппаратно-программное устройство, которое переводит объекты, описанные в трехмерном пространстве XYZ, с учетом положения наблюдателя, во множество пикселей на экране монитора.

Обработка объектов в GPU производится с помощью специальных программ - шейдеров, выполняемых внутри GPU. Любая видеокарта поддерживает несколько типов шейдеров:

- Вершинный шейдер оперирует расположением узлов пространственной сетки, которая формирует каркас 3D-модели. Как мы знаем, точка в 3D графике задается, как правило, набором из 4-х значений (x,y,z,w). Компонент w является масштабом
- Путем программирования вершинных шейдеров можно изменять расположение объекта в пространстве и рассчитывать эффекты его освещения.
- Пиксельные шейдеры позволяют изменить текстуру виртуальной кожи объекта, придавая ей соответствующую фактуру и цвет.

Геометрические шейдеры активируются при быстром приближении объекта к зрителю. добавляя изображению необходимые подробности для реализма.

Графический конвейер имеет следующие этапы:

Этап 1. Вначале видеопроцессор получает от CPU информацию об объектах, которые необходимо обработать и сцене:

- Описание объектов для визуализации, представленных в виде списка вершин для каждого объекта . Для каждой вершины заданы координты, нормали, цвет, текстура и др.
- Описание сцены, которое в простейшем случае влючает размеры сцены, расположение камеры (наблюдателя) и источников света.
- Расположение и поза объектов.

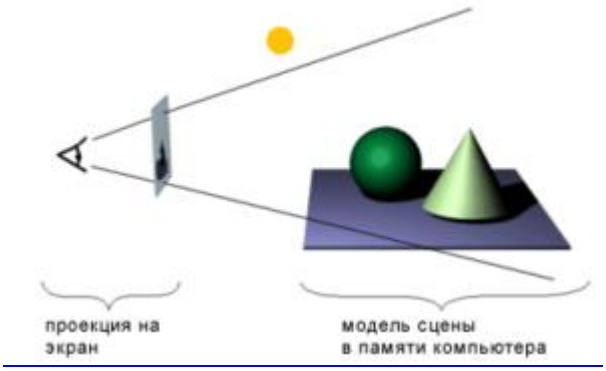


Схема проецирования сцены на экран компьютера

Этап 2. Затем вступает вершинный процессор ядра (ядер может быть много). Он на основании полученных данных строит конкретный объект в пространстве сцены с фиксированными координатами, называемый вершиной (vertice). На этом шаге графического конвейера все вершины объектов подвергаются аффинным преобразованиям - вращению, масштабированию и перемещению. За этот шаг отвечаетак же вершинный шейдер. Всеми визуальными преобразованиями в 3D-графике управляют матрицы (см. также: аффинное преобразование в линейной алгебре). В компьютерной графике используется три вида матриц:

- матрица поворота
- матрица сдвига
- матрица масштабирования

Здесь же происходит расчет освещенности в вершинах в зависимости от количества, места положения и типа источников света. Вершинный шейдер получает на вход одну вершину, содержащую координаты в локальной системе координат, и выдает ее же, но уже трансформированную в координатах системы наблюдателя (камеры).

Здесь вершины обрабатываются в вершинных процессорах в режиме MIMD (множество команд – множество данных). Другими словами, несколько вершинных шейдеров одновременно работают над разными объектами.

Этап 3. Следующая этап конвейера – сборка. На этом этапе вершины собираются в примитивы – треугольники (полигоны), линии или точки. *Подчеркнём, что ни о каком видимом объекте пока речь не идёт, это абстрактная информация о том, что вершины объединены в какой-то геометрический объект.*

Этап 4. Эта информация переходит дальше по конвейеру – в пиксельный процессор, который определяет конечные пиксели, которые будут выведены в кадровый буфер, и проводит над ними различные операции: затенение или освещение, текстурирование, присвоение цвета, данных о прозрачности и т.п. Над пикселями проводится Z-тестирование (выясняется глубина каждой точки, так как мы говорим о трёхмерном изображении), и линии сглаживаются (antialiasing). Вся эта информация передаётся на следующую стадию (Z-данные – в Z-буфер).

Z-буфер представляет собой двумерный массив, каждый элемент которого соответствует пикселу на экране. Когда видеокарта отрисовывает пиксел, его удалённость просчитывается и записывается в ячейку Z-буфера. Если пикселы двух рисуемых объектов перекрываются, то их значения глубины сравниваются, и рисуется тот, который ближе, а его значение удлённости сохраняется в буфер. Получаемое при этом графическое изображение носит название z-depth карта, представляющая собой полутоновое графическое изображение, каждый пиксел которого может принимать до 256 значений серого. По ним определяется удалённость от зрителя того или иного объекта трехмерной сцены. Карта широко применяется в постобработке для придания объёмности и реалистичности и создаёт такие эффекты, как глубина резкости, атмосферная дымка и т.д.

