Г.И. Шпаковский

КОРОТКО О ПАРАЛЛЕЛЬНОМ ПРОГРАММИРОВАНИИ И АППАРАТУРЕ

МИНСК
2013, январь

Справка о книге

Шпаковский Г.И. Коротко о параллельнм программировании им аппаратуре.

Суперкомпьютерные технологии в современном мире стали стратегической областью. Мощность национальных суперкомпьютеров сейчас так же важна, как мощность электростанций или количество боеголовок. И сейчас в мире началась гонка за экзафлопом.

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» Кроме того существует ряд сайтов [5 – 7] по различным проблемам параллельных вычислений.

Настоящая книга является сокращенным вариантом книги автора [8] и предназначена для первичного ознакомления со уровнями и механизмами параллельной обработки.

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

Книга будет полезна для программистов, желающих ознакомиться с параллельными вычислениями.

ОГЛАВЛЕНИЕ

ГЛАВА 1. Принципы параллельных вычислений **4**

1.1. Большие задачи
1.2. Методы повышения быстродействия
1.3. Формы параллелизма
1.4. Параллельные алгоритмы
1.5. Архитектуры параллеьных систем
1.6 Эффективность параллельных вычислений
1.7 Основные этапы развития параллельной обработки

ГЛАВА 2. Параллелизм на уровне команд **15**

2.1. Технологии VLIWи EPIC
2.2. Технология MMX
2.3. Расширение SSE

ГЛАВА 3. СИСТЕМЫ С ОБЩЕЙ ПАМЯТЬЮ **26**

3.1 Системы с общей памятью.
3.2. Стандарт OpenMP

ГЛАВА 4. Системы с распределенной памятью **34**
4.1. Параллельные алгоритмы
4.2. Кластерные системы
4.3. Метод Гаусса решения СЛАУ на кластерах
4.4. Стандарт MPI

ГЛАВА 5. Графические процессоры **55**
5.1. Графический процессор.
5.2. Неграфические вычисления на GPU
5.3. Модель программирования CUDA
5.4. Память CUDA
5.5. Программирование на CUDA
5.6. Гибридные вычислительные системы на основе GPU

ПРИЛОЖЕНИЕ 1. Установка и запуск параллельных программ **71**
ПРИЛОЖЕНИЕ 2. Матричные операции для MPI, OpenMP, CUDA
ПРИЛОЖЕНИЕ 3. Мнение

ИСТОЧНИКИ ИНФОРМАЦИИ **83**

ГЛАВА 1. ПРИНЦИПЫ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

1.1. Большие задачи
1.2. Методы повышения быстродействия
1.3. Формы параллелизма
1.4. Параллельные алгоритмы
1.5. Архитектуры параллельных систем
1.6. Эффективность параллельных вычислений
1.7. Основные этапы развития параллельной обработки

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.

<table>
<tr><th>Обозначение</th><th>Величина</th><th>Достигнутый уровень</th><th>Единицы измерения</th></tr>
<tr><td>Кило</td><td>10^{3}</td><td></td><td rowspan="6">Герцы,
байты,
FLOP/S</td></tr>
<tr><td>Мега</td><td>10^{6}</td><td></td></tr>
<tr><td>Гига</td><td>10^{9}</td><td>ПЭВМ до 5 Гфлопс</td></tr>
<tr><td>Тера</td><td>10^{12}</td><td rowspan="2">СуперЭВМ – тера и петафлопсы</td></tr>
<tr><td>Пета</td><td>10^{15}</td></tr>
<tr><td>Экза</td><td>10^{18}</td><td></td></tr>
</table>

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.

Ниже приведены времена решения некоторых больших задач на современной однопроцессорной системе

Проблема	Число (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

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

- За счет повышения быстродействия элементной базы (тактовой частоты). Быстродействие процессора растет пропорционально росту тактовой частоты, при этом не требуется изменения системы программирования и пользовательских программ. Но у повышения тактовой частоты есть физические границы.
- За счет увеличения числа одновременно работающих в одной задаче ЭВМ, процессоров, АЛУ, умножителей и так далее, то есть за счет параллелизма выполнения операций. Параллельная машина содержит множество процессоров П, объединенных сетью обмена данными

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.1.1. В своей первоначальной форме ИГ, тем не менее, не используется ни математиком, ни программистом, ни ЭВМ.

а *б*

$a_6 = (a_1 + a_2)a_3 + a_4 a_5$

Рис.1.1. Информационный граф математического выражения (*а*) и порядок выполнения операций в выражении (*б*)

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

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

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

Изучение ряда алгоритмов и программ показало, что можно выделить следующие основные формы параллелизма:

- 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.1.2.

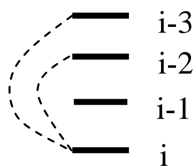


Рис.1.2. Предварительная подготовка операндов для команды i.

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

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

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. 1.3.).

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

Это параллелизм очень коротких ветвей и данный вид параллелизма должен автоматически выявляться аппаратурой ЭВМ в процессе выполнения машинной программы или программно путем использования сверхдлинных каманд.

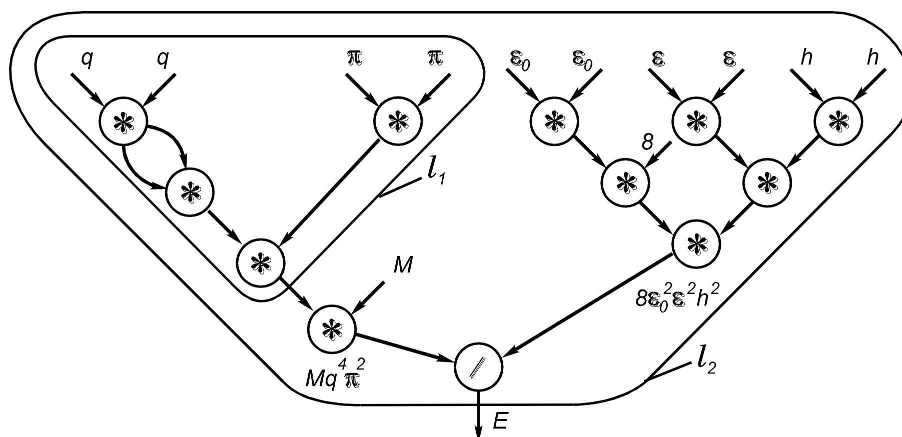


Рис.1.3. ЯПФ вычисления величины E

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

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 языке записывается в виде такой векторной операции:

```
      C (*, j) = A (*, j) + B (*, j).
```

Области применения векторных операций над массивами обширны.

Параллелизм независимых ветвей. Рассмотрим пример. Пусть задана система уравнений:

$$
\begin{aligned}
&f_1(x_1, x_2, \ldots, x_n) = 0,\\
&f_2(x_1, x_2, \ldots, x_n) = 0,\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&f_n(x_1, x_2, \ldots, x_n) = 0.
\end{aligned}
$$

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

$$
\begin{aligned}
x_1^{(k+1)} &= F_1(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}),\\
x_2^{(k+1)} &= F_2(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}),\\
x_3^{(k+1)} &= F_3(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}).
\end{aligned}
$$

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

```
L      FORK M1, M2, M3
M1     Z1 = F1 (X1, X2, X3)
       JOIN (R, K)
M2     Z2 = F2 (X1, X2, X3)
       JOIN (R,K)
```

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

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. Этот процесс представлен на рис.1.4.

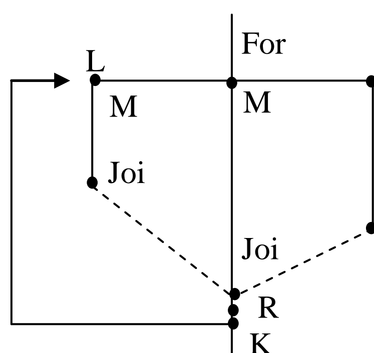


Рис.1.4. Блок-схема выполнения параллельной программы.

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.

1.4. Параллельные алгоритмы

Области применения паралleьных алгоритмов обширны: цифровая обработка сигналов (цифровые фильтры), механика, моделирование сплошных сред, метеорология, оптимизация, задачи движения, расчеты электрических характеристик БИС и т. д. Но все они основаны на математике, на вычислительных алгоритма. Имеются различные оценки паралелизма этих алгоритмов. Одна из них приведена ниже в таблице .

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	$0(\log n \log (n/\varepsilon))$	$0(n^3 / \varepsilon^2)$
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 выражений 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)

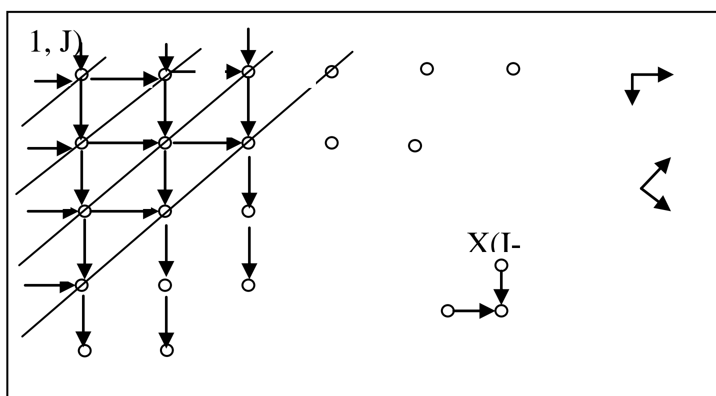
в) Итерация i s =

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):

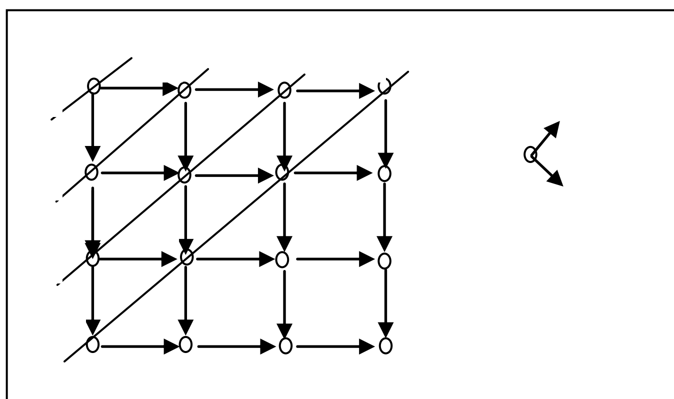
```
DO 1  I = 1,N
DO 1  J =  1,N
1   X (I, J) = X (I-1, J) + Y (I, J –1) + C
```



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, написав соответствующую программу.

1.5. Архитектуры параллельных систем.

Архитетура есть совокупность главных свойств, отличающих некоторую систему от других внутри некоторой общности объектов. Архитектура явлвется основой классификации этих систем.

Существует много классификаций 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, обрабатываемых одним потоком команд. На этой основе Флинн построил свою классификацию параллельных ЭВМ с крупнозернистым параллелизмом:

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

Таким образом, для крупноформатного параллелизма оказались существенными две позиции:

- OKMD - векторный машины и
- MKMD - машины для обработки независимых ветвей, которые в свою очередь делатся на ЭВМ с общей и распределенной память.

