Параллельное программирование

Преподает: Гуляев Олег Валерьевич

Число занятий: 13

Число лабораторных работ: 3+

План занятий:

1. Введение (виды параллельных вычислений, виды параллельных архитектур, классификация архитектур, законы Амдала)
2. MPI
3. OpenMP
4. Linda
5. CUDA
6. Дополнительная теория

Литература:

1. В. В. Воеводин, Вл. В. Воеводин. *Параллельные вычисления*.
2. parallel.ru
3. Интернет, programming guides (MPI, OpenMP, CUDA)

Закон Мура – эмпирическое наблюдение, изначально сделанное Гордоном Муром, согласно которому количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца. Также рассматривается интервал в 18 месяцев, связанный с прогнозом Давида Хауса из Intel, по мнению которого производительность процессоров должна удваиваться каждые 18 месяцев из-за сочетания роста количества транзисторов и быстродействия каждого из них.

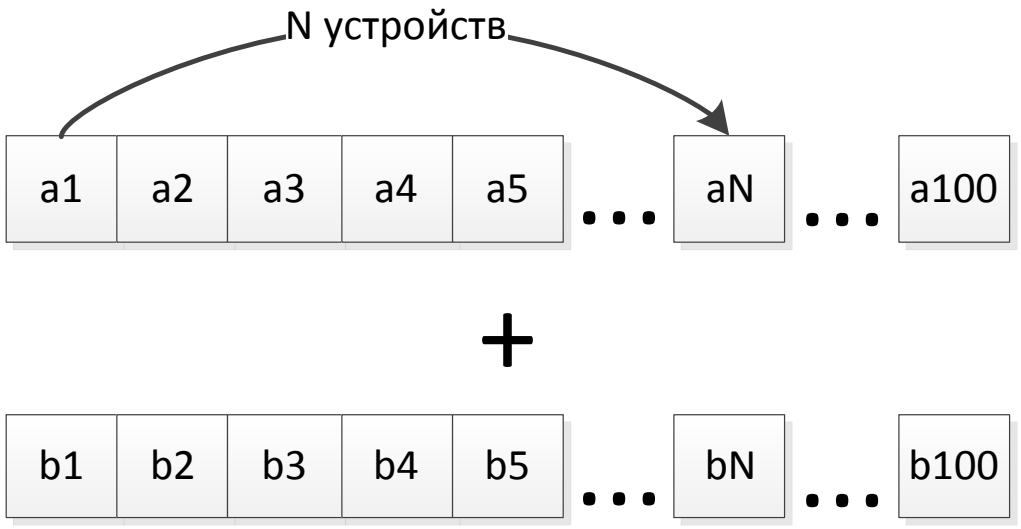
Пример:

	EDSAC *1949 год*		*HP Superdome* *2001 год*
тактовая частота	0,5 МГц	$1{,}5\times10^3$	770 МГц
производительность	10^2 оп/с	$1{,}9\times10^9$	$1{,}9\times10^{11}$ оп/с

Тактовая частота выросла на 3 порядка, а производительность на 9. Выигрыш в быстродействии достигается не только за счет тактовой частоты, но и за счет использования новых решений в архитектуре компьютеров, среди которых важное место занимает принцип параллельной обработки данных.

Параллельная обработка

Пусть нам нужно найти сумму двух векторов, состоящих из 100 чисел каждый с помощью устройства, которое выполняет суммирование за 5 тактов. В таком случае векторы будут сложены за 500 тактов.
Теперь предположим, что у нас в распоряжении N устройств, которые могут работать одновременно и независимо друг от друга. Постоянно загружая каждое устройство элементами векторов, сумму получаем уже за 500/N тактов.



Конвейерная обработка

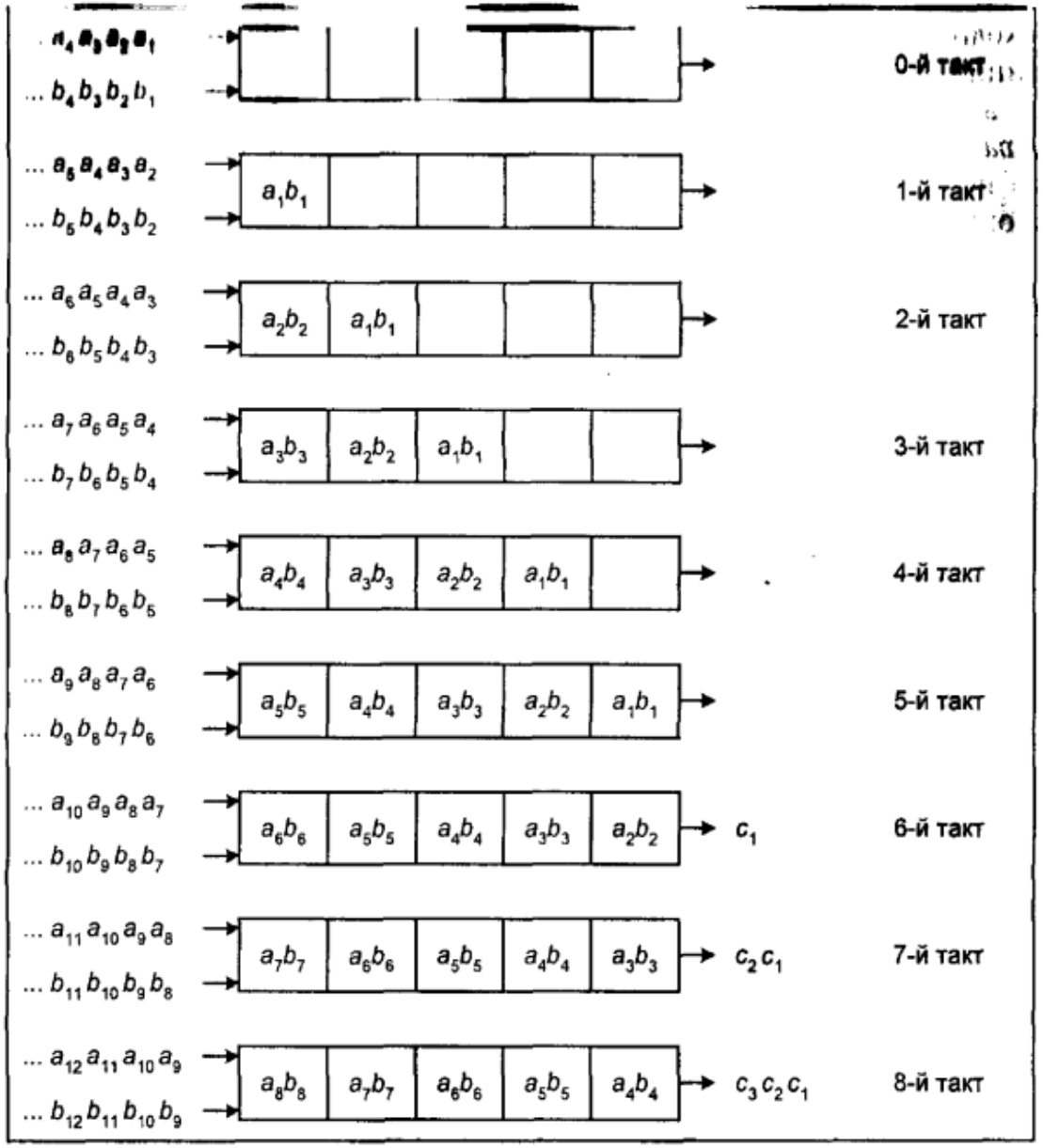
В предыдущем примере для выполнения одной операции сложения устройство блокировалось на 5 тактов. Но процесс обработки всех следующих элементов можно организовать более эффективно.

Рассмотрим операцию сложения вещественных чисел. Сложение каждой пары чисел выполняется в виде последовательность микроопераций – сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.д., причем микрооперация при завершении передает результаты следующей.

Каждую микрооперацию выделим в отдельную часть устройства и расположим их в порядке выполнения. В первый момент времени первая пара поступает для обработки первой частью. После обработки первой пары первая часть передает результаты второй части, а сама берет вторую пару, и т.д. Каждая часть устройства называется **ступенью конвейера**, а общее число ступеней – **длиной конвейера**.

Пусть устройство сложения вещественных чисел – конвейерное, имеющее длину 5. Одна операция сложения двух чисел выполнится за 5 тактов.
За счет одновременной обработки нескольких пар в любой момент времени на выполнение всей операции потребуется 104 такта – по сравнению с 500 тактами последовательного устройства выигрыш почти в 5 раз.

Общий случай – устройство содержит *l* ступеней, каждая срабатывает за один такт, время обработки *n* независимых операций этим устройством составит *l + n – 1* тактов, против *l * n* последовательного случая.



Виды архитектур

Повышение степени параллелизма в архитектуре компьютера ведет как к росту его пиковой производительности (поместили кучу процессоров), так и одновременно и к увеличению разрыва между производительностью пиковой и реальной.

Существует вариант разработки спецпроцессоров – некоторые операции могут быть реализованы аппаратно с помощью специальных машинных команд. Но алгоритмы разные, всего в архитектуре компьютера не предусмотришь, поэтому разработчикам железа необходимо искать компромисс между универсальностью и специализированностью.

ILP (Instruction-Level Parallelism) – параллелизм на уровне машинных команд, при котором последовательность инструкций одного потока выполнения может выполняться процессором одновременно. Существует два основных подхода к построению архитектуры процессоров, использующих ILP – суперскалярные процессоры и VLIW-процессоры. В обоих случаях предполагается, что процессор содержит несколько функциональных устройств, которые могут работать независимо друг от друга.

Суперскалярные процессоры не предполагают, что программа в терминах машинных команд будет включать в себя какую-то информацию о содержащемся в ней параллелизме, задача обнаружения параллелизма в машинном коде возлагается на аппаратуру, она же строит и последовательность выполнения команд. Планирование исполнения потока команд является динамическим.

VLIW-**процессоры** (Very Long Instruction Word) работают аналогично классическим, за исключением того, что команда, выдаваемая процессору на каждом цикле, определяет не одну операцию, а сразу несколько. Команда VLIW-процессора состоит из набора полей, каждое из которых отвечает за свою операцию, например, за активизацию функциональных устройств, работу с памятью, операции с регистрами и т.д. Если какая-то часть процессора на данном этапе выполнения программы не востребована, то соответствующее поле команды не задействуется. Для VLIW-процессора компилятор сам выявляет параллелизм в программе и явно сообщает аппаратуре, какие операции не зависят друг от друга.

Multicore (многоядерность) – несколько ядер процессора на одной микросхеме выполняют потоки исполнения параллельно.

Hyperthreading - в процессорах с использованием этой технологии каждый физический процессор может хранить состояние сразу двух потоков, что для операционной системы выглядит как наличие двух логических процессоров (англ. Logical processor).
Когда при исполнении потока одним из логических процессоров возникает пауза (в результате кэш-промаха, ошибки предсказания ветвлений, ожидания результата предыдущей инструкции), то управление передаётся потоку в другом логическом процессоре. Таким образом, пока один процесс ждёт, например, данные из памяти, вычислительные ресурсы физического процессора используются для обработки другого процесса.
Поток в одном ядре занимает только часть функциональных устройств, остальные могут быть переданы в использование другому потоку.

Многопроцессорные конфигурации

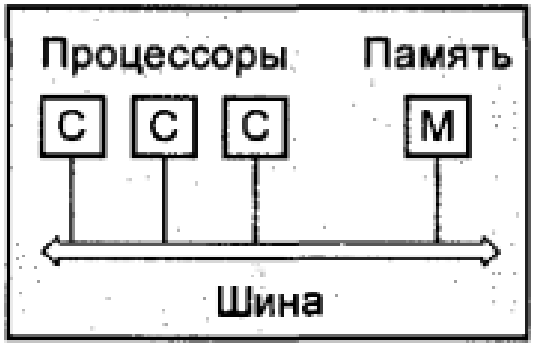
Ранее мы рассматривали принципы увеличения производительности отдельных процессоров, на основе которых могут строиться многопроцессорные конфигурации.