Также карта используется в 3D-пакетах для текстурирования, делая поверхность рельефной. Z-буфер очень эффективен и практически не имеет недостатков, если реализуется аппаратно.

Этап 5. Из Z-буфера вычитываются данные о расположении конкретных пикселей, чтобы отбросить те, которые будут скрыты другими объектами и не видны пользователю. Фрагменты снова собираются в полигоны, состоящие из отдельных пикселей, и весь массив уже отработанной картинки передаётся в кадровый буфер для последующего вывода на экран. Эта текстура обрабатывается в пиксельных процессорах которые по классификации многопрцессорных систем могут быть классифицированы как SIMD.

Вот такой процесс и называется конвейером.

2.4. Графический процессор.

Видеока́рта предназначена для преобразования графического образа, хранящегося в памяти компьютера, в форму, предназначенную для дальнейшего вывода на экран монитора. Видеокарта состоит из следующих основных частей:

- графический процессор (Graphics processing unit — графическое процессорное устройство) - занимается расчётами выводимого изображения;
- видеопамять - выполняет роль кадрового буфера, в котором хранится изображение, генерируемое и постоянно изменяемое графическим процессором и выводимое на экран монитора. Графические процессоры также используют в своей работе часть общей системной памяти компьютера;
- цифро-аналоговый преобразователь - служит для преобразования изображения в уровни интенсивности цвета, подаваемые на аналоговый монитор.

Основным элементом карты является графический процессор [2], который непосредственно и занимается формированием самого графического образа.

У графического процессора видеокарты работа простая и распараллеленная изначально. В соответствии с описанным выше графическим конвейером видеочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели. Обработка полигонов и пикселей независима, их можно обрабатвать параллельно, отдельно друг от друга. Поэтому, из-за изначально параллельной организации работы в GPU используется большое количество простых исполнительных блоков.

Сделаем качественную оценку количества этих блоков. Пусть сцена имеет размер 2.103×2.103 пикселей. Тогда получаем:

- Число пикселов сцены – 4.10^{6}
- Примерное число плавающих операций на обработку одного пиксела – 500
- Число тактов синхронизации на 1 Flop = 5
- Общее число однотактных операций -$5 \times 4.10^{6} \times 500 = 10^{10}$. Требуемое быстродействие для 50 кадров в сек - $10^{10} \times 50 = 5.10^{11}$ оп/сек
- Требуемое число АЛУ (SP) - $5^{11}/5.10^{9} = 100$

Вывод очевиден – для обеспечения компьютерной графики нужны параллельные вычисления с с большим числом вычислительных элементов.

Основной вычислительный элемент графического процессора - потоковый процессор (Streaming Processor – SP). SP проще CPU универсальных процессоров по следующим причинам:

- Обработка полигонов и пикселей независима, их можно обрабатывать параллельно, их легко загрузить, в отличие от распараллеливания последовательного потока команд для CPU. Поэтому не нужны большие количества транзисторов и площадь на буферы команд, аппаратное предсказание ветвления и огромные объёмы начиповой кэш-памяти.
- Видеочипы предназначены для параллельных вычислений с большим количеством арифметических операций. И значительно большее число транзисторов GPU работает по прямому назначению - обработке массивов данных.

Таким образом, потоковые процессоры по существу являются АЛУ, а не классическими процессорами.

Сделаем качественную оценку требуемого объема оборудования графического процессора. В таблице ниже показано количество транзисторов, необходимое дла построения различных узлов обычного компьютера (CPU). .

N	Структура	Число транзисторов
1	Сумматор (32)	30000
2	16 РОН	16000
3	Процессор (32)	300000

Из таблицы видно, что число транзисторов в процессоре и АЛУ отличается на порядок. На рисунке показано, что при тех же размерах кристалла, на нем можно поместить значительно больше SP, чем процесоров CPU. В CPU это место на кристалле тратится на память (Cache DRAM) и управление, а в GPU – преимущественно на АЛУ. Считается, что количество SP на кристалле графического процессора может составлять сотни и тысячи.