Развитие параллельных архитектур на начальных этапах определялось возможностями элементной базы. Когда была построена ЭВМ полностью на транзисторах, возможности микроэлекроники позволили ввести в состав машины несколько АЛУ. Это сделало возможным реализацию технологий VLIW и MMX.

Дальнейшие успехи микроэлектроники позволили построить многопроцессорную систему с общей памятью. В этих системах было крупное достоинство – простота программирования, так как не нужно было программировать пересылку данных между ветвями, но количество процесссоров ограничивалось пропускной способностью памяти и не превышало 32.

Следующим этапом стало построение многопроцессорных систем с распределенной памятью. Закон Мура гласит:

Каждые 2 года количество транзисторов на кристалле удваивается.

Сейчас на одном кристалле можно разместить миллиарды транзисторов, то есть сотни процессоров, которые можно связать в систему с распределенной памятью. Не ограничивается и возможность объединения таких кристаллов.

В списке TOP 500 на первом месте находится система, содержащая 1.5 млрд ядер (процессоров).

1.6. 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]:

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

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

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.

1.7 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.

Практическая реализация основных идей параллельной обработки началась только в 60-х гг. 20 - го столетия, когда появился транзистор, который позволил строить машины, состоящие из большого количества логических элементов.

Некоторые этапы развития параллельных ЭВМ качественно можно представить следующей таблицей:

№	Название ЭВМ	Годы	Новизна	Программы
1	D825 – одна из первых многопроцессорных систем	1962	Доказана возмоность построения многпроцессорных 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
5	Кластер Beowulf	1994	Сборка на серийном оборудовании	Использованы обычные сетевые ОС
7	Многоядерные процессоры	2001	Разработаны процессоры с общей и индивидуальной памятью	OpenMP и MPI. Нужны новые разработки
8	Квантовый компьютер Orion компании D-Wave	2007	Кубит, эспоненцальная скорость за счет суперпозиции	Алгоритмы Шора, Гровера. Языки моделирования

ГЛАВА 2. МЕЛКОЗЕРНИСТЫЙ ПАРАЛЛЕЛИЗМ

2.1. Технология VLIWи EPIC
2.2. Технология MMX
2.3. Расширение SSE

Мелкозернистый параллелизм обеспечивается за счет параллелизма смежных команд последовательной программы. Ускорение при этом не превышет нескольких раз, но и дополнительного оборудования требуется немного: в структуру процессора нужно добавить всего несколько АЛУ.

Если ваш компьютер поддерживает описанные ниже расширения системы команд, то это отражено в документации на компьютер и систему программирования. При отсутствии такой нформации можно написать простые тесты с использованием команд из этих наборов.

В этой главе будут рассмотрены две реализации мелкозернистого параллелизма – технологии VLIW и MMX.

2.1. 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.2.1 представлена структура процессора. Блоки выборки программ, диспетчирования и декодирования команд могут каждый такт доставлять функциональным блокам до восьми 32-разрядных команд. Обработка команд производится в в путях A и B, каждый из которых содержит 4 функциональных устройства (L, S, M, и D) и 16 32-разрядных регистра общего назначения. Каждое ФУ одного пути почти идентично соответствующему ФУ другого пути.

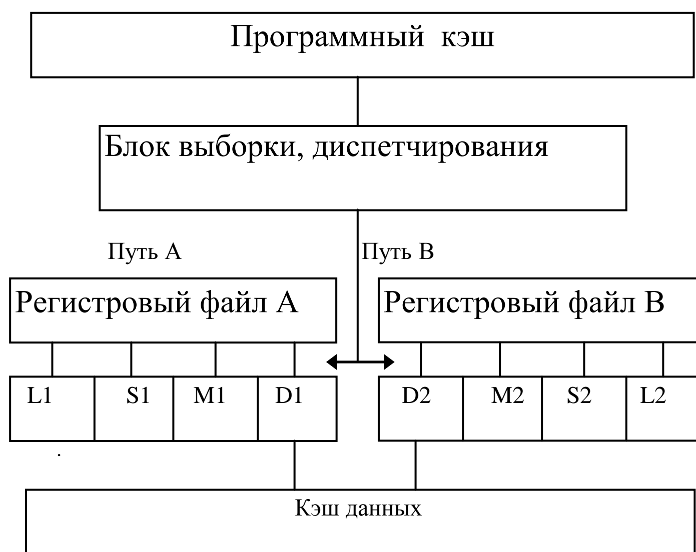


Рис.2.1. Структура микропроцессора С6

Каждое ФУ прямо обращается к регистровому файлу внутри своего пути, однако существуют дополнительные связи, позволяющие ФУ одного пути получать доступ к данным другого пути. Функции 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.2.2.

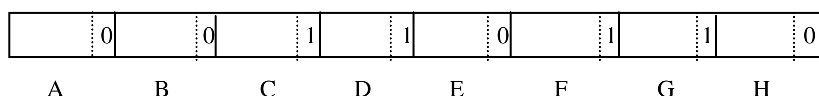


Рис.2.2. Частично последовательный пакет

Состав исполнительных пакетов для этой команды представлен ниже:

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

Технология сверхдлинной команды EPIC (Explicitly Parallel Instruction Computing) компаниии Intel, реализована в микропроцессоре Itanium. Принципы позволяют во время исполнения отказаться от проверки зависимостей между операциями, которые компилятор уже объявил как независимые. Кроме того, данная архитектура позволяет отказаться от сложной логики внеочередного исполнения операций.

Параллелизм в EPIC реализуется следующим образом. Команды упаковываются компилятором в *связку* длиною в 128 разрядов. Связка содержит 3 команды и шаблон, в котором указанываются зависимости между командами.

Команда EPIC

Перечислим все варианты составления связки из 3-х команд (знак || обозначает возможность параллельного исполнения смежных команд, s соответствует stop-разряду):

i1 || i2 || i3 - все команды исполняются параллельно
i1 s i2 || i3 - сначала i1, затем исполняются параллельно i2 и i3
i1 || i2 s i3 - параллельно исполняются i1 и i2, после них - i3
i1 s i2 s i3 - последовательно исполняются i1, i2, i3

Одна такая связка, состоящая из трех команд, соответствует набору из трех функциональных устройств процессора. Процессоры могут содержать разное количество таких блоков, оставаясь при этом совместимыми по коду. Ведь благодаря тому, что в шаблоне указана зависимость и между связками, процессору с N одинаковыми блоками из трех функциональных устройства будет соответствовать командное слово из N*3 команд (N связок). Таким образом должна обеспечиваться масштабируемость EPIC.

Поле шаблона используется процесором для быстрого декодирования связки и посылки команд на соответствующие устройства. Последний (пятый) разряд шаблона отмечает, может ли следующая связка команд выполняться параллельно с текущей связкой. Связки могут быть сцеплены, создавая группы произвольной длины.

Технология EPIC реализована Intel в процессорах Itanium [9].

2.2. 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, полноэкранное 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.

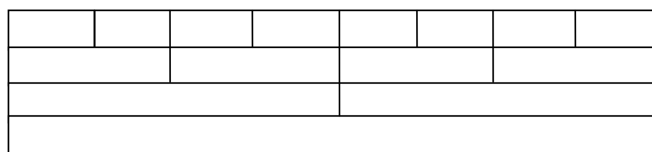
Это и определило новую структуру данных и расширение системы команд.

Технология MMX выполняет обработку параллельных данных по методу SIMD. По существу, это мелкозернистая реализация крупнозернистого параллелизма.

В технологии 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.

a3	a2	a1	FFFFh
+		+	+
b3	b2	b1	8000h
=	=	=	=
a3 + b3	a2 + b2	a1 + b1	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.

a3	a2	a1	FFFFh
+		+	+
b3	b2	b1	8000h
=	=	=	=
a3 + b3	a2 + b2	a1 + b1	FFFFh

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.

a3	a2	a	1
*	*	*	
b3	b2	b	1
	–		–
a3* b3 + a2*b2		a1* b1	

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).

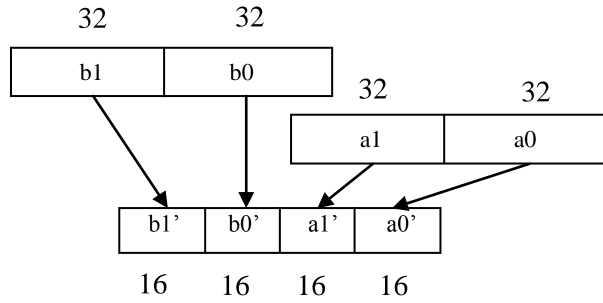
23	45	16	34
gt?	gt?	gt?	gt?
31	7	16	67
=	=	=	=
0000h	FFFFh	0000h	0000h

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 значений. Эти операции нуждаются в большей точности, чем та, которую может обеспечить 8-разрядный формат для пикселов. Решением проблемы точности является распаковка 8-разрядных пикселов в 16-разрядные слова, выполнение расчетов в 16-разрядной сетке и затем обратная упаковка в 8-разрядный формат для пикселов перед записью в память или дальнейшими вычислениями.

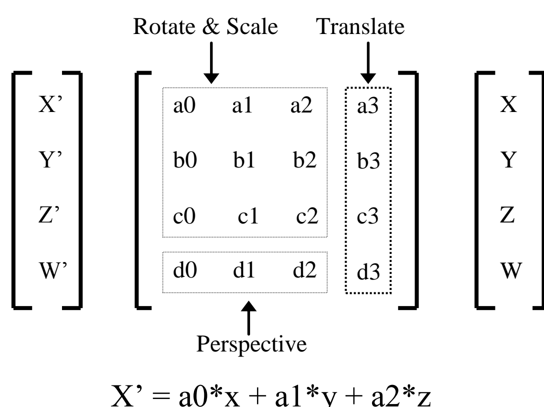
Рассмотрим некоторые примеры использования технологии MMX для кодирования базовых прикладных операций.

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

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

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.

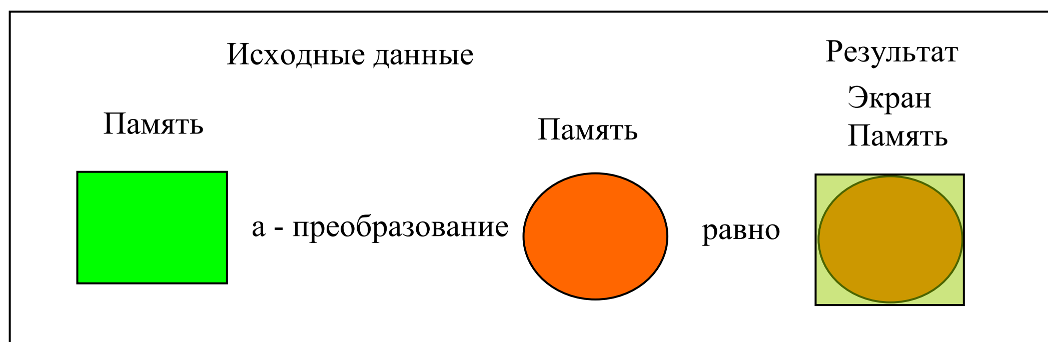


X' = a0*x + a1*y + a2*z

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

$$П_{рез} = П_{круг}*(альфа/255) + П_{квадр}*(1-(альфа/255)) \qquad альфа = 1\ldots255$$

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

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

Если использовать для представления образов экран с разрешением 640x480 и все 255 ступеней значения альфа, то преобразование квадрата в круг потребует для технологии MMX 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.