Компьютеры с общей памятью (SMP, Symmetric Multi Processors или Shared Memory Processors) – в системе присутствует несколько равноправных процессоров, имеющих одинаковый доступ к единой памяти. Все процессоры работают с единым адресным пространством.

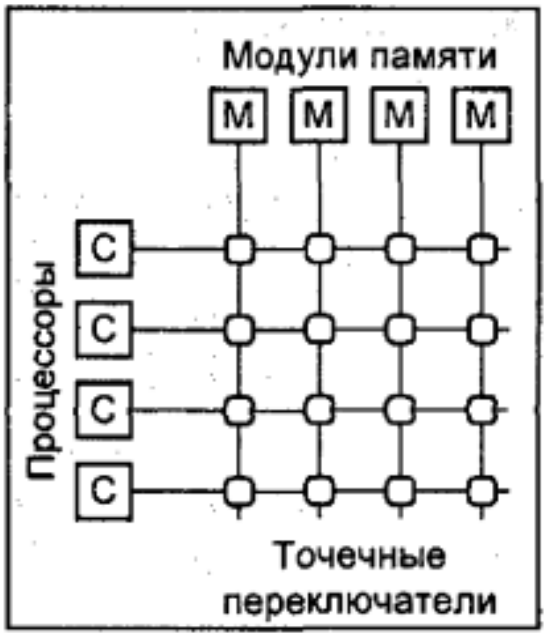
Компьютеры с распределенной памятью – каждый вычислительный узел является полноценным компьютером со своим процессором, памятью, подсистемой ввода/вывода, операционной системой и т.д. Интернет можно рассматривать как гигантский параллельный компьютер с распределенной памятью

Способ программирования систем с распределенной памятью – обмен сообщениями, например, с помощью PVM или MPI. Разработка программного обеспечения проще для SMP, и меньше накладные расходы на обмен данными между процессорами, однако по технологическим причинам не удается объединить большое число процессоров с единой памятью.

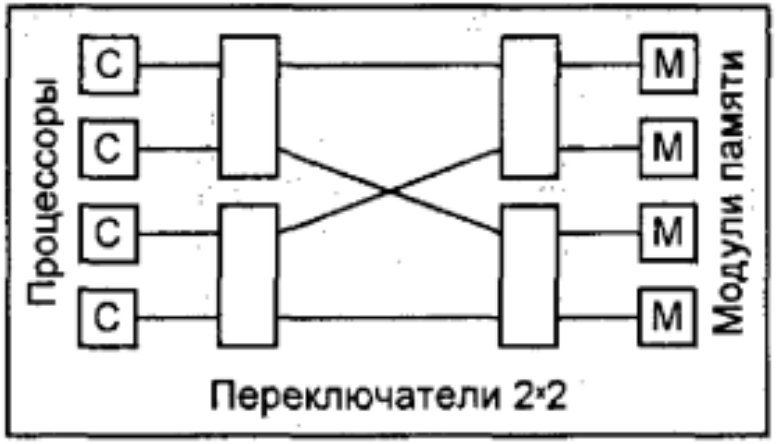
Самый простой способ организации SMP – использование **общей шины**. Чтобы предотвратить одновременной обращение нескольких процессоров к памяти, используются различные схемы арбитража, гарантирующие монопольное владение шиной захватившим ее устройством, однако при таком подходе добавление даже небольшого числа устройств на шине быстро делает ее узким местом, выдающим задержки при работе с памятью.



Для построения более мощных систем применяются другие подходы, например, **матричный коммутатор** – память разделяется на независимые модули, переключатели разрешают или запрещают передачу информации между процессорами и модулями памяти. Появляется возможность одновременной работы процессоров с различными модулями памяти. Недостаток – большой объем необходимого оборудования, например для связи *n* процессоров с *n* модулями памяти требуется n^2 переключателей.



Еще одни подход – **каскадные переключатели**, в которых каждый используемый коммутатор может соединить любой из своих входов с любым выходом, это позволяет любому процессору обращаться к любому модулю памяти. В общем случае для *n* процессоров и *n* модулей памяти требуется $\log_2 n$ каскадов по *n*/2 коммутаторов в каждом, т.е. всего $\frac{n}{2}\log_2 n$ коммутаторов. Оборудование дешевле, однако появляются задержки в связи со срабатываниями коммутаторов.



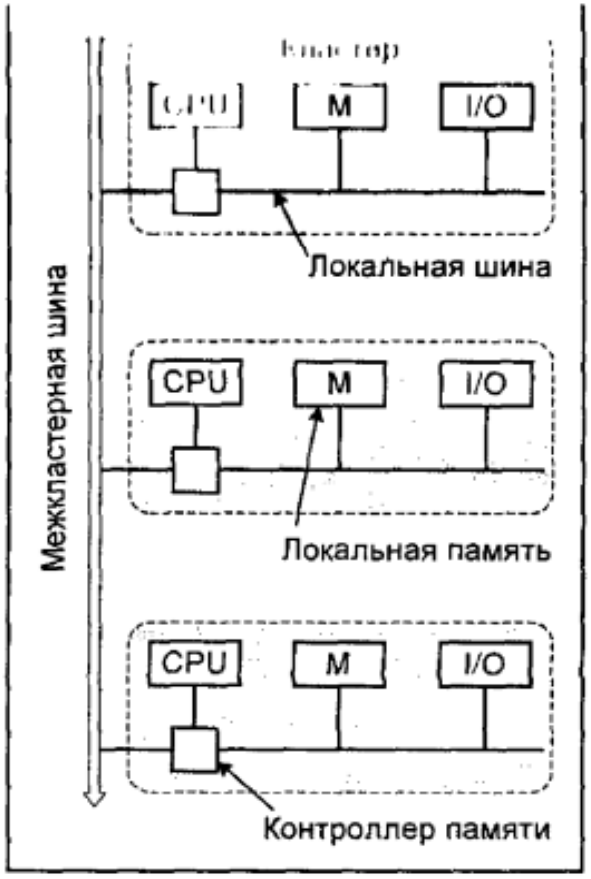
Достоинства обоих классов – SMP и систем с распределенной памятью – объединяются в архитектуре **NUMA** (Non Uniform Memory Access, неоднородный доступ к памяти). Адресное пространство у всех узлов едино, однако доступ к своей памяти быстрее, чем к памяти другого узла.

Рассмотрим NUMA на примере системы Cm*, разработанной в конце 70-х годов прошлого века.

Данный компьютер состоит из набора кластеров, состоящих из процессора, локальной памяти, контроллера памяти, и, возможно, некоторых устройств ввода/вывода, которые соединены между собой локальной шиной.

Когда процессору нужно выполнить операцию чтения/записи, он посылает запрос с адресом контроллеру памяти, который по старшим разрядам адреса анализирует, в каком модуле памяти хранятся нужные данные – локальном или у удаленного кластера.

Если адрес локальный, запрос выставляется на локальную шину, иначе на межкластерную шину.



Проблема архитектура NUMA – когерентность кешей. Пусть процессор 1 сохранил значение *x* в ячейке *q*, и затем процессор 2 хочет прочитать содержимое ячейки *q*. Если *x* при записи попал в кеш процессора 1, возникает проблема когерентности кешей. Архитектура **ccNUMA** (cache-coherent NUMA) решает эти проблемы аппаратно.

Конфигурации SMP серверов могут содержать порядка 16-32-64 процессоров, тогда как их расширения с архитектурой ccNUMA уже объединяют до 256 процессоров и больше.

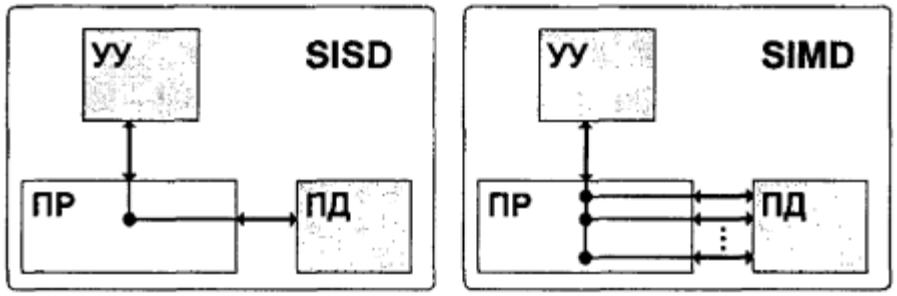
Классификация архитектур

Классификация Флинна (M. Flynn)

Основывается на понятии **потока**, под которым понимается последовательность команд или данных. На основе чисел потоков команд и данных Флинн выделил четыре класса архитектур.

SISD (Single instruction stream/Single data stream) – классические последовательные машины типа PDP 11 или VAX 11/780. Команды обрабатываются последовательно друг за другом и каждая команда инициирует одну скалярную операцию.

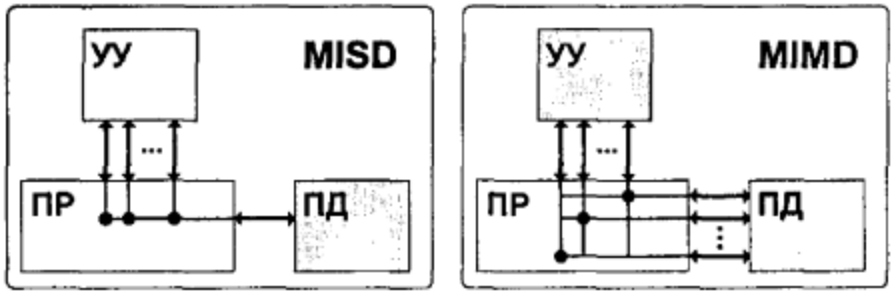
SIMD (Single instruction stream/Multiple data stream) – поток команд теперь включает в себя векторные команды, что позволяет выполнить арифметическую операцию над многими данными. ILLIAC IV, Cray 1.



(ПР – один или несколько процессорных элементов, УУ – устройство управления, ПД – память данных)

MISD (Multiple instruction stream/ Single data stream) – наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Можно считать, что данный класс пуст.

MIMD (Multiple instruction stream/ Multiple data stream) – в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных.



Отношение конкретных машин к конкретному классу сильно зависит от точки зрения исследователя.
Так, конвейерные машины могут быть отнесены и к классу SISD (конвейер – единое устройство), и к классу SIMD (векторный поток данных при работе с конвейером), и к классу MISD (множество ступеней конвейера обрабатывают один поток данных последовательно), и к классу MIMD (выполнение различных ступеней над множественным скалярным потоком данных).

Недостаток этой классификации – перегруженность класса MIMD.

Классификация Хокни (R. Hockney) MIMD архитектур.

Множественный поток команд может быть обработан двумя способами – одним **конвейерным** устройством обработки, работающем в режиме разделения времени для отдельных потоков, либо каждый поток обрабатывается своим собственным устройством (переключаемые, сети).

Переключаемые MIMD – возможна связь каждого процессора с каждым, реализуемая с помощью переключателя, может быть два случая: вся память распределена среди процессоров как их локальная память – **распределенная память**, либо память – разделяемый ресурс, доступный всем процессорам через переключатель, система с **общей памятью**.

Для **сетевых** MIMD предполагается, что все они имеют распределенную память, а дальнейшая классификация проводится в соответствии с топологией сети.