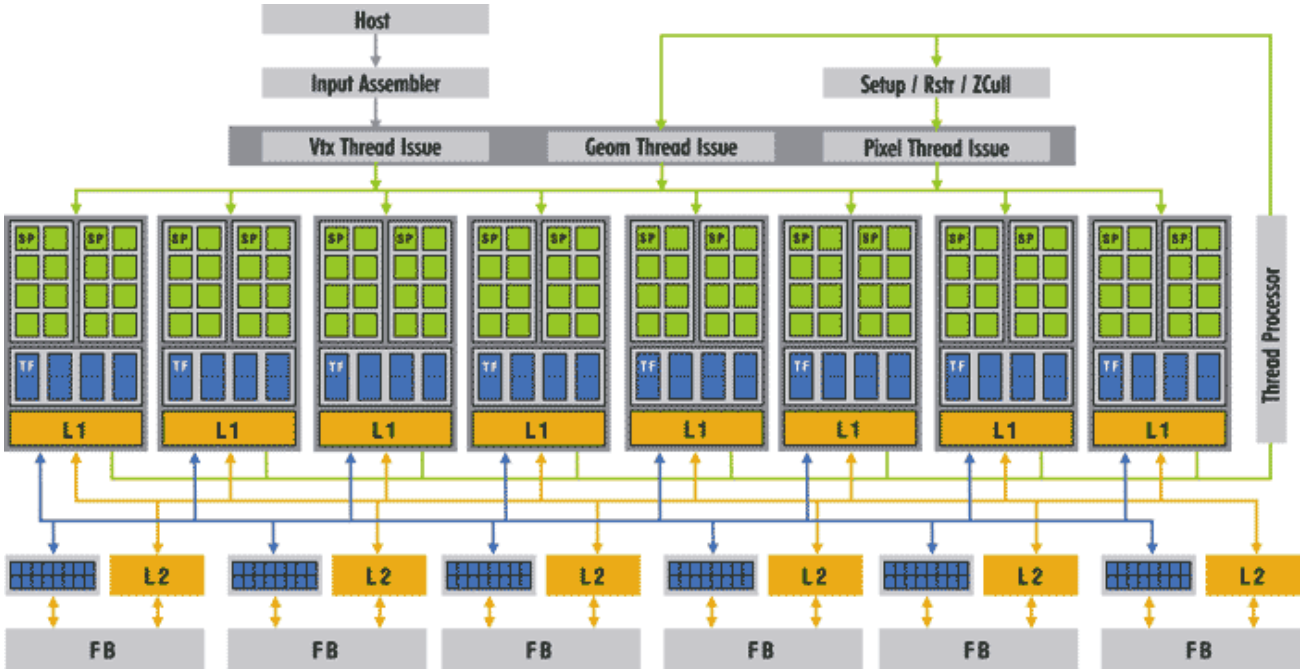
.

Для примера ниже представлена структурная схема конкретного GPU среднего класса NVIDIA GeForce 8800. Всего в ней 128 АЛУ, которые конструктив-

но объединены в в 8 мультипроцессоров (ядер), каждый из которых оснащен четырьмя текстурными модулями и общим L1-кэшем.

Каждое ядро представляет собой два шейдерных процессора (состоящих из восьми потоковых процессоров каждый), при этом все восемь блоков имеют доступ к любому из шести L2-кэшей и к любому из шести массивов регистров общего назначения.

На каждые четыре потоковых процессора приходится один текстурный блок, включающий один блок адресации текстур (Texture Address Unit, TA) и два блока фильтрации текстур (Texture Filtering Unit, TF).



Структурная схема графического процессора NVIDIA GeForce 8800

Источники информации

1. Онучин Алексей. Графическая система современного персонального компьютера и перспективы ее развития (есть в интернет).
2. Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. 2010 г. 232 с. (есть в интернет).

Приложение 3. Инструкции по компиляции и запуску многопоточных программ в домашних условиях.

Д.С. Глызин, ЯрГУ

Дата создания - 20.11.2010, обновлено 05.01.2011, 11.04.2011

I. Windows, Visual Studio 2008

Полная MS Visual Studio 2008 доступна для скачивания в сети 7-го корпуса после запроса в системе MSDN_AA.
Все остальные необходимые продукты распространяются свободно.