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)))

Кроме MMX имеется еще ряд расширений системы команд i86, например, технологии SSE (**Streaming SIMD Extensions**) . Для SSE в структуру процессора введено восемь новых 128-битных регистров. На этой базе кроме SSE появились наборы команд SSE, SSE2, SSE3, SSE4, которые значительно увеличивает производительность приложений с интенсивными вычислениями. Новые SIMD-команды, используются в следующих областях:

- видео
- комбинирование графики и видео
- обработка изображений
- звуковой синтез
- распознавание, синтез и компресия речи
- телефония
- видео конференции
- 2D и 3D графика.

В интернет имеесся много информации по по расширеням, включая MMX, и процессорах, поддерживающих эти расширения [9, 10].

ГЛАВА 3. СИСТЕМЫ С ОБЩЕЙ ПАМЯТЬЮ

3.1 Системы с общей памятью.
3.2. Стандарт OpenMP

3.1. Системы с общей памятью

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

Схема многопроцессорной системы с общей памятью (**Symmetric Multiprocessing - SMP)** представлена ниже:

Схема многопроцессорной системы с общей памятью

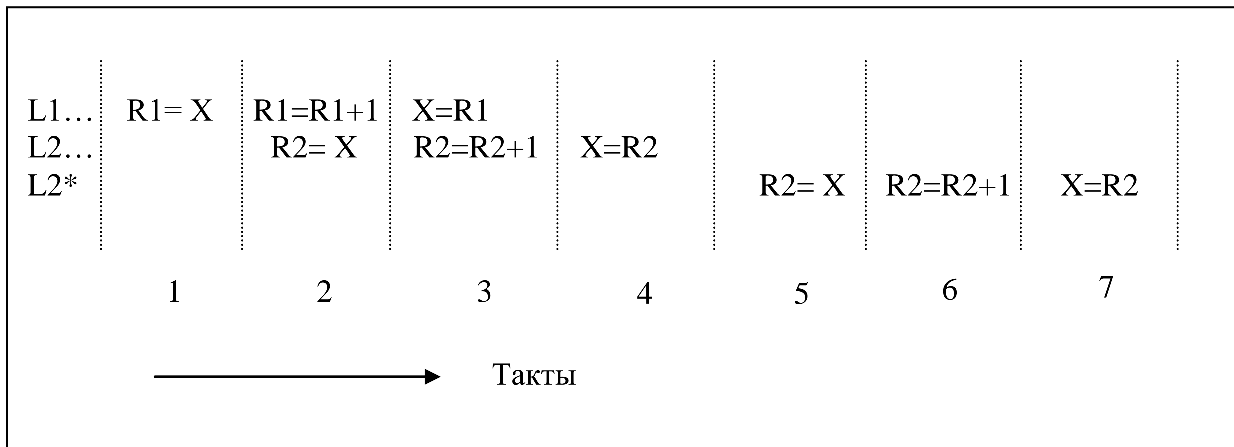
Наличие общей памяти вызывает как положительные, так и отрицательные последствия:

- 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).

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.

3.2. OpenMP

Интерфейс OpenMP [11] 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:

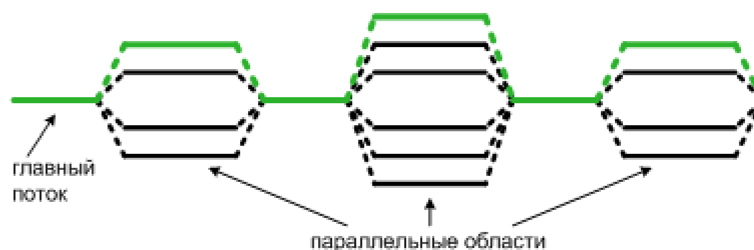
- Недостаточно эффективна на SMP-системах.
- Относительно сложна в освоении, так как требует мышления в "невычислительных" терминах. POSIX-интерфейс для организации нитей (**Pthreads**) поддерживается широко (практически на всех UNIX-системах), однако по многим причинам не подходит для практического 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 параллельной 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.

Если OpenMP-программа выполняется на однопроцессорной платформе, то 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. 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 на OpenMP (рис.3.1).

```
#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.3.1. 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 относительно 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 «за-

глушки» (**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, реализующаяю перемножение двух квадратных матриц, представлена на рис.3.2. 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.

```
#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.3.2. Перемножение матриц на языке Си.

Установка OpenMP описана в [11], подробности OpenMP даны в [12], компиляция и запуску многопоточных программ даны в [13].

ГЛАВА 4. СИСТЕМЫ С РАСПРЕДЕЛЕННОЙ ПАМЯТЬЮ

4.1. Параллельные алгоритмы
4.2. Кластерные системы
4.3. Метод Гаусса решения СЛАУ на кластерах
4.4. Стандарт MPI

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

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

4.1. Многопроцессорные системы с индивидуальной памятью или массивно-параллельные системы (MPP)

Проблема масштабируемости решается в системах с распределенной (индивидуальной) памятью (рис.4.1), в которых число процессоров практически не ограничено.

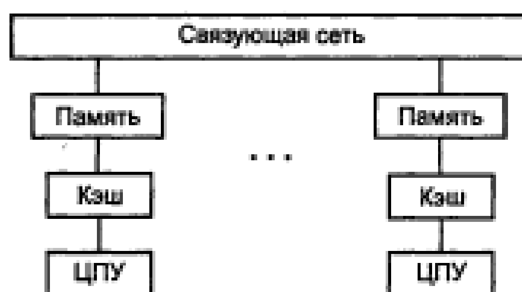


Рис.4.1. Схема Эвм с индивидуальной памятью.

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 $W_с$ – количество передач данных, $t_с$ – 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:

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.4.2.

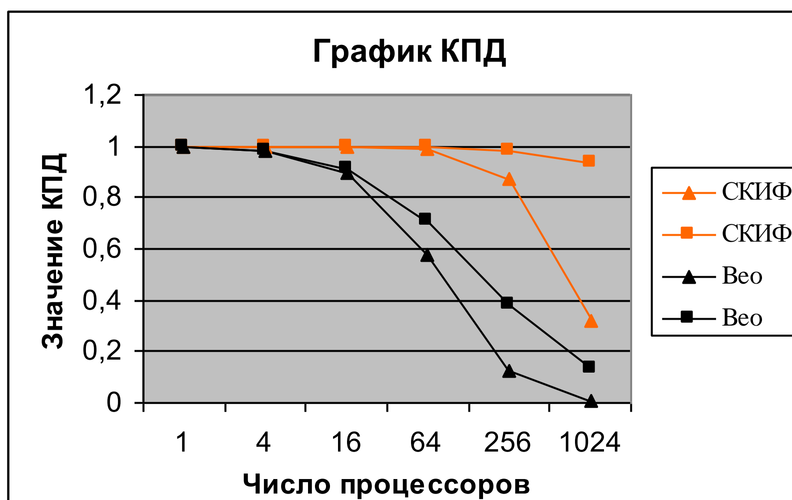


Рис. 4.2. График коэфициента утилизации для СКИФ и Beowulf. Квадратами отмечены кривые без учета латентности коммутатора, треугольниками – с учетом наличия латентности (2 мкс – для СКИФ, 80 мкс – для Beo).

4.2. 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 распределенной памятью идеально подходят подходят для реализации на основе электронной и программной продукции массового производства. При этом не требуется разработки новой 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 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 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. Кластер состоял из 16 компьютеров. Особенностью такого кластера является масштабируемость, то есть возможность увеличения количества узлов системы для увеличения производительности. Узлами в кластере могут служить любые серийно выпускаемые 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 некоторые кластеры СКИФ вошди в список Top 500 самых высокопроизводительных систем мира.

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. Кластер этого типа отличается от локальных сетей системным программным обеспечением.
- Монолитные кластеры, все оборудование которых компактно размещено в специализированных стойках массового производства. Это очень быстрые системы. Процессоры в монолитных кластерах не могут использоваться в персональном режиме.

Для программирования кластеров используется широко распространенная библиотека функций обмена **MPI** (Message Passsing Interfase), описанная далее. Для реализации этих функций разработчики MPI создали специальный пакет **MPICH**, решающий эту и многие другие задачи.

Организация кластера. Пример структуры кластера на базе локальной сети представлен на рис. 4.3. 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).

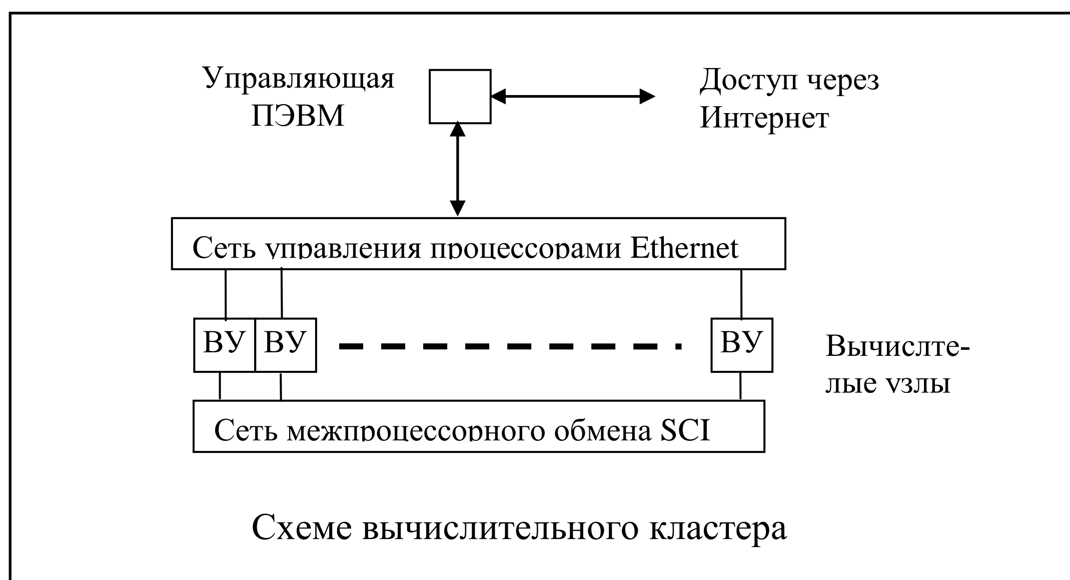


Рис.4.3. Сзема вычислительного кластера.

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.

Коммуникационные системаы вычислительных кластеров являются сложными аппаратно-программными комплексами и по ним имеется обширная литература. Первичные сведения можно получить в [8].

4.3. Метод Гаусса решения СЛАУ на кластере.