MIMD
переключаемые
конвейерные
сети
с общей памятью
с распределенной памятью
регулярные решетки
гиперкубы
иерархические структуры
изменяющие конфигурацию

Классификация Фенга (T. Feng).

n – число бит, обрабатываемых параллельно при выполнении машинных инструкций.
m – число слов, обрабатываемых одновременно данной вычислительной системой.
Каждая вычислительная система описывается парой (n, m).

Разрядно-последовательные (n = m = 1). В каждый момент времени обрабатывается один двоичный разряд – давняя система MINIMA.

Разрядно-параллельные пословно-последовательные (n > 1, m = 1). Большинство классических последовательных компьютеров: IBM 701 (36, 1), PDP-11 (16, 1).

Разрядно-последовательные пословно-параллельные (n = 1, m > 1). Такие вычислительные системы состоят из большого числа одноразрядных процессорных элементов, каждый из которых может независимо от других обрабатывать свои данные: STARAN (1, 256), MPP (1, 16384), SOLOMON (1, 1024).

Разрядно-параллельные пословно-параллельные (n > 1, m > 1). Большинство параллельных вычислительных систем, обрабатывают одновременно mn двоичных разрядов: ILLIAC IV (64, 64), TI ASC (64, 32), CDC 6600 (60, 10).
Недостаток: непонятно, за счет чего компьютер может обрабатывать более одного слова – из-за множества ФУ, их конвейерности или за счет нескольких независимых процессоров. Так, если в системе N независимых процессоров, каждый из которых имеет по F конвейерных ФУ с длинами конвейеров L, то m = N*F*L.

Классификация по способу взаимодействия процессоров с памятью

- SMP
- Распределенная память
- NUMA (ccNUMA)

Классификация Хендлера (W. Handler)

Рассматривается три уровня обработки данных в процессе выполнения программ:

1. Уровень выполнения программы – УУ производит выборку и дешифрацию команд программы.

2. Уровень выполнения команд – АЛУ исполняет команду, выданную ей устройством управления.

3. Уровень битовой обработки – все ЭЛС (элементарные логические схемы) процессора разбиваются на группы, необходимые для выполнения операций над одним двоичным разрядом.

Есть несколько процессоров, каждый со своим УУ. Каждое УУ связано с несколькими АЛУ. Каждое АЛУ объединяет несколько групп ЭЛС. Число групп ЭЛС есть не что иное, как длина машинного слова.

Без рассмотрения возможностей конвейеризации – k устройств управления, d арифметико-логических устройств и w групп ЭЛС в каждом АЛУ составят тройку t(C) = (k, d, w) для описания данной вычислительной системы C.

Примеры:

t(MINIMA) = (1,1,1)
t(IBM 701) = (1,1,36)
t(SOLOMON) = (1,1024,1)
t(ILLIAC) = (1,64,64)
t(BBN Butterfly GP1000) = (256,1,32)

Теперь допустим возможность конвейеризации на каждом уровне.

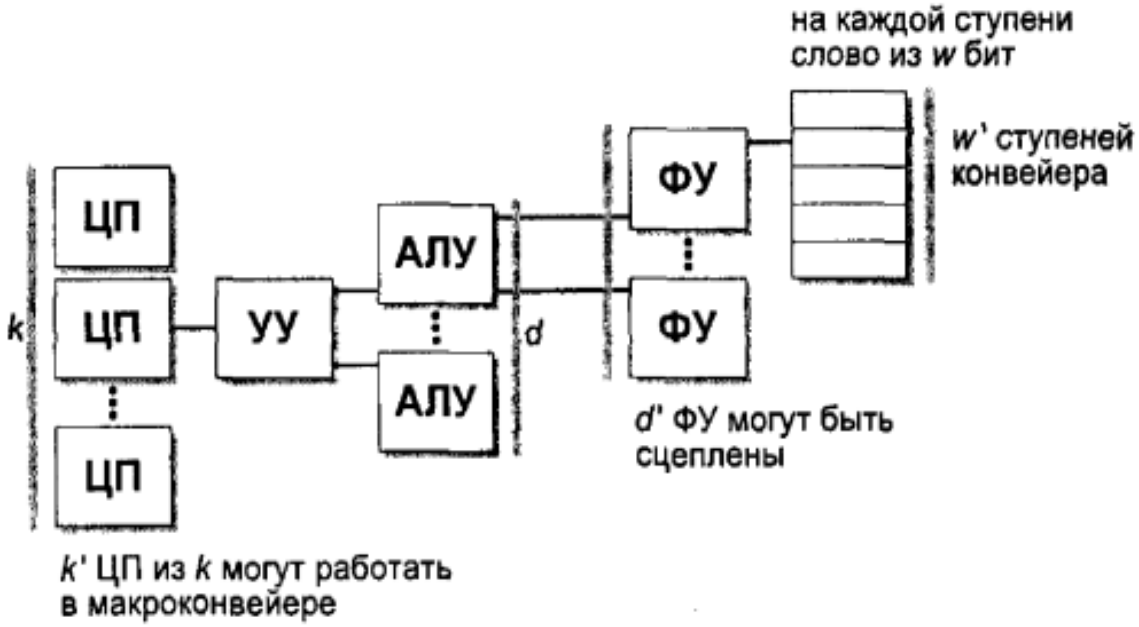
На уровне ЭЛС конвейерность – это конвейерность ФУ. Если ФУ обрабатывает w разрядные слова на каждой из w’ ступеней конвейера, то для характеристика параллелизма будет писать w x w’. Компьютер TI ASC имеет четыре конвейерных устройства по восемь ступеней каждое для обработки 64-разрядных слов, следовательно, может быть описан как t(TI ASC) = (1,4,64x8).

Следующий уровень - конвейеризация на уровне команд – в системе есть несколько ФУ, которые могут работать одновременно в режиме конвейера (зацепление ФУ). Компьютер CDC 6600 содержит 10 независимых последовательных ФУ, способных подавать результат своей работы на вход другим ФУ: t(CDC 6600) = (1,1x10,60).

Самый верхний уровень – макроконвейер – поток данных, проходя через один процессор, поступает на вход другому. Компьютер PEPE, имея три независимых системы из 288 устройств, описывается как t(PEPE) = (1x3,288,32).

С учетом конвейерности тройка выглядит так:
t(C) = (k x k’, d x d’, w x w’), где:

- k – число процессоров (каждый со своим УУ), работающих параллельно.
- k’ – число ступеней макроконвейера из отдельных процессоров
- d – число АЛУ в каждом процессоре, работающих параллельно
- d’ – число ступеней конвейера из функциональных устройств АЛУ
- w – число разрядов в слове, обрабатываемых АЛУ одновременно
- w’ – число ступеней в конвейере функциональных устройств АЛУ



Законы Амдала

Любая вычислительная система – совокупность функциональных устройств (ФУ). Содержательная часть функций устройств нас не интересует.

Реальная производительность r системы устройств - количество операций, реально выполненных в среднем за единицу времени.

Пиковая производительность π системы устройств – максимальное количество операций, которое может быть выполнено той же системой за единицу времени при отсутствии связей между ФУ.

Стоимость операции – время ее реализации, **стоимость работы** – сумма стоимостей всех выполненных операций. **Загруженность устройства** p на данном отрезке времени – отношение стоимости реально выполненной работы к максимально возможной стоимости. $p \in [0, 1]$

Утверждение: пусть система состоит из *s* устройств с пиковыми производительностями $\pi_1, \dots, \pi_s$ и работают с загруженностями $p_1, \dots, p_s$. В этом случае реальная производительность *r* системы выражается формулой (*)

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

Пусть алгоритм реализуется на системе из *s* устройств с пиковыми производительностями $\pi_1, \dots, \pi_s$, причем $\pi_1 \leq \dots \leq \pi_s$. При реализации алгоритма система достигает реальной производительности *r*. Сравним скорость работы системы со скоростью работы устройства с производительностью π_s.
$R = \frac{r}{\pi_s}$ – **ускорение** реализации алгоритма.
Используя (*), имеем

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

Анализ этого выражения показывает, что ускорение системы, состоящей из *s* устройств, никогда не превосходит *s* и может достигать *s* в том и только том случае, если все устройства системы имеют одинаковые пиковые производительности и полностью загружены.

Следствие (**): пусть система состоит из *s* устройств одинаковой пиковой производительности, тогда ускорение системы равно сумме загруженностей всех устройств.

Рассмотрим систему из *s* устройств. Построим ориентированный мультиграф, в котором вершины – ФУ, а дуги – связи между ними, т.е. передача аргумента от одного ФУ к другому. Этот мультиграф называется **графом системы**.

Утверждение: пусть система состоит из *s* ФУ с пиковыми производительностями $\pi_1, \dots, \pi_s$. Если граф системы связный, то максимальная производительность системы выражается формулой

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

Следствие (1-ый закон Амдала): производительность вычислительной системы, состоящей из связанных между собой устройств, в общем случае определяется самым непроизводительным ее устройством.

Асимптотическая производительность системы будет максимальной, если все устройства имеют одинаковые пиковые производительности.

Пусть на системе устройств решается алгоритм, в котором всего выполняется *N* операций, *n* из которых могут выполняться только последовательно.

$\beta = \frac{n}{N}$ – доля последовательных вычислений.

2-ой закон Амдала: пусть система состоит из *s* простых универсальных устройств. Предположим, что при выполнении параллельной части алгоритма все *s* устройств загружены полностью. Тогда максимально возможное ускорение равно

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

Доказательство: обозначим через π производительность отдельного ФУ, согласно (**) $R = \sum_{i=1}^{s} p_i$. Если всего выполняется *N* операций, то βN операций выполняется последовательно и $(1-\beta)N$ параллельно на *s* устройствах по $\frac{(1-\beta)N}{s}$ операций на каждом. Пусть все последовательные вычисления выполняются на первом ФУ. Всего алгоритм реализуется за время

$$T_1 = \frac{\beta N + \frac{(1-\beta)N}{s}}{\pi}$$

На параллельной части алгоритма работают как первое, так и все остальные устройства, тратя на это время ($2 \le i \le s$)

$$T_i = \frac{\frac{(1-\beta)N}{s}}{\pi}$$

$p_1 = 1, p_i = \frac{T_i}{T_1} = \frac{(1-\beta)N/s}{\beta N + (1-\beta)N/s}$

Следовательно, $R = 1 + \sum_{i=2}^{s} \frac{(1-\beta)N/s}{\beta N + (1-\beta)N/s}$

Следствие (3-ий закон Амдала): пусть система состоит из простых одинаковых универсальных устройств. При любом режиме работы ее ускорение не может превзойти обратной величины доли последовательных вычислений.

p - % кода, который может быть выполнен ||

Что лучше взять:
1-core CPU, 5 op/s per core
10-core CPU, 1 op/s per core

Пусть нужно выполнить 100 инструкций

1) **p = 0**
1-core: 100 / 5 = 20s
10-core: 100 / 1 = 100s

2) **p = 1**
1-core: 100 / 5 = 20s
10-core:
s = 1 / (1 - 1 + 1/10) = 10
100 / 10 = 10s

3) **p = 0.5**
1-core: 100 / 5 = 20s
10-core:
s = 1 / (1 - 0.5 + 0.5/10) ~ 1.82
100 / 1.82 ~ 55s

4) **p = 0.9**
1-core: 100 / 5 = 20s
10-core:
s = 1 / (1 - 0.9 + 0.9/10) ~ 5.26
100 / 5.26 ~ 19s

MPI

MPI – message passing interface, интерфейс передачи сообщений.
MPICH – реализация MPI, с которой мы будем работать.