0. Установите MS Visual Studio 2008

1. OpenMP

- 1.1. В Студии создайте новый проект: Visual C++->Win32->Win32 Console Application. Введите название проекта, снимите галочку с Create directory for solution.
- 1.2. Нажав OK, в появившемся окне выберите Application settings и отметьте пункт empty project. Нажмите Finish.
- 1.3. Создайте в проекте новый hello.cpp файл, скопируйте в него текст примера.
- 1.4. В свойствах проекта выберите C/C++->Language->OpenMP Support->Yes (/openmp)
- 1.5. Скомпилируйте и запустите программу по Ctrl+F5

2. MPI

- Инструкция проверена на 32-битной версии.
- 2.1. С Сайта MPICH скачайте MPICH2 Windows (binary), подходящий для вашей системы
- 2.2. Установите mpich2 с настройками по умолчанию
- 2.3. В Студии создайте новый проект: Visual C++->Win32->Win32 Console Application. Введите название проекта, снимите галочку с Create directory for solution
- 2.4. Нажав OK, в появившемся окне выберите Application settings и отметьте пункт empty project. Нажмите Finish
- 2.5. Создайте в проекте новый hello.cpp файл, скопируйте в него текст примера
- 2.6. В свойствах проекта выберите C/C++->General->Additional Include directories и добавьте туда C:\Program Files\MPICH2\include
- 2.7. В свойствах проекта выберите Linker->General->Additional Library Directories и добавьте туда C:\Program Files\MPICH2\lib
- 2.8. В свойствах проекта выберите Linker->Input->Additional Dependencies и добавьте туда cxx.lib и mpi.lib
- 2.9. Скомпилируйте и соберите программу
- 2.10. Запустив программу из Студии, вы получите однопроцессорное приложение. Для запуска в несколько потоков запустите Пуск->Программы->MPICH2->wmpiexec.exe
- 2.11. В строке Application выберите ваш собранный в студии exe-файл, задайте требуемое количество процессов, отметьте галочку "run in an separate window" и нажмите кнопку Execute

- 2.12. В появившемся окне введите имя пользователя Windows и пароль, нажмите Register и затем OK. Если у вашей учетной записи нет пароля, предварительно создайте его.
- Если вы пользуетесь Windows 7 или Windows Vista, и в результате выполнения предыдущего пункта выводится ошибка " No smpd passphrase specified through the registry or .smpd file", нужно запустить от имени администратора консоль cmd, перейти в папку C:\Program Files (x86)\MPICH2\bin и выполнить команду smpd -phrase behappy -install.
- Если возникает непонятная ошибка, создайте в системе нового пользователя с паролем и правами администратора (только латинские буквы в имени и пароле). Зарегистрируйте этого пользователя с помощью wmpiregister

3. CUDA

- 3.1. С сайта NVIDIA скачайте Developer Drivers и CUDA Toolkit, подходящие для вашей системы
- 3.2. Установите драйверы и тулкит с настройками по умолчанию
- 3.3. В Студии создайте новый проект: Visual C++->Win32->Win32 Console Application. Введите название проекта, снимите галочку с Create directory for solution
- 3.4. Нажав OK, в появившемся окне выберите Application settings и отметьте пункт empty project. Нажмите Finish
- 3.5. Создайте в проекте новый hello.cpp файл, переименуте его в hello.cu
- 3.6. Скопируйте туда код примера
- 3.7. Добавьте Custom Build rule: правая кнопка мыши на проекте->Custom Build Rules. В списке отметьте один из пунктов "CUDA Runtime API Build Rule (v*.*)" и нажмите OK
- 3.8. В свойствах проекта выберите Linker->General и добавьте в Additional Library Directories $(CUDA_PATH)\lib\$(PlatformName)
- 3.9. В свойствах проекта выберите Linker->Input и добавьте в Additional Dependencies бибилотеку cudart.lib
- 3.10. Скомпилируйте и запустите программу по Ctrl+F5
- 3.11. Включения кода из файлов .cu в файлы .cpp в свойствах проекта выберите C/C++ ->General и добавьте $(CUDA_PATH)\include в Additional Include Directories
- 3.12. Для корректной линковки .cpp и .cu кода в свойствах проекта выберите C/C++ ->Code Generation и измените Runtime Library на /MT (релиз) или /MTd (дебаг). Проверьте, что это поле совпадает с Runtime API -> Host -> Runtime Library
- 3.13. Помимо указанного, для линковки .c и .cu кода процедуры из .cu файла должны быть выеделены с помощью extern "C" { }

4. pyCuda

- 3.0. Установите Питон 2.7 32-bit и numpy
- 3.1. С сайта http://www.lfd.uci.edu/~gohlke/pythonlibs загрузите и установите pycuda-2011.1.win32-py2.7.exe
- 3.2. Загрузите и распакуйте pytools-2011.3.tar.gz с сайта http://pypi.python.org/pypi/pytools, в папке выполните setup.py -install
- 3.3. Создайте init.bat со следующими строками:
  set HOME=%HOMEPATH%
  PATH = %PATH%;C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin
- 3.4. Теперь в консоли можно выполнить
  init
  имяфайла.py