Метод решения СЛАУ с постолбцовым выбором главного элемента. Этот материал основан на статье [14]. Рассматриваются системы вида:

$$
\begin{aligned}
&a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1,\\
&a_{21}x_{11} + a_{22}x_2 + \ldots + a_{2n}x_n = b_2,\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&a_{n1}x_1 + a_{n2}x_2 + \ldots. + a_{nn}x_n = b_n,
\end{aligned} \qquad (4.1)
$$

или иначе, системы векторно-матричных уравнений

$$Ax=b.$$

Наиболее известным методом решения систем вида (4.1) 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 $-\frac{a_{21}}{a_{11}}, -\frac{a_{31}}{a_{11}}, \ldots.., -\frac{a_{n1}}{a_{11}}$. Результатом этого этапа преобразования будет эквивалентная (5.1) система:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\
a_{32}^{(1)}x_2 + a_{33}^{(1)}x_3 + \ldots + a_{3n}^{(1)}x_n &= b_3^{(1)} \qquad (4.2) \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_{n2}^{(1)}x_2 + a_{n3}^{(1)}x_3 + \ldots + a_{nn}^{(1)}x_n &= b_n^{(1)}
\end{aligned}
$$

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

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

На втором этапе проделываем такие же операции, как и на первом, с подсистемой системы (4.2), получающейся исключением первого уравнения. Эквивалентный (4.2) результат второго этапа будет иметь вид:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \\
a_{33}^{(2)}x_3 + \ldots + a_{3n}^{(2)}x_n &= b_3^{(2)} \qquad (4.2\text{ а}) \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_{n3}^{(2)}x_3 + \ldots + a_{nn}^{(2)}x_n &= b_n^{(2)}
\end{aligned}
$$

где

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

Продолжая этот процесс, на (n -1)–ом этапе так называемого прямого хода метода Гаусса данную систему (4.1) приведем к треугольному виду:

$$
\begin{aligned}
a_{11}x_1 + a_{21}x_2 + a_{13}x_3 + \ldots + a_{1n}x_{1n} &= b_1 \\
a_{22}^{(1)}x_2 + a_{23}^{(1)}x_3 + \ldots + a_{2n}^{(1)}x_n &= b_2^{(1)} \qquad (4.3) \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
a_{nn}^{(n-1)}x_n &= b_n^{(n-1)}
\end{aligned}
$$

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

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

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 (4.3) позволяет последовательно одно за другим вычислять значения неизвестных, начиная с последнего:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}};$$

....

$$x_2 = \frac{b_2^{(1)} - a_{23}^{(1)} x_3 - ... - a_{2n}^{(1)} x_n}{a_{22}^{(1)}};$$

$$x_1 = \frac{b_1 - a_{12} x_2 - ... - a_{1n} x_n}{a_{11}}.$$

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. $\textit{Для } k = 1, 2, ..., n-1, 2$
2. $\quad \textit{Найти } m \geq k, \textit{что} \left| a_{mk} = \max \left| a_{ik} \right| \right|, \textit{обменять строки } m \textit{ и } k$
3. $\quad \textit{Для } i = k+1, ..., n:$
4. $\quad\quad t_{ik} := a_{ik} / a_{kk},$
5. $\quad\quad b_i := b_i - t_{ik} b_{k,};$
6. $\quad \textit{Для } j = k+1, ..., n:$ (4.4)
7. $\quad\quad a_{ij} := a_{ij} - t_{ik} a_{kj}.$
8. $x_n := b_n / a_{nn};$
9. $\textit{Для } k = n-1, ..., 2, 1:$
10. $\quad x_k := \left(b_k - \sum_{j=k+1}^{n} a_{kj} x_j \right) \Big/ a_{kk.}$

Подав на его вход квадратную матрицу коэффициентов при неизвестных системы (4.1) 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:

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

Очевидно, что все y_i могут быть последовательно найдены при $i=1, 2, \ldots, n$ по формуле

$$y_i = b_i - \sum_{k=1}^{i-1} l_{ik} y_k \qquad (4.5)$$

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

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \ldots\ldots + u_{1n}x_n &= y_1 \\
+ u_{22}x_2 + \ldots\ldots + u_{2n}x_n &= y_2 \\
\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\
u_{nn}x_n &= y_n
\end{aligned} \qquad (4.6)$$

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

$$x_i = \frac{1}{u_{ii}}\left(y_i - \sum_{k=i+1}^{n} u_{ik} x_k \right) \qquad (4.7)$$

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 (4.3) для МГ. Переход от МГ к КСГ рассмотрим на примере разложения в МГ (4.2): 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.4.2 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 (4.3) для МГ. Переход от МГ к КСГ рассмотрим на примере разложения в МГ (4.2): 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 (4.2 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
[14]. Будет обсужден порядок слоев данных, начиная с самого простого, но не эффективного варианта, и более эффективные варианты. Система обозначений показана на следующем рис.4.4.

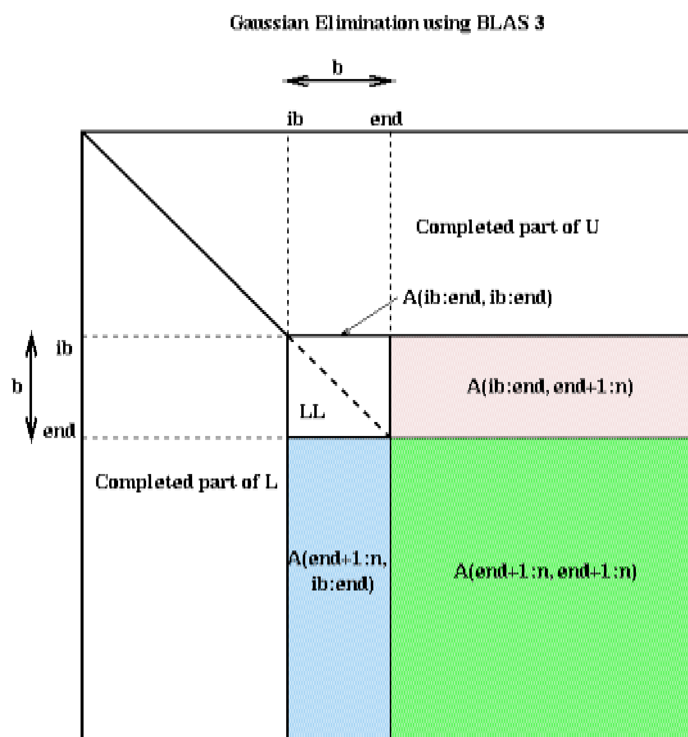
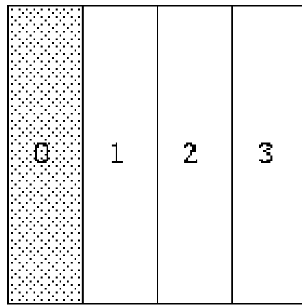


Рис. 4.4. Методы блочного размещения данных в кластере

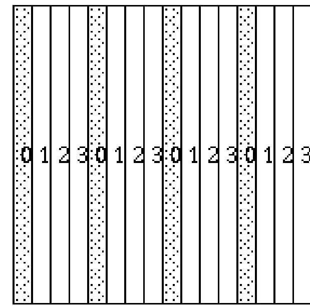
Двумя главными затруднениями в выборе размещения данных для гауссова исключения являются:

- 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

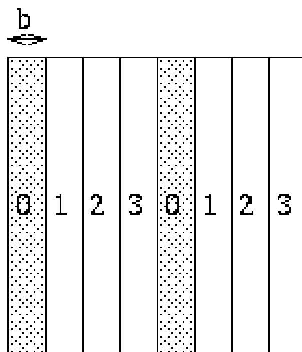
Примечание. Уровенень BLAS1 библиотеки BLAS используется для выполнения операций вектор-вектор, уровенень BLAS2 – для выполнения матрично-векторных операций, уровенень BLAS3 – для выполнения матрично-матричных операций.



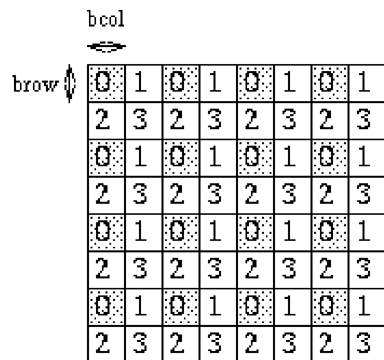
1) Column Blocked Layout



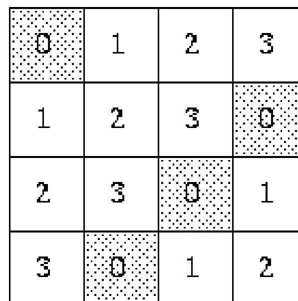
2) Column Cyclic Layout



3) Column Block Cyclic Layout



4) Row and Column Block Cyclic Layout



5) Block Skewed Layout

Рис.4.5. Варианты секционирования матриц

Понять эти проблемы помогает понять это рис.4.5. 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 напра-

во. 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 стол-

бец 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, 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.

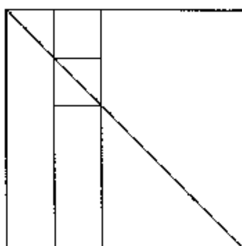


Figure 3.6 Left-Looking LU Algorithm

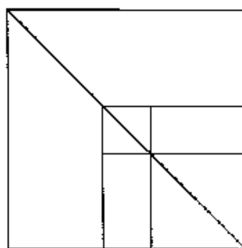


Figure 3.7 Right-Looking LU Algorithm

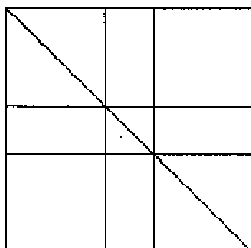


Figure 3.8 Crout LU Algorithm

Блочные методы (BLAS-3)