Особенности:

- Модель организации вычислений – «одна программа – множество процессов». Под параллельной программой понимается множество параллельных процессов. Любой процесс порождается на основе одного и того же программного кода.
- **Коммуникатор** – группа процессов, которая используется при выполнении операции передачи данных. Существует коммуникатор MPI_COMM_WORLD, включающий в себя все процессы приложения. Каждый процесс имеет номер (ранг) в пределах некоторого коммуникатора. Процесс может входить в разные коммуникаторы под разными номерами, поэтому он имеет два атрибута - <коммуникатор, номер>.
- Основным способом общения процессов является посылка сообщений. Сообщение – набор данных некоторого типа, имеющее несколько атрибутов – номер процесса-отправителя, номер процесса-получателя, идентификатор сообщения и т.д. Функции передачи сообщений бывают двух видов: point-to-point и коллективные.
- При выполнении операции передачи сообщений нужно указывать тип данных, которые передаются: MPI_INT, MPI_FLOAT, MPI_CHAR, … (полный список в mpi.h)

MPI рассчитан на гетерогенную систему, на разных узлах могут различаться типы данных. MPI берет на себя их преобразование. Также MPI позволяет создавать собственные типы – структуры.

В MPI 1.1 число процессов задается при запуске программы и процессе выполнения измениться не может. В MPI 2 предусмотрено изменение числа процессов.

Преимущества MPI:

- Стандартизация – MPI единственная библиотека обмена сообщениями, которая может быть рассмотрена как стандарт. Поддерживается практически на всех HPC (High-perfomance computing) платформах.

- Кроссплатформенность – нет нужды модифицировать код при его переносе на другую платформу, поддерживающую MPI.

- Производительность – реализации MPI используют нативные особенности аппаратных средств для оптимизации производительности.

- Функциональность – свыше 115 функций.

- Доступность – доступно множество реализаций.

Общие функции

```
MPI_Init(int *argc, char *argv[])
```

Инициализация параллельной части программы, все другие функции MPI могут быть вызваны только после **MPI_Init**.

```
MPI_Finalize(void)
```

Заключение параллельной части приложения. Все последующие обращения к функциям MPI запрещены. К моменту вызова **MPI_Finalize** каждым процессом все действия, требующие его участия в обмене сообщениями, должны быть завершены.

```
MPI_Comm_size(MPI_comm comm, int *size)
```

comm – идентификатор коммуникатора
OUT size – число процессов в коммуникаторе comm
Определение общего числа параллельных процессов в коммуникаторе **comm**.

```
MPI_Comm_rank(MPI_comm comm, int *rank)
```

comm – идентификатор коммуникатора
OUT rank – номер процесса в коммуникаторе **comm**
Определение номера процесса в коммуникаторе **comm**.

Парные передачи сообщений

Все функции данной группы делятся на два класса: функции с блокировкой (с синхронизацией) и без блокировки (асинхронные).

Функции с блокировкой

```
MPI_Send(void *buf, int count, MPI_Data datatype, int dest, int msgtag, MPI_Comm comm)
```

buf – адрес начала буфера с посылаемым сообщением
count – число передаваемых элементов в сообщении
datatype – тип передаваемых элементов
dest – номер процесса-получателя
msgtag – идентификатор сообщения
comm – идентификатор коммуникатора

Блокирующая посылка сообщения с идентификатором **msgtag**, состоящего из **count** элементов типа **datatype**, процессу с номером **dest**. Все элементы посылаемого сообщения расположены подряд в буфере **buf**.
Блокировка гарантирует корректность повторного использования всех параметров после вызова функции – передаваемое сообщение не испортится.

В момент завершения **MPI_Send** состояние пересылаемого сообщения может быть различным:

- сообщение находится в процессе отправителя
- сообщение находится в процессе передачи
- сообщение хранится в процессе получателя
- сообщение принято функцией **MPI_Recv** получателя

Для специальных режимов посылки предусмотрены еще три функции (параметры те же):

MPI_BSend(…) – передача сообщения с буферизацией. Если прием посылаемого сообщения еще не был инициализирован процессом-получателем, то сообщение будет записано в буфер и произойдет немедленный возврат из функции.

MPI_SSend(…) – передача сообщения с синхронизацией. Выход из функции произойдет только тогда, когда прием посылаемого сообщения будет инициализирован процессом-получателем.

MPI_RSend(…) – передача сообщения по готовности. Данной функцией можно пользоваться только в том случае, если процесс-получатель уже инициализировал прием сообщения, в противном случае вызов функции считается ошибочным и результат ее выполнения не определен.

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype, int
source, int msgtag, MPI_Comm comm, MPI_Status *status)
```

OUT buf – адрес начала буфера для приема сообщения
count – максимальное число элементов в принимаемом сообщении
datatype – тип элементов принимаемого сообщения
source – номер процесса-отправителя
msgtag – идентификатор принимаемого сообщения
comm – идентификатор коммуникатора
OUT status – параметры принятого сообщения

Прием сообщения с идентификатором **msgtag** от процесса **source** с блокировкой. Блокировка гарантирует, что после возврата из функции все элементы сообщения уже будут приняты и расположены в буфере **buf**.
Примечание: **source** может быть **MPI_ANY_SOURCE**, что означает готовность получить сообщение от любого источника, аналогично **tag** может быть **MPI_ANY_TAG**.

```
MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int
*count)
```

status – параметры принятого сообщения
datatype – тип элементов принятого сообщения
OUT count – число элементов сообщения

По значению параметра **status** данная функция определяет число уже принятых (после обращения к **MPI_Recv**) элементов сообщения типа **datatype**.

Пример программы, в которой нулевой процесс посылает сообщение процессу с номером один и ждет от него ответа.

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int numtasks, rank, dest, source, rc, count;
    int tag = 1;
    char inmsg, outmsg = 'x';
    MPI_Status Stat;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
    {
        dest = 1;
        source = 1;
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
                      tag, MPI_COMM_WORLD);
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
                      tag, MPI_COMM_WORLD, &Stat);
    }

   // продолжение на следующем слайде
```

```
else if (rank == 1)
    {
        dest = 0;
        source = 0;
        rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source,
                      tag, MPI_COMM_WORLD, &Stat);
        rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest,
                      tag, MPI_COMM_WORLD);
    }

    rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf("Task %d: Received %d char(s) from task \
           %d with tag %d \n", rank, count,
           Stat.MPI_SOURCE, Stat.MPI_TAG);

    MPI_Finalize();
}
```

Функции без блокировки

Возврат из функций данной группы происходит сразу без какой-либо блокировки процессов.

```
MPI_ISend(void *buf, int count, MPI_Data datatype, int dest, int msgtag, MPI_Comm comm, MPI_Request *request)
```

OUT request – идентификатор асинхронной операции.
Определить момент, когда можно повторно использовать буфер **buf** без опасения испортить передаваемое сообщение, можно с помощью параметра request и функций **MPI_Wait** и **MPI_Test**.
Аналогично модификациям **MPI_Send**, предусмотрены варианты **MPI_Ibsend**, **MPI_Issend**, **MPI_Irsend**.

```
MPI_IRecv(void *buf, int count, MPI_Datatype datatype, int
source, int msgtag, MPI_Comm comm, MPI_Request *request)
```

Прием сообщения, аналогичный **MPI_Recv**, однако с возвратом сразу после инициализации приема без ожидания получения и записи всего сообщения в буфер **buf**.
Примечание: Сообщение, отправленное любой из функций **MPI_Send**, **MPI_Isend** и любой из их модификаций, может быть принято любой из функций **MPI_Recv** и **MPI_Irecv**.

```
MPI_Wait(MPI_Request *request, MPI_Status *status)
```

request – идентификатор операции асинхронного приема или передачи
OUT status – параметры сообщения
Ожидание завершения асинхронной операции с идентификатором **request** – пока асинхронная операция не будет завершена, процесс, выполнивший функцию **MPI_Wait**, будет заблокирован. Если завершен прием, атрибуты и длину полученного сообщения можно определить с помощью параметра **status**.

```
MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

request – идентификатор операции асинхронного приема или передачи
OUT flag – признак завершенности операции обмена
OUT status – параметры сообщения
Проверка завершенности асинхронных функций, ассоциированных с идентификатором **request**, в параметре **flag** возвращается значение 1, если операция завершена, и 0 в противном случае. Если завершен прием, атрибуты и длину полученного сообщения можно определить с помощью параметра **status**.

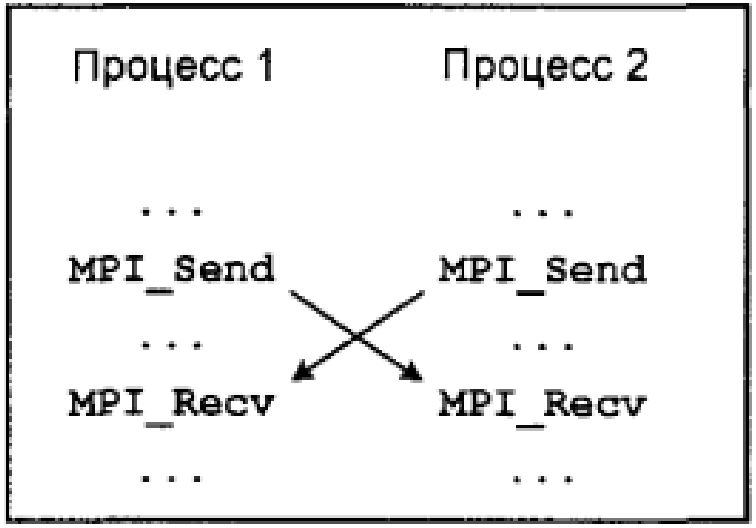
Также существуют функции **MPI_Waitall**, **MPI_Waitany**, **MPI_Waitsome**, **MPI_Testall**, **MPI_Testany**, в которых указываются несколько идентификаторов асинхронных операций.

Одно из достоинств асинхронных функций – возможность скрыть операции приема/передачи сообщений на фоне полезных вычислений.

```
MPI_Irecv(…, &request)
// принимаемое сообщение пока что не нужно
…
// полезные вычисления
…
MPI_Test(&request, …)
```

Ситуация тупика (deadlock)

Первый процесс не может вернуться из функции посылки, поскольку второй не начинает прием сообщения. А второй процесс не может начать прием сообщения, поскольку сам застрял по той же причине.



Первый вариант устранения – использование асинхронного приема сообщения. У процесса 2 изменяем код на:

```
…
MPI_Irecv
…
MPI_Send
…
```

Второй вариант – использование функции совмещенного приема и передачи сообщений.

```
MPI_Sendrecv(void *sbuf, int scount, MPI_Datatype stype, int
dest, int stag, void *rbuf, int rcount, MPI_Datatype rtype, int
source, int rtag, MPI_Comm comm, MPI_Status *status)
```

sbuf – адрес начала буфера с посылаемым сообщением
scount – число передаваемых элементов в сообщении
stype – тип передаваемых элементов
dest – номер процесса-получателя
stag – идентификатор посылаемого сообщения
OUT rbuf – адрес начала буфера приема сообщения
rcount – число принимаемых элементов сообщения
rtype – типа принимаемых элементов
source – номер процесса-отправителя
rtag – идентификатор принимаемого сообщения
comm – идентификатор коммуникатора
OUT status – параметры принятого сообщения

Посылка и прием сообщений объединены, отсутствие тупиков гарантировано. Сообщение, отправленное **MPI_Sendrecv** может быть принято обычным образом, точно так же **MPI_Sendrecv** может принять сообщение, отправленное обычным образом (**MPI_Send**).

Коллективное взаимодействие процессов

В операциях коллективного взаимодействия процессов участвуют все процессы коммуникатора. Как и для блокирующих функций, возврат означает то, что разрешен свободный доступ к буферу приема или посылки. Асинхронных коллективных операций в MPI нет. Синхронизация процессов с помощью коллективных операций не осуществляется – например, если какой-то процесс уже завершил свое участие в коллективной операции, это не означает то, данная операция завершена другими процессами или вообще начата ими.