II. Windows, GCC

1. OpenMP

- 1.1. Скачайте и установите TDM-GCC последней версии.
- 1.2. Добавьте в пути папку bin из установленного mingw-tdm
- 1.3. Скомпилируйте и соберите пример с помощью следующей строки:
  gcc hello_omp.c -o hello_omp.exe -fopenmp -lgomp -lpthread

2. MPI

- 2.1. Скачайте и установите MinGW последней версии
- 2.2. Добавьте в пути папку bin из установленного mingw
- 2.3. Скачайте и установите MPICH2
- 2.4. Скомпилируйте и соберите пример с помощью команд
  gcc -c hello_mpi.c -o hello_mpi.o -I"C:\Program Files\MPICH2\include"
  gcc -o hello_mpi.exe hello_mpi.o -L"C:\Program Files\MPICH2\lib" -lmpi
- 2.5. Запустив программу из командной строки, вы получите однопроцессорное приложение. Для запуска в несколько потоков запустите Пуск->Программы->MPICH2->wmpiexec.exe
- 2.6. В строке Application выберите ваш собранный в студии exe-файл, задайте требуемое количество процессов, отметьте галочку "run in an separate window" и нажмите кнопку Execute.

Исходники примеров:

- 1. OpenMP
- 2. MPI
- 3. CUDA

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. Ортега Дж. Введение в параллельные и векторные методы решения линейных систем: Пер. с англ. М.: Мир, 1991.367 с.
4. Нильсен М., Чанг И. Квантовые вычисления и квантовая информация: Пер. с англ. – М.: Мир, 2006 г. – 824 с., ил
5. Сайт http://www.mcs.anl.gov/mpi (Argonne National Laboratory, США).
6. Сайт http://www.parallel.ru (НИВЦ МГУ).
7. Сайт http://www.cluster.bsu.by (Белгосуниверситет, Минск).
8. Шпаковский Г. И., Серикова Н. В. Программирование для многопроцессорных систем в стандарте MPI. Мн.: БГУ, 2002. 323 с. Есть в интернет.
9. Закон Мура. ru.wikipedia.org/wiki/Закон_Мура
10. Векторизация программ: теория, методы, реализация. Сб. статей: пер. с англ. и нем. – М.: Мир, 1991. -275 с.
11. Шпаковский Г.И., А.Е.Верхотуров, Н.В.Серикова. Организация работы на вычислительном кластере. Учеб. пособие. – Мн.: БГУ, 2004. -181 с. Есть в интернет.
12. Кузюрин Н.Н., Фрумкин М.А. Параллельные вычисления: теория и алгоритмы // Программирование. 1991. N 2. С. 3–19.
13. Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009. - 77 с.
14. Казеннов А.М. Основы технологии CUDA. Компьютерные исследования и моделирование. 2010. Т.2. №3. с. 295-308.
15. Метод Гаусса (LINPAK) решения СЛАУ на кластере www.cs.berkeley.edu/~demmel/cs267/lecture12/lecture12.html
16. Введение в Grid Computing и Globus Toolkit. hext.mx1.ru/MyPresentation.ppt (на русском языке)
17. Облачные вычисления. http://ru.wikipedia.org/wiki/Облачные_вычисления
18. Александр Молдовян, Николай Молдовян, Борис Советов. Криптография. СПб.: "Лань", 2000. - 224 с., илл.
19. N.Karonis, B.Toonen, and I.Foster, “MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface”// Journal of Parallel and Distributed Computing (JPDC), Vol. 63, No. 5, pp. 551-563
20. Многоядерные микропроцессоры. http://ru.wikipedia.org/wiki/многоядерный процессор
21. В.Дьяконов. «Закон Мура» и компьютерная математика. EXPonenta Pro. Математика в приложениях. №1, 2003
22. С. Пахомов. Эра трехмерных транзисторов. КомпьютерПресс 1”2003.
23. А.В. Кудин. Взгляд в будущее. http://www.software.unn.ru/ccam/multicore/materials/arch/Ar18.pdf
24. Kane B.E. // Nature. 1998. V.393. P.133o137.

25. Валиев К.А., Кокин А.А. Квантовые компьютеры. Надежды и реальность. Москва. РХД, 2001, 352 стр.
26. Алгоритм Гровера. ru.wikipedia.org/.../Алгоритм_Гровера
27. Логинов О.В. Цыганов А.В. Квантовый алгоритм Гровера. www.exponenta.ru/.../grover/index.asp
28. Алгоритм Шора. www.rec.vsu.ru/rus/ecourse/quantcomp/sem9.pdf
29. Н.В.Сухочева. shk-internat@yandex.ru
30. С.Килин. Квантовая информация. ru.wikipedia.org/.../Килин С. Я.