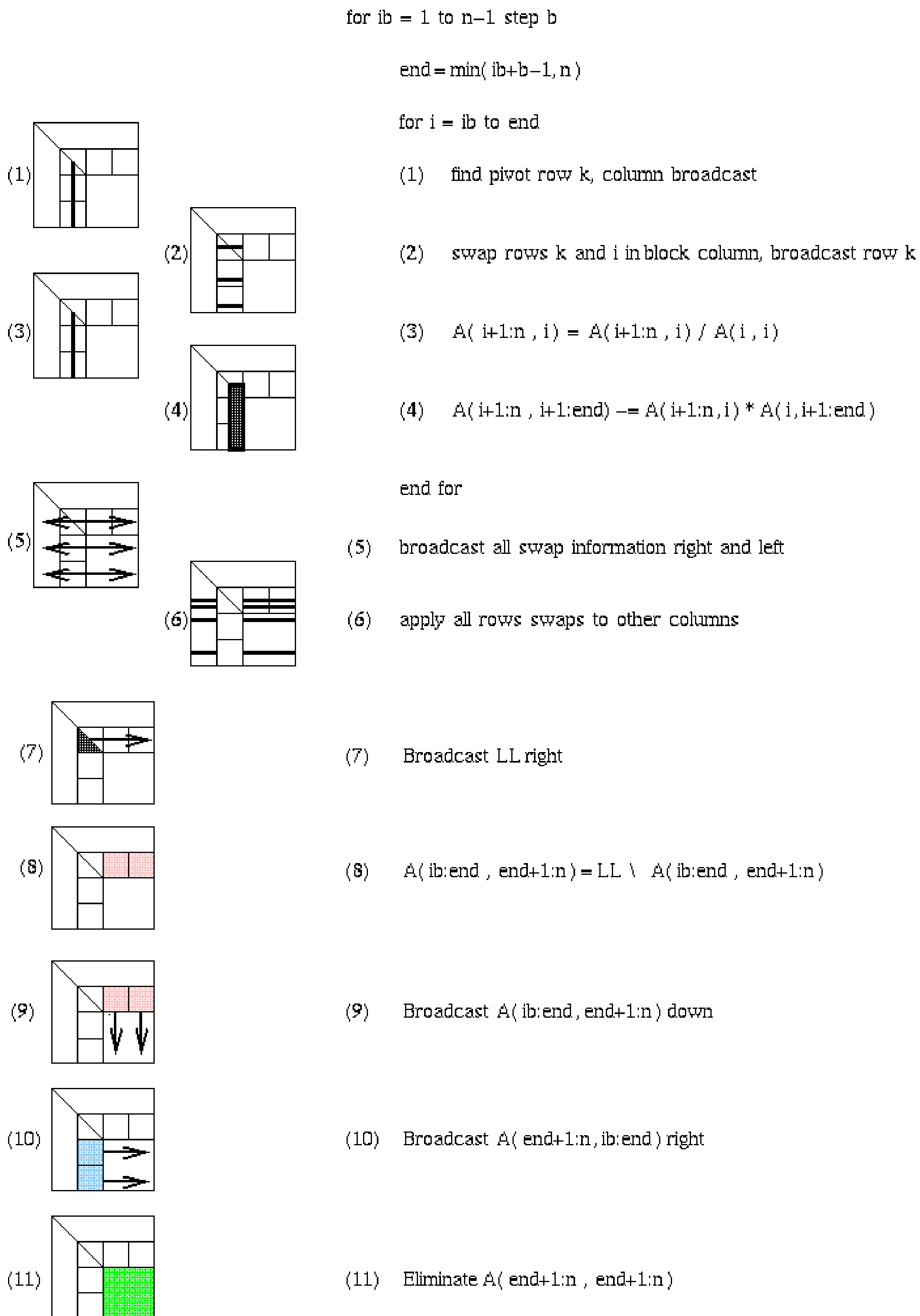


Рис.4.6. Алгоритм BLAS3 на блочно-циклическом размещении матрицы.

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 рис. 4.6 повторяется ниже, она во многом соответствует алгоритму (4.4).

```
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)                    (4.8)
        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.

Эффективность 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 (4.9)
+ (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 (4.10)
+ (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.

4.4. 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. Вся информация по MPI сборнике [15], в том числе и примеры. В сборнике размещены три книги:

- Антонов - Параллельное программирование с использованием технологии MPI Год выпуска: 2004
- Шпаковский, Серикова - Программирование для многопроцессорных систем в стандарте MPI Год выпуска: 2002
- Корнеев - Параллельное программирование в MPI Год выпуска: 2002

MPI 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 С.

Для первой параллельной программы удобна программа вычисления числа π, поскольку в ней нет загрузки данных и легко проверить ответ. Вычисления сводятся к вычислению интеграла 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.4.7.

```
#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 = 3.141592653589793238462643;  /* 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. 4.7. Программа вычисления числа π 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). Текст программы размещен в Приложении 2.

ГЛАВА 5. ВЫЧИСЛЕНИЯ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

5.1. Графический процессор.
5.2. Неграфические вычисления на GPU
5.3. Модель программирования CUDA
5.4. Память CUDA
5.5. Программирование на CUDA

5.1. Графический процессор.

В 2007 г. корпорация **NVIDIA** представила первую версию технологии **CUDA** (**Compute Unified Device Architecture**) для программирования на **GPU** (**Graphics Processing Unit**). Разработчики ставили цель, чтобы GPU использовались не только для создания изображений в 3D приложениях, но и применялись в других параллельных расчетах.

Но, конечно, GPU не заменят **CPU (Central Processing Unit)** . Сейчас ясно, что видеочипы движутся постепенно в сторону CPU, становясь всё более унивесальными (расчёты с плавающей точкой одинарной и двойной точности, целочисленные вычисления), так и CPU становятся всё более «параллельными», обзаводясь большим количеством ядер, технологиями многопоточности, не говоря про появление блоков SIMD и проектов гетерогенных процессоров. Скорее всего, GPU и CPU в будущем просто сольются. Известно, что многие компании, в том числе Intel и AMD работают над подобными проектами. И неважно, будут ли GPU поглощены CPU, или наоборот.

Видеокарта, графический процессор GPU - устройство компьютера, выполняющее графический **рендеринг**, то есть преобразование графического образа, хранящегося в памяти компьютера, в форму, предназначенную для дальнейшего вывода на экран монитора.

Основным элементом карты является графический процессор, который непосредственно и занимается формированием самого графического образа.

У графического процессора видеокарты работа простая и распараллеленная изначально. В соответствии с графическим конвейером GPU принимает на входе группу **полигонов** (плоскость между тремя вершинами), проводит все необходимые операции, и на выходе выдаёт пиксели (биты изображения для экрана). Обработка полигонов и пикселей независима, их можно обрабатвать параллельно, независимо друг от друга. Поэтому, изначально в GPU используется большое количество простых параллельно работающих исполнительных блоков, называемых потоковыми процессорами **SP** (**Streaming Processor**). SP - это просто **АЛУ**, что и показано ниже на рис.X. Понятно, что при тех же размерах кристалла, на нем можно поместить значително больше SP, чем процесоров CPU. Считается, что количество SP на кристалле графического процессора может сосtвлять сотни и тысячи.

.

Ниже представлена структурная схема конкретного GPU среднего класса NVIDIA GeForce 8800. Он содержит 8 мультипроцессоров, называемых ядрами. Каждое ядро содержит 16 АЛУ. Значит, всего в этом GPU имеется 128 АЛУ, в то время, как в универсальные CPU содержат только несколько ядер (до 8 по состоянию на 2012 г.)

Тенденции увеличения количества вычислительных блоков продолжается, что приведет к появлению к 2015 году графических процессоров, оснащенных несколькими тысячами ядер.

Если считать, что мультипроцессор – это один или несколько многоядерных процессоров с общей памятью, то графический процессор по структуре является обычным кластером.

5.2. Неграфические вычисления на графических процессорах

В 1999-2000 годах специалисты в компьютерной области и научные работники из разных сфер перешли на использование GPU для ускорения ряда научных приложений [16,17]. Это стало началом движения **GPGPU (вычисления**

общего назначения на GPU). Производительность GPGPU в некоторых случаях превышала производительность CPU более чем в 100 раз. Однако, для GPGPU были необходимы удобные средства программирования. Компания NVIDIA поэтапно пришла к системе программирования CUDA, которая опиралась на известные языки C, C++ и Fortran.

Комбинация CPU + GPU обеспечивает большое быстродействие, потому что CPU состоят из нескольких ядер, оптимизированных для последовательной обработки данных, а на кристалле GPU размещаются тысячи маленьких, более производительных ядер, созданных для параллельной обработки данных. Последовательные части кода обрабатываются на CPU, а параллельные части - на GPU. Естественно, CPU выполняет также и административные функции.
Такие системы называются **гибридными.**

В настоящее время гибридные системы широко используются во всем мире, в том числе и в России, в частности, в МГУимеются компьютеры Ломоносов, СКИФ-МГУ, Чебышев, BlueGene, ГрафИТ, которые будут применяться для решения задач сейсморазведки, молекулярной динамики, криптографии, газо- и гидродинамики, компьютерного дизайна лекарств, и ряда других задач.

Эффективность суперкомпьютеров - это отношение реальной производительности к пиковой. Эффекивность суперкомпьютеров гибридной архитектурой CPU+GPU рейтингу значительно ниже, чем систем с однородной архитектурой CPU. Так, в МГУ реальная производительность суперкомпьютера ГрафИТ составляет 11,98 Тфлопс, а эффективность – около 45%. У суперкомпьютера ИПМ им. М.В. Келдыша этот показатель и того меньше – 38%. **Однако, эффективность в расчете на единицу стоимости или на единицу потребляемой электрической энергии существенно выше, чем на CPU.**

Отметим, что создание суперкомпьютеров с архитектурой CPU+GPU можно назвать общемировой тенденцией. В списке **top500** большинство суперкомпьютеров используют ускорители на графических процессорах.

Проблема в программировании под GPU не столько в «сложности написания кода», сколько в крайне узком спектре задач и алгоритмов, которые принципиально выигрывают от выполнения на них.

GPU — единственная доступная по деньгам возможность получить «персональный суперкомпьютер» терафлопсной мощности в обычном настольном корпусе.

5.3. Модель программирования CUDA

CUDA основана на концепции SIMD, которая подразумевает, что одна инструкция позволяет одновременно обработать множество данных.

Графический процессор (ГП) представлен на рис. ниже. Он состоит из состоит из мультиплексоров и различных видов памяти. Именно в ГП реализуется принцип SIMD: каждая команда из Instruction Unit выполняется всеми активированными процессорами мультиплексора (рис.5.1).

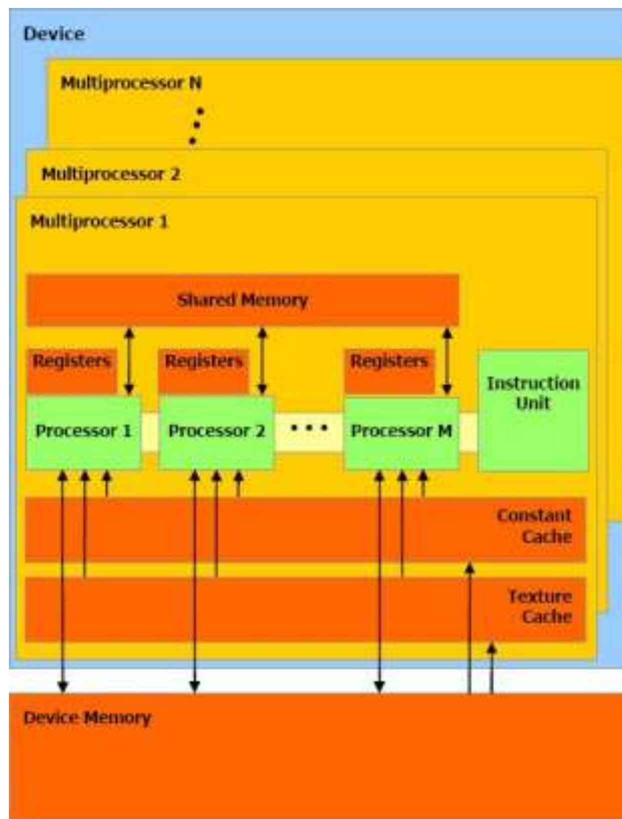


Рис.5.1. Устройство графического процессора

В документации NVIDIA использует собственную терминологию. Поясним некоторые термины:

Мультипроцессор — это многоядерный SIMD процессор, позволяющий в каждый определенный момент времени выполнять на всех ядрах только одну инструкцию. Каждое ядро мультипроцессора скалярное, т.е. оно не поддерживает векторные операции в чистом виде.