```
MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int
source, MPI_Comm comm)
```

OUT buf – адрес начала буфера посылки сообщения
count – число передаваемых элементов в сообщении
datatype – тип передаваемых элементов
source – номер рассылающего процесса
comm – идентификатор коммуникатора
Рассылка сообщения от процесса **source** всем процессам, включая рассылающий процесс. При возврате из функции содержимое буфера **buf** процесса **source** будет скопировано в локальный буфер каждого процесса коммуникатора **comm**.

`MPI_Bcast(array, 100, MPI_INT, 0, comm)` – первые сто целых чисел из массива **array** нулевого процесса будут скопированы в локальные буфера **array** каждого процесса.

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

sbuf – адрес начала буфера посылки
scount – число элементов в посылаемом сообщении
stype – тип элементов отсылаемого сообщения
OUT rbuf – адрес начала буфера сборки данных
rcount – число элементов в принимаемом сообщении
rtype – тип элементов принимаемого сообщения
dest – номер процесса, на котором происходит сборка данных
comm – идентификатор коммуникатора
Сборка данных со всех процессов в буфере **rbuf** процесса **dest**. Каждый процесс, включая **dest**, посылает содержимое своего буфера **sbuf** процессу **dest**, собирающий процесс сохраняет данные в **rbuf**, располагая их в порядке возрастания номеров процессов. Параметр **rcount** обозначает число элементов, принимаемых не от всех процессов в сумме, а от каждого процесса. С помощью похожей функции **MPI_Gatherv** можно принимать от процессов массивы данных различной длины.

```
MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype, void
*rbuf, int rcount, MPI_Datatype rtype, int source, MPI_Comm
comm)
```

sbuf – адрес начала буфера посылки
scount – число элементов в посылаемом сообщении
stype – тип элементов отсылаемого сообщения
OUT rbuf – адрес начала буфера сборки данных
rcount – число элементов в принимаемом сообщении
rtype – тип элементов принимаемого сообщения
source – номер процесса, который рассылает данные
comm – идентификатор коммуникатора
По действию обратна **MPI_Gather** – процесс **source** рассылает порции данных из массива **sbuf** всем остальным процессам. Можно считать, что массив **sbuf** делится на n равных частей (n – число процессов коммуникатора), состоящих из **scount** элементов типа **stype**, после чего i-ая часть посылается i-му процессу.

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

sbuf – адрес начала буфера для аргументов
OUT rbuf – адрес начала буфера для результата
count – число аргументов у каждого процесса
datatype – тип аргументов
op – идентификатор глобальной операции
root – процесс-получатель результата
comm – идентификатор коммуникатора
Выполняется **count** операций **op**. В буфере **sbuf** каждого процесса располагается count аргументов типа **datatype**. Первые элементы массивов **sbuf** участвуют в первой операции **op**, вторые – во второй и т.д. Результаты всех **count** операций записываются в буфер **rbuf** процесса **root**.

Операции с группами процессов и коммуникаторами

Группа – упорядоченное множество процессов. Каждому процессу в группе сопоставлено целое число – номер. **MPI_GROUP_EMPTY** – пустая группа, не содержащая не одного процесса.
Новые группы можно создавать на основе существующих групп или на основе коммуникаторов. Базовая группа создается на основе **MPI_COMM_WORLD**.

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Получение группы **group**, соответствующей коммуникатору comm.

```
MPI_Group_rank(MPI_Group group, int *rank)
```

Определение номера процесса **rank** в группе **group**. Если вызвавший процесс не входит в группу **group**, то возвращается значение **MPI_UNDEFINED**.

```
MPI_Group_size(MPI_Group group, int  *size)
```

Определение количества **size** процессов в группе **group**.

```
MPI_Group_translate_ranks(MPI_Group group1, int count,
int *ranks1, MPI_Group group2, int *ranks2)
```

В массиве **ranks2** возвращаются ранги в группе **group2** процессов с рангами **ranks1** в группе **group1**. Параметр **count** задает число процессов, для которых нужно определить ранги.

```
// half1_group и half2_group – равные по размеру
// непересекающиеся группы, образованные из world_group.
//             world_group
// номера      0 1 2 3 4 5
//             half1|half2
// номера      0 1 2|0 1 2
// узнаем номером процесса world_rank из группы world_group,
// соответствующий процессу с номером half_rank из группы
half1_group

MPI_Group_translate_ranks(half2_group, 1, &half_rank,
world_group, &world_rank)
```

```
MPI_Group_incl(MPI_Group group, int count, int *ranks,
MPI_Group *new_group)
```

group – группа, на основе которой создается новая
count – число процессов в новой группе
ranks – номера процессов, которые войдут в новую группу
new_group – новая группа
Создание группы **new_group** из **count** процессов группы **group** с номерами **ranks**.

```
MPI_Group_excl(MPI_Group group, int count, int *ranks,
MPI_Group *new_group)
```

То же самое, но исключая процессы с номерами **ranks**.

```
MPI_Group_intersection(MPI_Group group1, MPI_Group
group2, MPI_Group *new_group)
```

Создание группы **new_group** из пересечения групп **group1** и **group2**.
Процессы группы упорядочены как в группе **group1**.
Аналогично **MPI_Group_union** и **MPI_Group_difference**.

```
MPI_Group_free(MPI_Group *group)
```

Уничтожение группы **group**, после этого переменная **group** примет значение **MPI_GROUP_NULL**.

Коммуникаторы, создающиеся после вызова **MPI_Init**:

MPI_COMM_WORLD – коммуникатор, объединяющий все процессы приложения
MPI_COMM_NULL – значение, используемое для ошибочного коммуникатора
MPI_COMM_SELF – коммуникатор, включающий только что вызванный процесс

```
MPI_Comm_dup(MPI_Comm comm, MPI_Comm *new_comm)
```

Создание нового коммуникатора **new_comm** с той же группой процессов, что и у **comm**.

```
MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *new_comm)
```

Создание нового коммуникатора **new_comm** из коммуникатора **comm** для группы процессов **group**, которая должна быть подмножеством группы, связанной с коммуникатором **comm**.

```
MPI_Comm_split(MPI_Comm comm, int color, int key,
MPI_Comm *new_comm)
```

Разбиение коммуникатора **comm** на несколько новых коммуникаторов по числу значений параметра **color** – в один коммуникатор попадают процессы с одним значением **color**. Процессы с большим значением параметра **key** получат больший номер в новом коммуникаторе. Процессы, которые не должны войти в новые коммуникаторы, указывают в качестве цвета **MPI_UNDEFINED**, им в параметре **new_comm** возвращается **MPI_COMM_NULL**.

```
MPI_Comm_split(MPI_COMM_WORLD, rank % 3, rank, &new_comm)
```

В этом примере коммуникатор **MPI_COMM_WORLD** разбивается на три части, в первую войдут процессы с номерами 0, 3, 6, …, во вторую - 1, 4, 7, …, и в третью – 2, 5, 8, … Задание в качестве параметра **key** номера процесса в коммуникаторе **MPI_COMM_WORLD** гарантирует, что порядок нумерации процессов в создаваемых коммуникаторах соответствует порядку нумерации в исходном коммуникаторе.

```
MPI_Comm_free(MPI_Comm *comm)
```

Удаление коммуникатора **comm**. После выполнения этой функции переменной **comm** присваивается значение **MPI_COMM_NULL**. Если с этим коммуникатором к моменту вызова функции уже выполняется какая-то операция, то она будет завершена.

Производные типы данных

Производные типы данных создаются во время выполнения программы с помощью функций-конструкторов на основе существующих типов данных. Создание типа состоит из двух этапов:

1. Конструирование типа
2. Регистрация типа

После регистрации новый тип можно использовать в операциях пересылки.

```
MPI_Type_struct(int count, int *blocklens, int *offsets,
MPI_Datatype *types, MPI_Datatype *newtype)
```

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

```
MPI_Type_extent(MPI_Datatype datatype, int *extent)
```

Определение диапозона **extent** (разницы между верхней и нижней границами элемента данного типа) типа данных **datatype** в байтах.

```
MPI_Type_commit(MPI_Datatype *datatype)
```

Регистрация созданного типа данных **datatype**.

Пример: создание типа, состоящего из одиночных int и char.

```
MPI_Datatype newType;
MPI_Datatype types[2] = { MPI_INT, MPI_CHAR };

int blockLens[2] = { 1, 1 };

int intExt = MPI_Type_extent( MPI_INT, &intExt );
int offsets[2] = { 0, intExt };

MPI_Type_struct( 2, blockLens, offsets, types, &newType );
MPI_Type_commit(&newType);
```

Виртуальные топологии

Топология – это механизм сопоставления процессам некоторого коммуникатора альтернативной схемы адресации. В MPI топологии виртуальны, т.е. они не связаны с физической топологией коммуникационной сети. Топология используется программистом для более удобного обозначения процессов, и, таким образом, приближения параллельной программы к структуре математического алгоритма.

В MPI предусмотрены два типа топологий:
1. Декартова топология (прямоугольная решетка произвольной размерности)
2. Топология графа

```
MPI_Topo_test(MPI_Comm comm, int *type)
```

Процедура определения типа топологии, связанной с коммуникатором **comm**. Возможные возвращаемые значения параметра **type**:

- **MPI_GRAPH** для топологии графа.
- **MPI_CART** для декартовой топологии.
- **MPI_UNDEFINED** – с коммуникатором **comm** не связана никакая топология.

Декартова топология

Обобщением линейной и матричной топологий на произвольное число измерений является декартова топология. Для создания коммуникатора с декартовой топологией используется функция **MPI_Cart_create**. С помощью этой функции можно создавать топологии с произвольным числом измерений, причем по каждому измерению в отдельности можно накладывать периодические граничные условия. Таким образом, для одномерной топологии мы можем получить или линейную структуру, или кольцо в зависимости от того, какие граничные условия будут наложены. Для двумерной топологии, соответственно, либо прямоугольник, либо цилиндр, либо тор.

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
int *periods, int reorder, MPI_Comm *comm_cart)
```

comm_old – родительский коммуникатор
ndims – число измерений
dims – массив размера **ndims**, в котором задается число процессов вдоль каждого измерения
periods – логический массив размера **ndims** для задания граничных условий (true - периодические, false - непериодические)
reorder - логическая переменная, указывает, производить перенумерацию процессов (true) или нет (false)
OUT comm_cart - новый коммуникатор
Функция является коллективной, т.е. должна запускаться на всех процессах, входящих в группу коммуникатора **comm_old**. При этом, если какие-то процессы не попадают в новую группу, то для них возвращается результат **MPI_COMM_NULL**. В случае, когда размеры заказываемой сетки больше имеющегося в группе числа процессов, функция завершается аварийно. Значение параметра **reorder** = false означает, что идентификаторы всех процессов в новой группе будут такими же, как в старой группе. Если **reorder** = true, то MPI будет пытаться перенумеровать их с целью оптимизации коммуникаций.

```
MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
```

comm – коммуникатор с декартовой топологией
coords – координаты в декартовой системе
OUT rank – номер процесса
Функция получения номера процесса по координатам **coords**.

```
MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int
*coords)
```

comm – коммуникатор с декартовой топологией
rank – номер процесса
ndim – число измерений
OUT coords – координаты процесса в декартовой топологии
Функция определения координат процесса по его номеру.

```
MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm
*newcomm)
```

comm – коммуникатор с декартовой топологией
remain_dims – логический массив размера **ndims**, указывающий, входит ли i-е измерение в новую подрешетку (**remain_dims[i]** = true)
OUT newcomm – новый коммуникатор, описывающий подрешетку, содержащую вызывающий процесс.

Функция выделения подпространства в декартовой топологии. Функция является коллективной. Действие функции проиллюстрируем следующим примером. Предположим, что имеется декартова решетка 2 x 3 x 4, тогда обращение к функции **MPI_Cart_sub** с массивом **remain_dims** (true, false, true) создаст три коммуникатора с топологией 2 x 4. Каждый из коммуникаторов будет описывать область связи, состоящую из 1/3 процессов, входивших в исходную область связи.

Топология графа

```
MPI_Graph_create(MPI_Comm comm_old, int nnodes, int
*index, int *edges, int reorder, MPI_Comm *comm_graph)
```

Создание на основе коммуникатора **comm_old** нового коммуникатора **comm_graph** с топологией графа. Параметр **nnodes** задает число вершин графа, **index[i]** содержит суммарное количество соседей для первых **i** вершин. Массив **edges** содержит упорядоченный список номеров процессов-соседей всех вершин.
Функция является коллективной, т.е. должна запускаться на всех процессах, входящих в группу коммуникатора **comm_old**. При этом, если **nnodes** меньше числа процессов коммуникатора **comm_old**, то какие-то процессы не попадают в новую группу, для них возвращается результат **MPI_COMM_NULL**. Если **nnodes** больше числа процессов коммуникатора **comm_old**, функция завершается аварийно. Значение параметра **reorder** = false означает, что идентификаторы всех процессов в новой группе будут такими же, как в старой группе. Если **reorder** = true, то MPI будет пытаться перенумеровать их с целью оптимизации коммуникаций.

Пример графа:

Процесс	Соседи
0	1, 3
1	0
2	3
3	0, 2

Заполнение массивов:

```
index = {2, 3, 4, 6}
edges = {1, 3, 0, 3, 0, 2}
```

Прочие функции

```
MPI_Barrier(MPI_Comm comm)
```

comm – идентификатор коммуникатора
Функция блокирует работу вызвавших ее процессов до тех пор, пока все оставшиеся процессы коммуникатора **comm** также не выполнят эту функцию.

```
MPI_Get_processor_name(char *name, int *max_size)
```

OUT name – название узла, на котором запущен процесс
max_size – максимальный размер для названия
Получить название узла **name**. В константе **MPI_MAX_PROCESSOR_NAME** содержится максимальный размер для названия узла.

Установка MPI и настройка проекта

1. Устанавливаем «mpich.nt.1.2.5» на все узлы вашего кластера (в нашем случае – один узел).

2. На каждом узле: Пуск -> Все программы -> MPICH -> mpd -> MPICH Configuration tool
select -> add всех узлов, на которых будут работать ваши приложения.

3. Создаем студийный проект.

4. Project -> project properties (ALT+F7). Прописываем:
   Configuration properties -> C/C++ -> General -> Additional include directories: C:\Program Files\MPICH\SDK\Include
   Configuration properties ->Linker -> General -> Additional library directories: C:\Program Files\MPICH\SDK\Lib
   Configuration properties ->Linker -> Input -> Additional dependencies: odbc32.lib odbccp32.lib ws2_32.lib mpichd.lib

5. Для Release версии в последней строке предыдущего пункта заменяем mpichd.lib на mpich.lib.

6. Простая программа для проверки корректности установки и работы MPICH: нулевой процесс посылает сообщение процессу с номером один и ждет от него ответа.

7. Компилируем. Полученный exe кладем в папку, например C:\test

8. Запускаем 2 приветствующих друг друга процесса из консоли:

```
cd “C:\Program Files\MPICH\mpd\bin”
MPIRun.exe -np 2 c:\test\pp.exe
Task 0: Received 1 char(s) from task 1 with tag 1
Task 1: Received 1 char(s) from task 0 with tag 1.
```

Лабораторная работа

1. Создать коммуникатор, в котором нумерация процессов будет вестись в обратном порядке по сравнению с коммуникатором MPI_COMM_WORLD и напечатать ранги процессов в обоих коммуникаторах.

2. Создать две непересекающихся группы процессов и организовать обмен сообщениями через коммуникатор MPI_COMM_WORLD процессов с одинаковым рангом в этих группах (с использованием MPI_Group_translate_ranks).

3. Разбить все процессы приложения на три произвольных группы и напечатать ранги в MPI_COMM_WORLD тех процессов, что попали в первые две группы, но не попали в третью.

4. Переслать нулевому процессу от всех процессов приложения структуру (структуру создать на уровне MPI), состоящую из ранга процесса и названия узла, на котором данный процесс запущен (полученного с помощью MPI_GET_PROCESSOR_NAME).

5. Сравнить эффективность реализации функции MPI_SENDRECV с моделированием той же функциональности при помощи неблокирующих операций.

6. Смоделировать глобальное суммирование методом сдваивания и сравнить эффективность такой реализации с функцией MPI_Reduce.
Примечание: для более адекватного сравнения суммировать большие массивы

7. Сравнить эффективность широковещательной посылки данных с их эквивалентом из нескольких пересылок типа точка-точка.

8. Использовать двумерную декартову топологию процессов при реализации параллельного перемножения матриц.
Каждый процесс считает элемент результирующей матрицы.

Замер времени при помощи rdtsc

rdtsc (Read Time Stamp Counter) – ассемблерная инструкция для платформы x86, читающая счетчик TSC и возвращающая его в регистрах EDX:EAX 64-битное количество тактов с момента последнего сброса процессора.
Реализация на C:

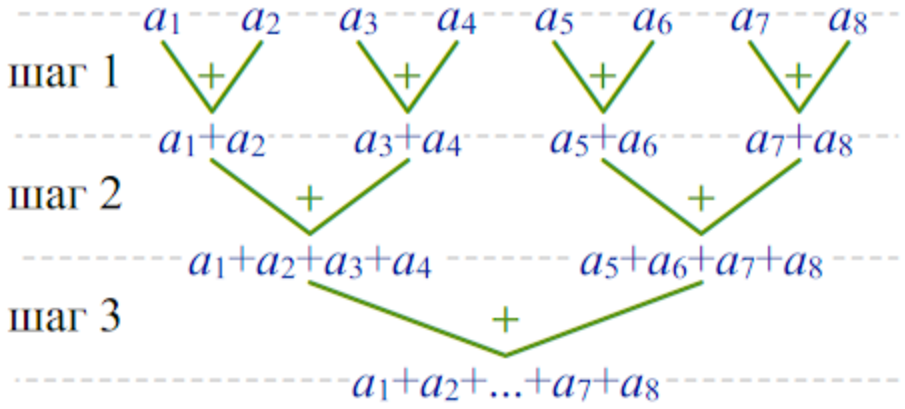
```
__int64 rdtsc()
{
    __int64 res;
    __asm
    {
        rdtsc
        mov    dword ptr [res], eax
        mov    dword ptr [res + 4], edx
    }
    return res;
}
```

Использование:

```
__int64 start =  rdtsc();
// код, время которого замеряем
__int64 finish = rdtsc();
printf("elapsed time: %lld", finish – start);
```

Суммирование методом сдваивания

Для n = 2^q чисел алгоритм сдваивания состоит q = log2(n) этапов; на первом этапе выполняются n/2 сложений, на втором — n/4, и так далее, пока на последнем этапе не будет выполнено единственное сложение.



Intel Itanium

Архитектура – **EPIC**: Explicity Parallel Instruction Computing.
Компилятор явным образом говорит процессору, какие команды могут быть выполнены параллельно, то есть специфицирует, на каких устройствах может быть выполнена команда.

VLIW	**SuperScalar**
Инструкции имеют одинаковую длину и запакованы в очень большие слова	Инструкции имеют переменную длину и никак не запакованы
Статическое планирование инструкций	Динамическое планирование инструкций
Нет переименования регистров	Есть переименование регистров
Статическая привязка инструкций к исполняющему устройству	Динамическая привязка инструкций к исполняющему устройству
Бинарная несовместимость кода для разных версий процессора – система команд связана с архитектурой	Четкое разделение между набором команд и микроархитектурой, т.е. бинарная совместимость

(серым – как работает EPIC)

Переименование регистров - техника, используемая некоторыми процессорами, чтобы убрать зависимости между различными частями кода. Пример:

```
MOV EAX, [MEM1]
IMUL EAX, 6
MOV [MEM2], EAX
MOV EAX, [MEM3]
INC EAX
MOV [MEM4], EAX
```

Здесь последние три инструкции независимы от трех первых в том смысле, что им не требуется результат, полученный после их выполнения. Чтобы оптимизировать этот код на ранних процессорах вы были должны использовать другой регистр, отличный от EAX, в последних трех инструкциях и перегруппировать инструкции так, чтобы последние три выполнялись параллельно с первыми тремя.
Процессоры, поддерживающие переименование регистров, делают это автоматически. Они назначают новый временный регистр для EAX каждый раз, когда вы пишете в него. Таким образом инструкции 'MOV EAX, [MEM3]' становятся независимыми от предшествующих инструкций. С помощью выполнения не по порядку 'MOV [MEM4], EAX' может выполниться до того окончания обработки медленной инструкции IMUL.

VLIW: компилятор указывает, на каких устройствах вычислять инструкцию.
SuperScalar: указывается, что должно быть вычислено
EPIC: указывается, как может быть вычислена инструкция.

Особенности архитектуры EPIC:
- Бандлы (bundles)
- Спекуляция
- Предикация
- Программная конвейеризация циклов (SWP)

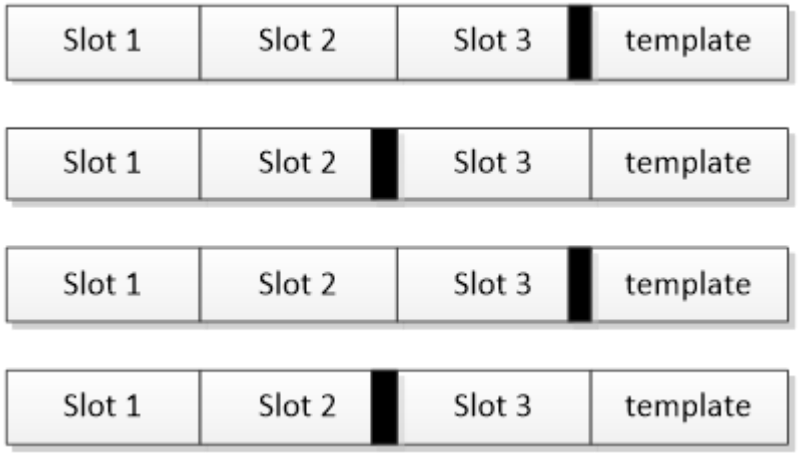
Бандлы – большие слова, в которые запакованы инструкции.

В **слотах** располагаются инструкции.
Шаблон определяет тип каждой из инструкций и позицию стопа.

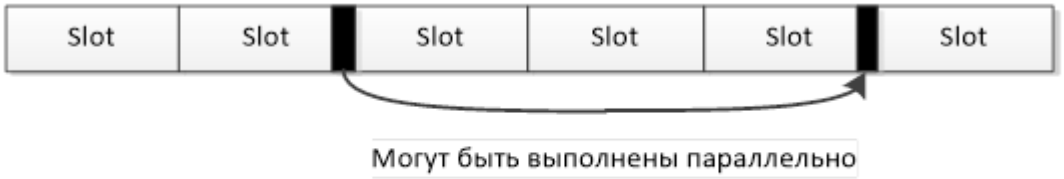
Типы инструкций:

M	Memory/Move operations
I	Complex Integer/Multimedia operations
A	Simple integer/Logic/Multimedia operations
F	Floating point operation (Normal/SIMD)
B	Branch Operations
L	Специальные

Itanium не умеет определять зависимости по данным, это определяется во время компиляции. Стоп указывает процессору, какие инструкции можно выполнить параллельно. Пример бандлов и позиций стопа:



Между стопами инструкции не имеют зависимости по данным, т.е. могут быть выполнены параллельно.