Устройством (device) - специализированное устройство, предназначенное для исполнения программ, использующих CUDA. В нашей статье мы рассмотрим GPU только как логическое устройство, избегая конкретных деталей реализации.

Хост (host) – это программа в обычной оперативной памяти компьютера, использующая CPU и выполняющую управляющие функции по работе с устройством.

Ядро (kernel) - в CUDA это процедура в N потоках, запущенная в устройстве (не связано с понятием ядра в многоядерном процессоре).

Поток (thread) – набор данных, который необходимо обработать (не требует больших ресурсов при обработке). Отличается от CPU/

Варп (warp) – группа из 32 потоков. Данные обрабатываются только варпами, следовательно варп – это минимальный объем данных.

Блок (block) – совокупность потоков (от 64 до 512) или совокупность варпов (от 2 до 16).

Сетка (grid) – это совокупность блоков. Такое разделение данных применяется исключительно для повышения производительности.

Фактически, та часть программы, которая работает на CPU — это хост, видеокарта GPGPU— устройство, а ядро – программа CUDA. Логически **устройство** можно представить как набор мультипроцессоров плюс драйвер CUDA.

Особенностью архитектуры CUDA является блочно-сеточная организация данных (рис.5.2.). Схема подчинености данных такова:

- Нить (thread) – минимальная единица обработки.
- Блок (blok) – объединяет определенное количество нитей.
- Сеть (grid) - самая крупная единица данных, состоит из блоков

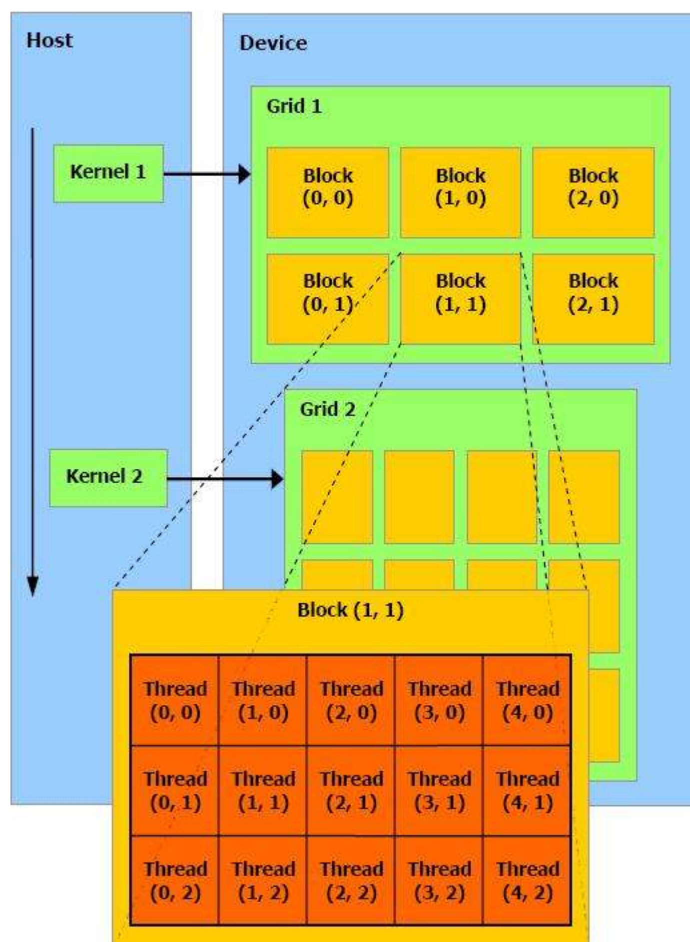


Рис.5.2. Организация потоков

На рис.5.2. ядро обозначено как Kernel. Kernel может быть программой одного ядра многоядерного процессора. В случае объединения нескольких вычислительных многоядерных узлов с несколькими графическими устройствами задача распараллеливается:

- на отдельные подзадачи путем распределения между ядрами процессора элементов массивов и обмен призводится с помощью MPI,
- витков циклов с помощью функций библиотеки OpenMP,
- которые в свою очередь распараллеливаются с помощью CUDA.

5.4. Память CUDA

В CUDA выделяют шесть видов памяти (рис.5.3.). Это регистры, локальная, глобальная, разделяемая, константная и текстурная память.

Такое обилие обусловлено спецификой видеокарты и первичным ее предназначением, а также стремлением разработчиков сделать систему как можно дешевле, жертвуя в различных случаях либо универсальностью, либо скоростью.

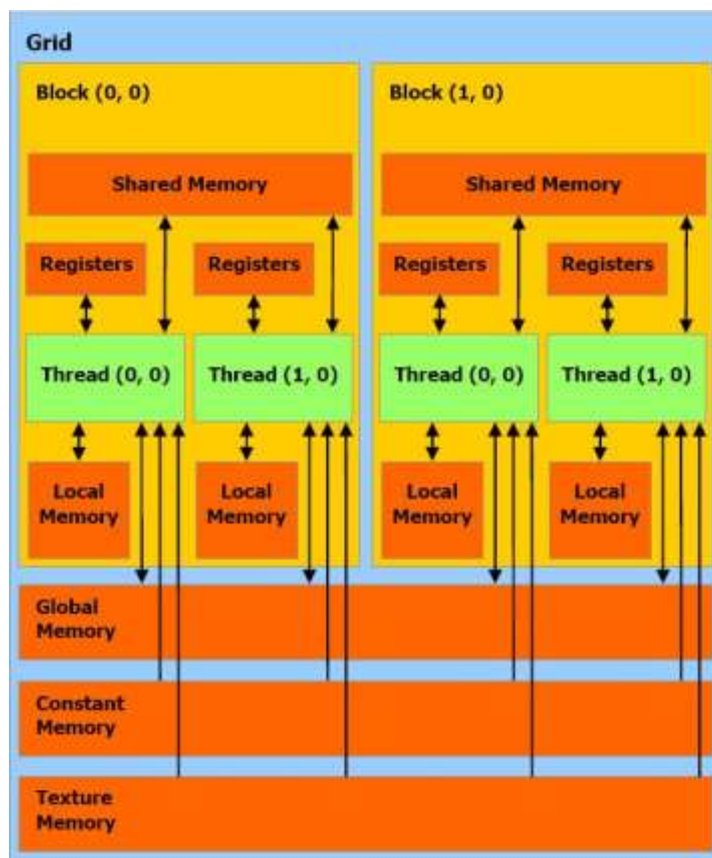


Рис.5.3. Виды памяти в CUDA

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 ограничивается 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 (до 4 Гбайт на графическом процессоре Tesla). Обладает высокой пропускной способностью, но очень большими задержками в несколько сот тактов. Не кэшируется, поддерживает обобщённые инструкции load и store, и обычные указатели на память.

Глобальная память необходима в основном для сохранения результатов работы программы перед отправкой их на хост (в CPU).

Регистровая память - наиболее простjq вид памяти. Каждый потоковый мультипроцессор содержит до 16 384 32-битовых регистров. 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 на чтение и на запись. Нить не имеет доступа к регистрам других нитей, но свои регистры доступны ей на протяжении выполнения данного ядра. Поскольку регистры расположены непосредственно в мультипроцессоре, то они обладают максимальной скоростью доступа.

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.

Память констант — доступна 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. 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.

5.5. Программирование на CUDA

CUDA - это С-подобный язык программирования со своим компилятором и библиотеками для вычислений на GPU. CUDA строится на концепции, что GPU выступает в роли сопроцессора к CPU. Программа на CUDA задействует как CPU, так и GPU.

CUDA работает на графических процессорах GeForce восьмого поколения и старше (серии GeForce 8, GeForce 9, GeForce 200).

Среда разработки CUDA (CUDA Toolkit) включает:

- **компилятор nvcc,**
- **профилировщик,**
- **отладчик gdb для GPU,**
- **CUDA runtime драйвер в комплекте стандартных драйверов NVIDIA,**
- **руководство по программированию,**
- **CUDA Developer SDK (исходный код, утилиты и документация).**

Имеется две библиотеки:

- **CUBLAS** — CUDA вариант BLAS (Basic Linear Algebra Subprograms).
- **CUFFT** — CUDA вариант библиотеки Fast Fourier Transform.

Компиляция и загрузка ядра на GPU

В систему устанавливается все необходимое для работы с CUDA, включая **runtime** и компилятор **nvcc**. Причем сам компилятор фактически представляет собой препроцессор, обрабатывающий исходник и строящий отдельный код для GPU и CPU. Для компиляции кода для CPU (включая код, необходимый для запуска ядра) nvcc использует обычный C/C++ компилятор.

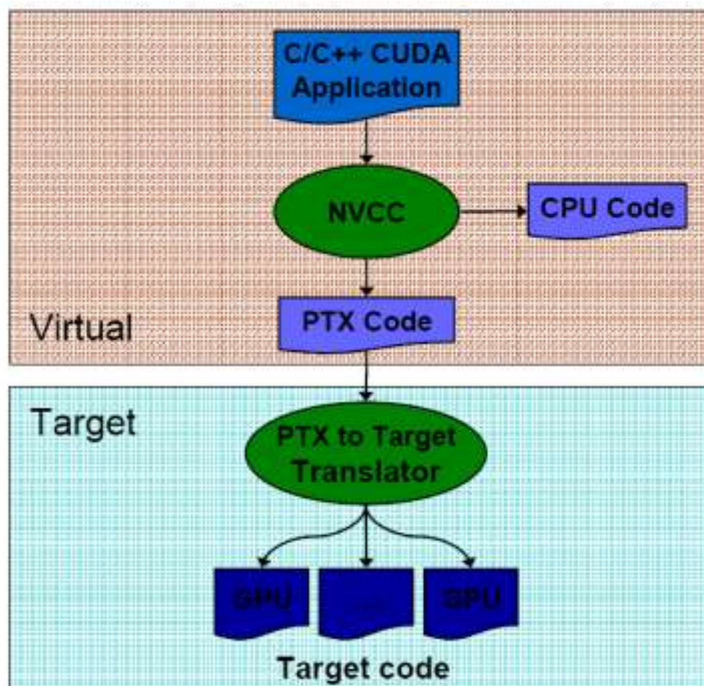
Процесс загрузки ядра на GPU можно описать следующим образом: исходник (txt) компилируется в объектник (object), затем один или несколько объектников линкуются в образ (image), который после этого загружается в модуль GPU (module), а уже из модуля можно получить указатель на точку входа ядра (по этому указателю мы сможем запустить ядро на исполнение).

В модели GPU рассматривается как специализированное вычислительное устройство (называемое device), которое:

- является сопроцессором к CPU (называемому host)
- обладает собственной памятью (DRAM)
- обладает возможностью параллельного выполнения огромного количества отдельных нитей (threads)

Нити GPU обладают крайне "небольшой стоимостью" - их создание и управление требует минимальных ресурсов (в отличии от CPU). Нити внутри блока могут взаимодействовать между собой (т.е. совместно решать подзадачу) через:

- общую память (shared memory)
- функцию синхронизации всех нитей блока (__synchronize)



Код, предназначенный для, сначала преобразовывается в промежуточный платформенно незевисимый язык PTX. Он подобен ассемблеру и позволяет изучать код в поисках потенциальных неэффективных участков. Наконец, последняя фаза заключается в компиляция PTX с помощью компилятора ptxas в код **cuubin**, двоичные версии которого размещаются по мультипроцессорам.