Примечания:

1. В бандле максимум одна F инструкция
2. Два шаблона имеют встроенный стоп

Типы шаблонов:

template	Slot 1	Slot 2	Slot 3
0	M	I	I
1	M	I	I
2	M	I	I
…	…	…	…

Причина введения **спекуляции** заключается в том, чтобы закончить загрузку данных из памяти до того, как они будут использованы. Поэтому компилятор старается поместить инструкцию загрузки данных как можно раньше. Пример:

До спекуляции:

```
0       st4 [r11] = r12, 4
1       ld4 r3 = [r4], 4
2       ...
3       add r7 = r5, r3
```

После спекуляции:

```
ld4 r3 = [r4], 4
st4 [r11] = r12, 4
chk r3, recover
back:
...
add r7 = r5, r3

// somewhere else in program
recover:
ld4 r3 = [r4], 4
goto back
```

Мы хотим переставить инструкцию загрузки 0 с инструкцией сохранения 1, но мы не знаем, будут ли r4 и r11 равны (ссылаться на одну и ту же ячейку памяти).

Проверка факта r[4] == r[11] осуществляется аппаратно с помощью специальной таблицы ALAT (Advanced Load Address Table).
Механизм спекуляции данных будет эффективен тогда, когда равенство регистров r11 и r4 будет маловероятно.

Программная конвейеризация циклов (Software pipelining, SWP).

```
L1:     ld4 r4 = [r5], 4 ;;    // такт 0; инструкция 1,загрузка 4
                               // байт из памяти [r5]
        add r7 = r4, r9 ;;     // такт 2; инструкция 2,
                               // сложение r7 = r4 + r9
        st4 [r6] = r7, 4       // такт 3;     инструкция 3,
                               // сохранение 4 байт в память [r6]
        br.cloop L1 ;;         // такт 3;     инструкция 4,
                               // переход на следующую итерацию
```

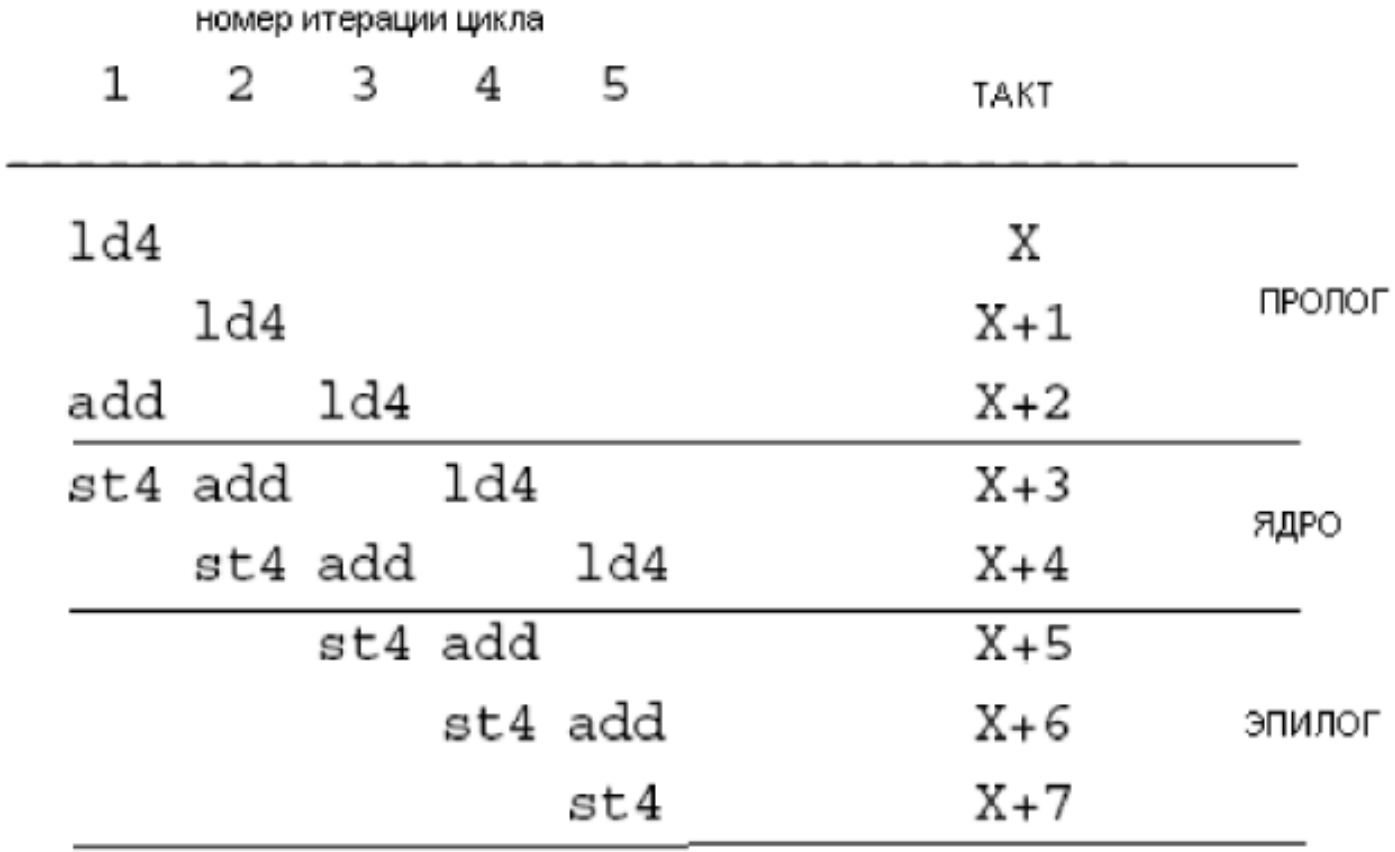
Инструкция 1 выполняется 2 такта, остальные – один. Существуют зависимости по данным, препятствующие распараллеливанию цикла. Itanium предоставляет специальную инструкцию ветвления br.ctop, которая используется для организации конвейеризованных циклов – во времени совмещается выполнение нескольких итераций цикла.

Разобьем тело цикла на ступени.

```
ступень 1:     ld4 r4 = [r5], 4
ступень 2:     --------------------
ступень 3:     add r7 = r4, r9
ступень 4:     st4 [r6] = r7, 4
```

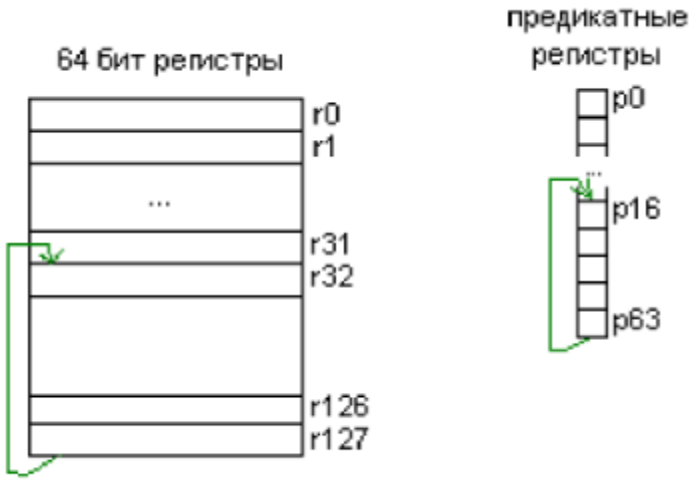
На такте X процессора выполняется первый такт загрузки данных для первой итерации. На такте X+1 – второй такт загрузки данных для первой итерации и первый такт загрузки данных для второй итерации и т.д.

Процесс выполнения конвейеризованного цикла делится на пролог, ядро и эпилог. Ядро – часть выполнения цикла, в котором ФУ процессора загружены по-максимуму.


номер итерации цикла
1
2
3
4
5
ТАКТ
ld4
X
ld4
X+1
ПРОЛОГ
add
ld4
X+2
st4
add
ld4
X+3
st4
add
ld4
X+4
ЯДРО
st4
add
X+5
st4
add
X+6
ЭПИЛОГ
st4
X+7

Рассмотрим, как работает команда br.ctop, для этого вводятся понятия вращения регистров и предикации в Itanium.

Два регистровых файла (совокупность регистров) – целочисленные регистры r0, …, r127 и предикатные p0, …, p63.

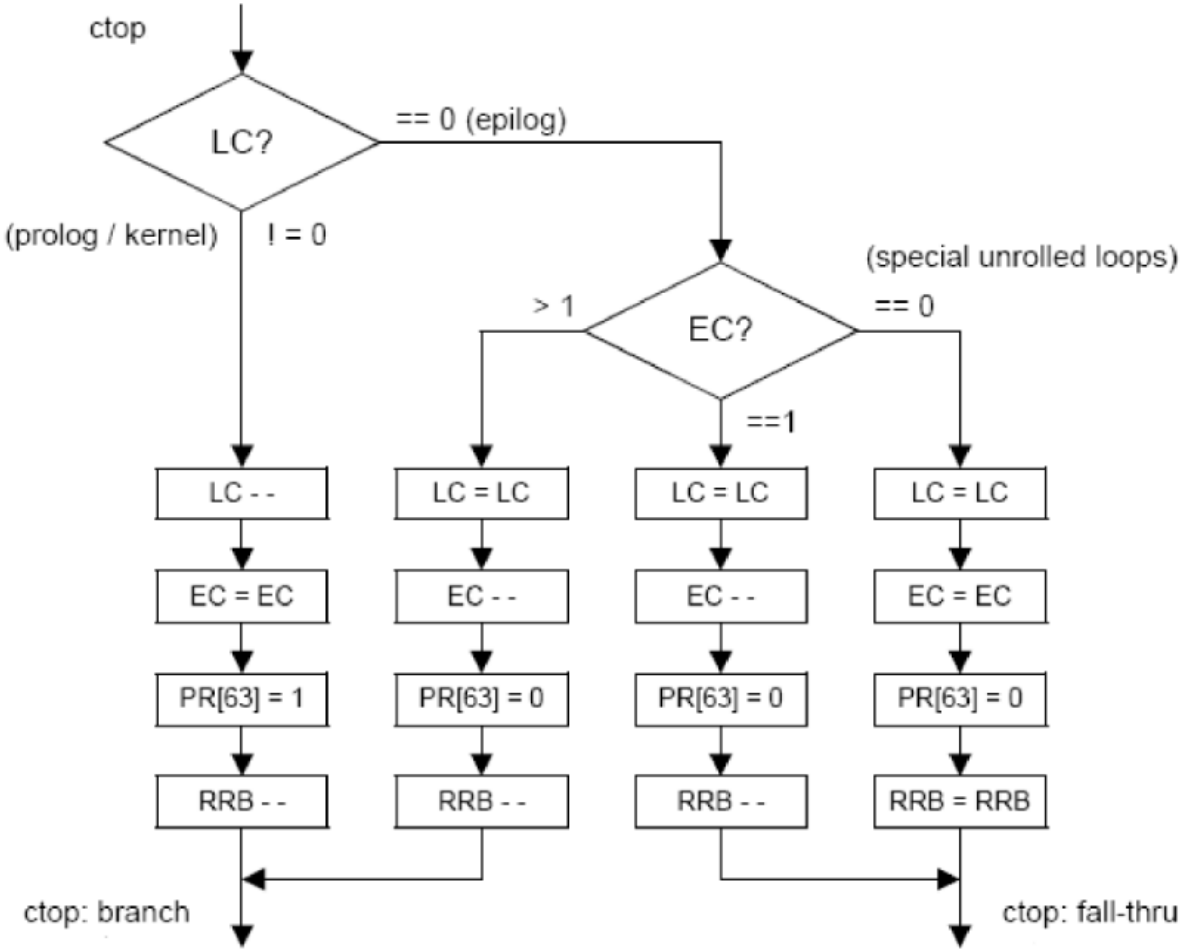


Предикатные регистры – однобитовые, нужны для условного выполнения команд. Например,

```
(p10) ld4 r4 = [r5], 4
```

выполнится, если p10 равен 1. Все команды в Itanium выполняются в предикатном режиме. Если предикатный регистр не указан, то используется константный регистр p0, равный 1.