Процесс загрузки ядра на GPU можно описать в соответствием с деревом следующим образом: исходник (txt) компилируется в объектник (object), затем один или несколько объектников линкуются в образ (image), который после этого загружается в модуль GPU (module), а уже из модуля можно получить указатель на точку входа ядра (по этому указателю мы сможем запустить ядро на исполнение).

Эмуляция. Ключ компилятора "-deviceemu" позволяет компилировать и запускать прграмму на CPU, что позволяет отлаживать её с помощью обычного отладчика. Макрос __DEVICE_EMULATION__ позволяет осуществлять условную компилцию отладочной печати и т.п.. При эмуляции на каждый предполагаемый поток GPU создаётся поток CPU, а каждый поток требует 256KB стека! Эмуляция - это не полная симуляция, например, не могут быть найдены шибки синхронизации, точности вычислений, ошибки адресации.

Программы, скомпилированные в режиме эмуляции, можно запускать вне зависимости от наличия CUDA. Но в первый раз нужен nvcc, чтоб выполнить режим эмуляции

Адресация нитей. 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 threadldx и blockldx. 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.

Система команд ПП.Система команд ПП включает арифметические команды для вещественных и целочисленных вычислений с 32-разрядной точностью, команды управления (ветвления и циклы), а также команды обращения к памяти. Из-за высоких задержек команды доступа к оперативной памяти выполняются асинхронно. С целью сокрытия задержек в очереди выполнения GPU может одновременно находиться несколько сотен потоков, и если текущий поток блокируется по доступу к памяти, на исполнение ставится следующий. Поскольку контекст потока полностью хранится на регистрах графического процессора, переключение осуществляется за один такт. За переключение потоков отвечает диспетчер потоков, который не является программируемым.

Расширение языка С

Программы для CUDA (соответствующие файлы имеют расширение .cu) 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 сгене 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 staticпеременные внутри функции;
- не поддерживается переменное число входных аргументов.

Для задания размещения в памяти GPU переменных используются следующие спецификаторы device, constant и shared.

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).

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 );
```

Добавленные функции
CUDA поддерживает все математические функции из стандартной библиотеки языка С. Однако при этом следует иметь в виду, что большинство стандартных математических функций используют числа с двойной точностью (double). Однако, поскольку для современных GPU операции с double числами выполняются медленнее, чем операции с f1oat числами, то предпочтительнее там, где это

возможно, использовать f1oat аналоги стандартных функций. Так, f1oat – аналогом функции s i n является функция s i n f .

Кроме того, CUDA предоставляет также специальный набор функций пониженной точности, но обеспечивающих еще большее быстродействие. Таким аналогом для функции вычисления синуса является функция s i n f.

Программа сложения векторов в разделяемой памяти

```
     // Ядро, выполняется параллельно на большом числе нитей.
__global__ void sumKernel ( float * a, float * b, float * c )
{
     // Глобальный индекс нити.
  int idx = threadIdx.x + blockIdx.x * blockDim.x;

     // Выполнить обработку соответствующих данной нити данных.
  c [idx] = a [idx] + b [idx];
}

void sum ( float * a, float * b, float * c, int n )

{
  int numBytes = n * sizeof ( float );
  float * aDev = NULL;
  float * bDev = NULL;
  float * cDev = NULL;

     // Выделить память на GPU.
  cudaMalloc ( (void**)&aDev, numBytes );
  cudaMalloc ( (void**)&bDev, numBytes );
  cudaMalloc ( (void**)&cDev, numBytes );

     // Задать конфигурацию запуска n нитей.
  dim3 threads = dim3(512, 1);
  dim3 blocks  = dim3(n / threads.x, 1);

     // Скопировать входные данные из памяти CPU в память GPU.
  cudaMemcpy ( aDev, a, numBytes, cudaMemcpyHostToDevice );
  cudaMemcpy ( bDev, b, numBytes, cudaMemcpyHostToDevice );

     // Вызвать ядро с заданной конфигурацией для обработки данных.
  sumKernel<<<blocks, threads>>> (aDev, bDev, cDev);

     // Скопировать результаты в память CPU.
  cudaMemcpy ( c, cDev, numBytes, cudaMemcpyDeviceToHost );

     // Освободить выделенную память GPU.
  cudaFree ( aDev );
  cudaFree ( bDev );
  cudaFree ( cDev );
}
```

Очевидно, что программа содержит операции, выполняемые как в CPU так и в GPU, которые будут разделены компилятором **nvcc.**

В приведенном листинге первая функция (sumKernel) является ядром, она будет параллельно выполняться для каждого набора элементов а [i] , b [i] и c[i]. Спецификатор gobal используется для обозначения того, что это ядро, то есть функция, которая работает на GPU и которая может быть вызвана (точнее, запущена сразу на большом количестве нитей) **только с CPU**. Вначале функция sumKernel при помощи встроенных переменных вычисляет соответствующий данной нити глобальный индекс, для которого необходимо произвести сложение соответствующих элементов и записать результат.

Выполняемая на CPU функция sum осуществляет выделение памяти на GPU (поскольку GPU может непосредственно работать только со своей памятью), копирует входные данные из памяти CPU в выделенную память GPU, осуществляет запуск ядра (функции sumKernel), после чего копирует результат обратно в память CPU и освобождает выделенную память GPU.

Хотя для массивов, расположенных в памяти GPU, мы и используем обычные указатели, так же как и для данных, расположенных в памяти CPU, важно помнить о том, что CPU не может напрямую обращаться к памяти GPU по таким указателям. Вся работа с памятью GPU ведется CPU при помощи специальных функций.

Обратите внимание, что мы фактически для каждого допустимого индекса входных массивов запускаем отдельную нить для осуществления нужных вычислений. Все эти нити выполняются параллельно, и каждая нить может получить информацию о себе через встроенные переменные.

Важным моментом является то, что хотя подобный подход очень похож на работу с SIMD моделью, есть и принципиальные отличия (компания Nvidia использует термин SIMT (Single Instruction, Multiple Thread). Нити разбиваются на группы по 32 нити, называемые warp'ами. Только нити в пределах одного warp'a выполняются физически одновременно. Нити разных warp'ов могут находиться на разных стадиях выполнения программы. При этом управление warp'ми прозрачно осуществляет сам GPU.

Для решения задач CUDA использует очень большое количество паралельно выполняемых нитей, при этом обычно каждой нити соответствует один элемент вычисляемых данных.

Подобное разделение всех нитей является общим приемом использования CUDA: исходная задача разбивается на отдельные подзадачи, решаемые независимо друг от друга (рис. 2.2). Каждой такой подзадаче соответствует свой блок нитей. При этом каждая подзадача совместно решается всеми нитями своего блока.

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

Подобный подход является удачным компромиссом между необходимостью обеспечить взаимодействие нитей между собой и стоимостью обеспечения подобного взаимодействия, обеспечить возможность взаимодействия каждой нити с каждой было бы слишком сложно и дорого.

Существуют всего два механизма, при помощи которых нити внутри блока могут взаимодействовать друг с другом:

- разделяемая (shared) память;
- барьерная синхронизация.

Каждый блок получает в свое распоряжение определенный объем быстрой разделяемой памяти, которую все нити блока могут совместно использовать. Поскольку нити блока не обязательно выполняются физически параллельно (то есть мы имеем дело не с чистой SIMD архитектурой, а имеет место прозрачное управление нитями), то для того, чтобы не возникало проблем с одновременной работой с shared памятью, необходим некоторый механизм синхронизации нитей блока.

CUDA использует барьерную синхронизацию. Для ее осуществления вызывается встроенная функция syncthreads (), которая блокирует вызывающие нити блока до тех пор, пока все нити блока не войдут в эту функцию. Таким образом, при помощи syncthreads () мы можем организовать барьеры внутри ядра, гарантирующие, что если хотя бы одна нить прошла такой барьер, то не осталось ни одной, не прошедшей его.

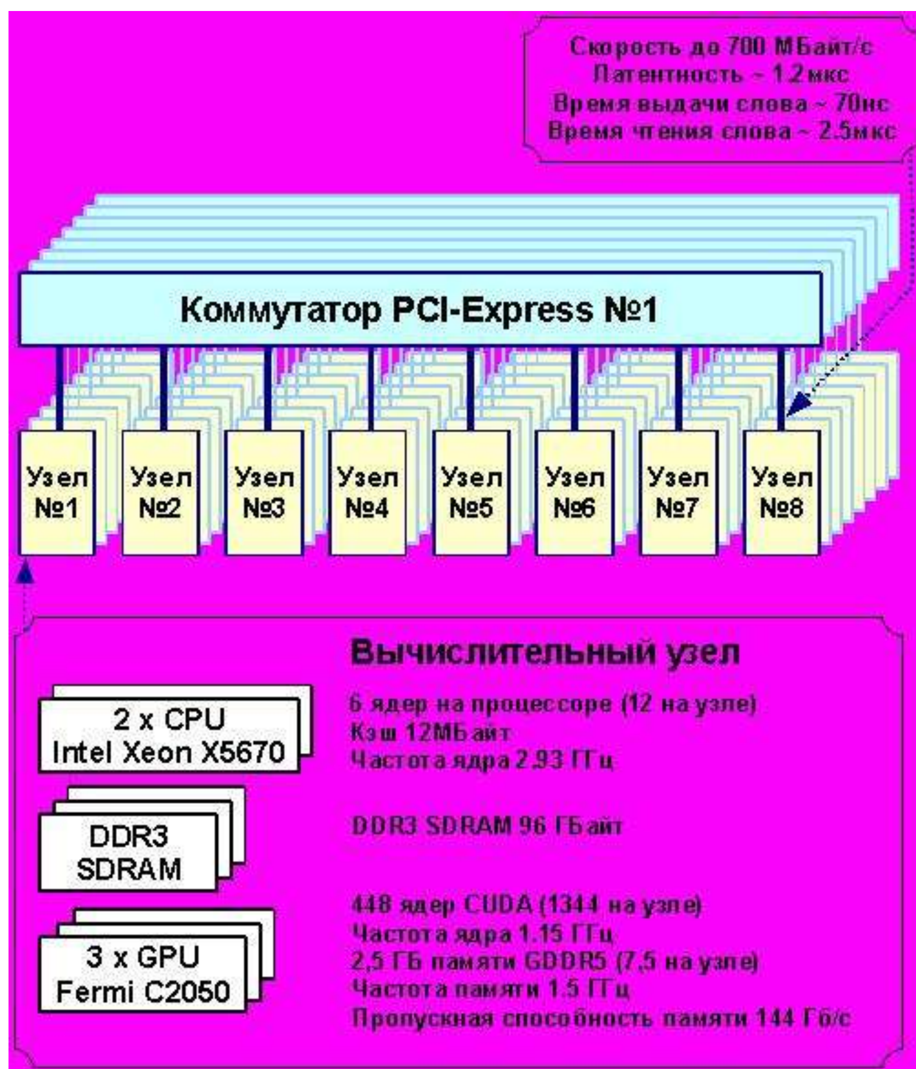
Основной процесс приложения CUDA работает на универсальном процессоре (host), он запускает несколько копий процессов kernel на видеокарте. Код для CPU делает следующее:

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

5.6. Гибридные вычислительные системы на основе GPU

На рисунке, взятом из [18], представлена конфигурация гибридной системы среднего быстродействия (около 230 Tflops). Вычислительный узел содержит стандартные для гибридной ЭВМ два CPU и три GPU с 1334 ядрами, которые и обеспечивают указанное быстродействие всей супер ЭВМ. Все вычислительные узлы объединяются через быстродействующий коммутатор. Эта конфигурация характерна для всех гибридных ЭВМ.



В гибридных ЭВМ, занимающих первые места в списке TOP500 число GPU приближается к 10 тысячам и достигнуто быстродействие в 2 Pflops.

Крупнейшие суперкомпьютерные центры мира начали активное внедрение решений на GPU. Согласно последней, 36-ой, редакции рейтинга TOP 500 самых быстрых в мире суперкомпьютеров, графические процессоры лежат в основе трех из пяти самых мощных систем.

Развитие GPU достигло той точки, когда множество существующих приложений реализуются на них и работают значительно быстрее, чем на многоядерных системах. Будущие вычислительные архитектуры будут представлять собой гибридные системы, которые буду состоять из GPU с параллельными ядрами и работающих с ними в тандеме многоядерных CPU!

В настоящее время большинство из российских кластеров прошли или проходят оснащение графическими ускорителями [19].

Рассмотрим простой пример получения программы для гибридной ЭВМ.

ЭТАП MPI всей задачи – Разбиение исходной программы 1 по ядрам. Это может быть узел с несколькими многоядерными процессорами (МЯП). Программа

на С пишется для всех ядер многоядерных процессоров (МЯП). Существуют следующие схемы обмена:

- Внутри МЯП – через общую память МЯП
- Между процессорами узла – через коммутатор узла
- Между узлами – через коммутатор всей системы.

Программа содержит для реализации обмена между узлами операторы MPI. Производится компиляция этой прораммы на С, отладка и выполнение на тестовом примере для заданного числа ядер. Для вычислений МЯП может считаться как 8 ядер (Nehalem), оставляя 8 АЛУ в каждом ядре для OpenMP. При этом для OpenMP игнорируется.

Продолжение программы осуществляется через операторы синхронизации окончания работы всех ядер.
ЭТАП OpenMP для одного ядра – на ядре находится программа 2, это сектор программы 1. OpenMP ничего не изменяет в программе, реагируя только на появление оператора Pragma, разделяя итерации цикла на на 8 АЛУ, компилирует, исполняет. Тут тоже нужна синхронизация. Теперь программа состоит из множества вложенных циклов с отметкой Pragma.
ЭТАП CUDA для одного АЛУ - это сектор программы 2. Здесь программирование выполняется с помощью CUDA. Для обмена между нитями существуют два механизма:

- разделяемая (shared) память;
- барьерная синхронизация.

Каждый блок получает в свое распоряжение определенный объем быстрой разделяемой памяти, которую все нити блока могут совместно использовать. Поскольку нити блока не обязательно выполняются физически параллельно (то есть мы имеем дело не с чистой SIMD архитектурой, а имеет место прозрачное управление нитями), то для того, чтобы не возникало проблем с одновременной работой с shared памятью, необходим некоторый механизм синхронизации нитей блока.

CUDA предлагает довольно простой способ синхронизации так называемая барьерная синхронизация. Для ее осуществления используется вызов встроенной функции syncthreads (), которая блокирует вызывающие нити блока до тех пор, пока все нити блока не войдут в эту функцию. Таким образом, при помощи syncthreads () мы можем организовать барьеры внутри ядра, гарантирующие, что если хотя бы одна нить прошла такой барьер, то не осталось ни одной за барьером (не прошедшей его).

Более подробно система программирования рассмотрен в [17], простой примере CUDA программы дан в [20], а более сложный - в [21].

ПРИЛОЖЕНИЕ 1.

Установка программного обеспечения и оборудования, компиляция и запуск параллельных программ.

MPI

- Антонов, Шпаковский, Серикова, Корнеев. Сборник «MPI»
- Сбитнев. cluster.linux-ekb.info/clusters.pdf
- Сухинов http://iproc.ru/programming/mpich-windows/
- Глызин (текст в конце этого «Приложения 1»)

OpenMP

- Сухинов. OpenMP — это не какая-то программа, которую можно «скачать». Это — интерфейс (набор функций и директив компилятора), который поддерживается или не поддерживается вашим компилятором. Visual C++ 2005 и выше поддерживает OpenMP в редакциях Professional
- Сухинов. http://iproc.ru/programming/openmp-visual-studio
- Антонов. OpenMP
- Глызин (текст в конце этого «Приложения 1»)

CUDA

- Установка CUDA: зайдите на http://www.nvidia.com/ object/ cuda get.htrnl и выберите свою операционную систему. Необходимо будет скачать и установить CUDA SDK и CUDA Toolkit. При этом устанавливается все, включая runtime и компилятор nvcc. Сам компилятор фактически представляет из себя препроцессор, обрабатывающий код и строящий отдельный код для GPU и CPU. Для компиляции кода для CPU (включая код, необходимый для запуска ядра) nvcc использует обычный C/C++-компилятор
- Боресков. steps3d.narod.ru/tutorials/**cuda**-tutorial.htm
- Bog BOS: Использование модели массового параллелизма CUDA
- Глызин. текст в конце этого «Приложения 1»

ГЛЫЗИН Д.С.

Инструкции по компиляции и запуску многопоточных программ в домашних условиях. Обновлено 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

ПРИЛОЖЕНИЕ 2. Программы матричные операции

1. Кластер
2. OpenMP
3. Графические процессоры

Кластер . 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.1), код главного процесса (рис.2) и код подчиненного процесса (рис.2).

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.1. Программа умножения матрицы на вектор: общая часть

Код главного процесса представлен на рис. 2. 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.2. 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 А, на запросы подчиненных процессов он отвечает сообщением с отметкой 0.

Код подчиненного процесса представлен на рис.3.

```
!     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.3. 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.

OpenMP. Умножение матриц в системах с общей памятью

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, реализующаю перемножение двух квадратных матриц, представлена на ниже (рис.4). 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.

```
#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 Перемножение матриц на языке Си.

Графические процессоры. Умножение матриц. Боресков

Рассмотрим использование разделяемой памяти на примере перемножения двух квадратных матриц A и B. Для вычисления подматрицы C' произведения A x B нам приходится постоянно обращаться к двух полосам 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. Но, если каждую из этих полос мы разобьем на квадраты 16x16, то становится видно, что результирующая матрица C просто является суммой попарных произведений подматриц из этих двух полос:

C’ == A1’ x B1’ + A2’ x B2’ + ... + AN’/16 x 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.

После этого считается произведение подматриц, загруженных в разделяему память, и суммируется нужный элемент произведения, и идет переход к следую щей паре подматриц. Соответствующая программа показана ниже.

```
#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;
}
```

Таким образом, при вычислении произведения матриц нам на каждый элемент произведения C нужно выполнить всего 2xN/16 чтений из глобальной памяти, в отличие от предыдущего варианта без использования глобальной памя-

ти, где нам требовалось на каждый элемент 2xN чтений. Количество арифметических операций не изменилось и осталось равным 2xN 1. В таблице приведено сравнение быстродействия этой версии и перемножения в лоб без использования разделяемой памяти.

Способ перемножения матриц	Время (миллисек)
«В лоб», без разделяемой памяти	2484.06
С использованием разделяемой памяти	133.47

Из таблицы видно, что быстродействие выросло больше чем на порядок.

В первом случае основное время было затрачено на чтение из глобальной памяти, причем почти все оно было uncoalesced, а на вычисления было затрачено менее одной десятой от времени чтения из глобальной памяти. Во втором случае свыше 80% времени было затрачено на вычисления, доступ к глобальной памяти занял менее 13%, и весь доступ был coalesced.

ПРИЛОЖЕНИЕ 3. Мнение http://gimlis.ru/2010/11/22/intel-vs-amdnvidia/

Область моей деятельности связана с вычислительной гидро- и аэродинамикой, поэтому я имею некоторое представление о научных вычислениях. Если вы думаете, что видео-чипы выйдут победителями сражения с Intel? Я думаю, что вы ошибаетесь.

Первое, я не знаю, что должно произойти, чтобы ученые взяли свои библиотеки написанные на фортране лет 10-20 назад и переписали под новые условия. Эти библиотеки написаны не на C++, и не на Java, и даже не на обычном C — они написаны на fortran-е, причем так, что напрямую на видеочипы не переносятся.

Второе, казалось бы, новые технологии должны заставить ученых шевелиться и переписывать, но там безумное количество формул, в которых каждый символ отлажен и выверен годами. При переносе появятся сразу ошибки, которые быстро не выявятся.

Третье, новые алгоритмы, оптимизированные под видео-чипы, требует математического доказательства. Учитывая методы выполнения расчетов на видеокартах, получается нетривиальная задача.

Четвертое, сейчас большинство используют MPI и OpenMP. При выходе даже 48-ядерного процессора от Intel — эти средства будут быстро оптимизированы под новые условия — старые программы продолжат свою работу.

Итого: на данный момент, из-за остановки роста производительности процессоров по частоте и смещения акцента в сторону многоядерности, видеочипы выглядят привлекательнее для научных разработок. Но я думаю пройдет пару лет, Intel выпустит свои многоядерники на рынок, доступный научным организациям и видео-чипы будут вытеснены. Но я не исключаю, что AMD/nVidia выпустят видеокарты с более дружественными технологиями, чем CUDA и openCL.

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. Параллельное программировани и аппаратура. Минск. 2010.
9. Процессор Itanium. http://ru.wikipedia.org/wiki/Itanium
10. MMX/ SSE - http://ru.wikipedia.org/wiki/SIMD
11. OpenMP. http://iproc.ru/programming/openmp-visual-studio
12. А.С. Антонов. Параллельное программирование с использованием технологии OpenMP. Изд. МГУ. 2009 г.
13. Сухинов. OpenMP. WINDOWS.
14. Метод Гаусса (LINPAK) решения СЛАУ на кластере www.cs.berkeley.edu/~demmel/cs267/lecture12/lecture12.html
15. Антонов, Шпаковский, Серикова, Корнеев. Сборник «MPI»
16. Боресков А. В., Харламов А. А. Основы работы с технологией CUDA. - 2010 г. 232 с.
17. Bog BOS: Использование модели массового параллелизма CUDA для разработки программ
18. Супер ЭВМ К100. http://www.kiam.ru/MVS/resourses/k100.html
19. .Программа «Униврситетский кластер» посвящена параллельным вычислениям в России
20. Два гиганта в одной программе — Nvidia CUDA и MPI
21. Десять простых шагов к прикладной программе для гибридных вычислений