В Itanium предусмотрены регистры LC и EC. LC – количество итераций цикла минус один. EC – количество стадий в эпилоге плюс один (4 в нашем случае). Теперь рассмотрим блок-схему команды br.ctop.



Конвейеризованный вариант цикла:

```
                mov LC = 199
                mov EC = 4
                PR[16] = 1  //все остальные предикатные регистры
                            //получат значение 0
L1:(p16)        ld4 r32 = [r5], 4
   (p18)        add r35 = r34, r9
   (p19)        st4 [r6] = r36, 4
                br.ctop L1 ;;
```

На такте 0 процессора выполняется первый такт загрузки данных в регистр r32 для первой итерации цикла и инструкция br.ctop. p16 == 1, т.е. выполняется ld4 r32 = [r5], 4. Инструкции для регистров p18 и p19, соответственно, не выполняются, поскольку регистры равны нулю. Далее происходит проворот регистров и декремент LC.

ТАКТ	команды				состояние предикатных регистров				состояние специальных регистров	
	M	**I**	**M**	**B**	**p16**	**p17**	**p18**	**p19**	**LC**	**EC**
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
…	…	…	…	…	…	…	…	…	…	…
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
…	…	…	…	…	…	…	…	…	…	…
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
…					0	0	0	0	0	0

Linda

В этой системе параллельная программа есть множество параллельных процессов, и каждый процесс работает согласно обычной последовательной программе. Все процессы имеют доступ к общей памяти, единицей хранения в которой является **кортеж**. Отсюда происходит и специальное название для общей памяти - пространство кортежей. Каждый кортеж это упорядоченная последовательность значений. Например:

```
( "Hello", 42, 3.14 )
( "P", 5, FALSE, 97, 1024, 2)
( "worker", 5 )
```

Первый элемент кортежа всегда является символьной строкой и выступает в роли имени кортежа.

Отличия пространства кортежей от традиционной памяти:

1. Процесс может забрать кортеж из пространства кортежей, после чего данный кортеж станет недоступным остальным процессам.

2. Если в пространство кортежей положить два кортежа с одним и тем же именем, то не произойдет привычного для нас "обновления" значения переменной - в пространстве кортежей окажется два кортежа с одним и тем же именем.

3. В отличие от традиционной памяти, изменить кортеж непосредственно в пространстве нельзя. Для изменения значений элементов кортежа, его нужно сначала оттуда изъять, затем процесс, изъявший кортеж, может изменить значения его элементов и вновь добавить измененный кортеж в память.

Процессы в системе Linda никогда не взаимодействуют друг с другом явно, и все общение всегда идет через пространство кортежей.

С точки зрения системы Linda в любой последовательный язык достаточно добавить лишь четыре новые функции – in, out, read, eval, как он становится средством параллельного программирования!

OUT

Функция **out** помещает кортеж в пространство кортежей. Например:

```
out ( "GoProcess", 5);
```

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

IN

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

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

Этой функции соответствует любой кортеж, который состоит из трех элементов: значением первого элемента является "P", второй элемент может быть любым целым числом, а третий должен иметь значение FALSE. Подходящими кортежами могут быть ("P", 5, FALSE) или ("P", 135, FALSE) и т.п., но не ("P", 7.2, FALSE) или ("Proc", 5, FALSE).

Если параметру функции **in** соответствуют несколько кортежей, то случайным образом выбирается один из них. После нахождения кортеж удаляется из пространства кортежей, а неопределенным формальным элементам параметра-кортежа, содержащимся в вызове данной функции, присваиваются соответствующие значения. В предыдущем примере переменной i присвоится 5 или 135. Если в пространстве кортежей ни один кортеж не соответствуют функции, то вызвавший ее процесс блокируется до тех пор, пока соответствующий кортеж в пространстве не появится.

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

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

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

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

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

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

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

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

READ

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

EVAL

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

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

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

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

Пример №1 – нумерация процессов и вычисление их количества.
Программа в самом начале вызывает функцию out:

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

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

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

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

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

Пример №2 – барьерная синхронизация.
Не имея в системе Linda никаких явных средств для синхронизации процессов, совсем не сложно их смоделировать самому. Предположим, что в некоторой точке нужно выполнить барьерную синхронизацию N процессов. Какой-то один процесс, например, стартовый, заранее помещает в пространство кортеж ("ForBarrier", N). Подходя к точке синхронизации, каждый процесс выполняет следующий фрагмент, который и будет выполнять функции барьера:

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

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

Пример №3 – перемножение матриц.
Рассмотрим возможную схему организации программы для перемножения C=A*B двух квадратных матриц размера N*N. Инициализирующий процесс использует функцию **out** и помещает в пространство кортежей исходные строки матрицы A и столбцы матрицы B:

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

Для порождения Nproc идентичных параллельных процессов можно воспользоваться следующим фрагментом:

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

Входные данные готовы, и нахождение всех N^2 элементов Cij результирующей матрицы можно выполнять в любом порядке. Главное - это распределить работу между процессами, для чего процесс, инициирующий вычисления, в пространство помещает следующий кортеж:

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

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

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

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

И, наконец, для вычисления элемента Cij каждый процесс get_elem_result выполнит следующий фрагмент:

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

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

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

И, наконец, для вычисления элемента Cij каждый процесс get_elem_result выполнит следующий фрагмент:

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

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

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

Вывод о Linda

Простота и стройность концепции Linda является основным козырем, однако эти же факторы оборачивается большой проблемой на практике.

1. Как эффективно поддерживать пространство кортежей на практике?

2. Если вычислительная система обладает распределенной памятью, то общение процессов через пространство кортежей заведомо будет сопровождаться большими накладными расходами.

3. Если процесс выполняет функции read или in - как среди потенциально огромного числа кортежей в пространстве быстро найти подходящий?

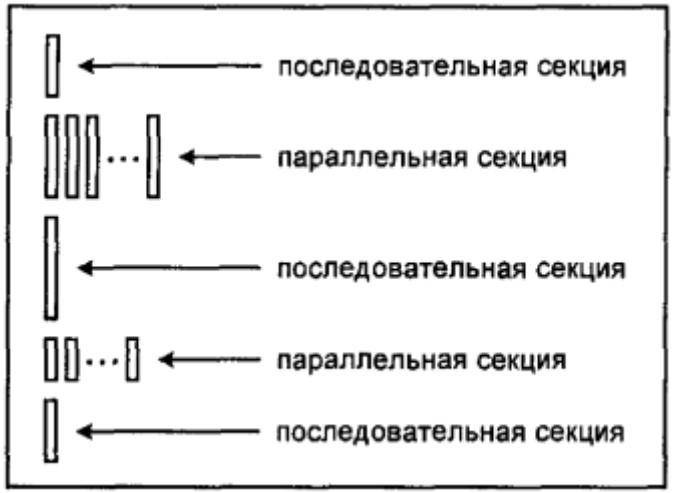
Подобные проблемы пытаются решить разными способами, например, введением нескольких именованных пространств, но все предлагаемые решения либо усложняют систему, либо являются эффективными только для узкого класса программ.

OpenMP

OpenMP – средство программирования для компьютеров с общей памятью. За основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляется набор директив, функций и переменных окружения. Разработан для языков Fortran, C/C++.

Особенности:

1) Исходный код разбит на последовательные и параллельные секции
2) В начальный момент времени порождается поток-мастер, которая выполняет последовательные секции.
3) При входе в параллельную секцию порождаются дополнительные нити. Каждая нить получает свой собственный номер. Поток-мастер всегда имеет номер 0. При выходе из параллельной области поток-мастер дожидается завершения остальных нитей (схема FORK/JOIN)



4) В параллельной области все переменные делятся на общие (shared) и локальные (private). Общие переменные существуют в одном экземпляре и доступны всем потокам под одним и тем же именем. Объявление локальной переменной вызывает порождение своего экземпляра каждой переменной для каждого потока.

`#pragma omp` – общий вид директив

`#pragma omp parallel { … }` – задание параллельной области

Число потоков, которые создадутся в параллельной области, по умолчанию берется из переменной окружения `OMP_NUM_THREADS`.

`omp_set_num_threads(int num)` – изменить число потоков (изменяется `OMP_NUM_THREADS`).

`omp_get_num_threads()` – получить число потоков.

`omp_get_thread_num()` – получить номер текущего потока.

```
int i, i1, i2;
#pragma omp parallel num_threads(4) private(i) shared(i1, i2)
{
        …
}
```

4 потока, в том числе поток-мастер, будут выполнять код внутри области, с локальными переменными i и общими i1, i2.

`#pragma omp parallel if(<условие>)`

Если условие не выполнено, то порождения потоков не происходит, и код в области исполняет только поток-мастер.

Выполнение потоками различного кода:

```
#pragma omp parallel
{
	if(omp_get_thread_num() == 3)
	{
		// код потока 3
		…
	}
	else
	{
		…
	}
}
```

Если в параллельной секции встретился оператор цикла, то каждый поток выполнит все итерации этого цикла. Для распределения итераций цикла между потоками используется `#pragma omp parallel for`:

```
#pragma omp parallel for
for(int i = 0; i < n; i++)
{
        …
}
```

Или, если уже находимся в параллельной области:

```
#pragma omp parallel
{
        …
        #pragma omp for
        for(int i = 0; i < n; i++)
        {
                …
        }
}
```

Опция schedule определяет способ распределения итераций цикла по потокам:

`static [, m]` – блочно-циклическое распределение, первый блок из m итераций выполняет первый поток, второй блок – второй поток и т.д. до последнего потока, затем распределение снова начинается с первого потока. Если m не указано, итерации делятся на непрерывные куски примерно одинакового размера и распределяются между потоками.

`dynamic [, m]` – динамическое распределение с фиксированным размером блока, сначала все потоки получают порции из m итераций, а затем каждый поток, заканчивая свою работу, получает следующую порцию из m итераций. Если m не указано, оно принимается равным единице.

`guided [, m]` – динамическое распределение блоками уменьшающегося размера. Сначала размер выделяемых блоков берется достаточно большим, а в процессе работы он все время уменьшается до минимального размера m. Размер первоначально выделяемого блока зависит от реализации. Если m не указано, оно принимается равным единице.

`runtime` – способ распределения итераций выбирается во время работы программы в зависимости от значения переменной OMP_SCHEDULE.

`#pragma omp parallel for schedule (dynamic, 3) { … }` – динамическое распределение блоками по 3 итерации.

Опция schedule определяет способ распределения итераций цикла по потокам:

`static [, m]` – блочно-циклическое распределение, первый блок из m итераций выполняет первый поток, второй блок – второй поток и т.д. до последнего потока, затем распределение снова начинается с первого потока. Если m не указано, итерации делятся на непрерывные куски примерно одинакового размера и распределяются между потоками.

`dynamic [, m]` – динамическое распределение с фиксированным размером блока, сначала все потоки получают порции из m итераций, а затем каждый поток, заканчивая свою работу, получает следующую порцию из m итераций. Если m не указано, оно принимается равным единице.

`guided [, m]` – динамическое распределение блоками уменьшающегося размера. Сначала размер выделяемых блоков берется достаточно большим, а в процессе работы он все время уменьшается до минимального размера m. Размер первоначально выделяемого блока зависит от реализации. Если m не указано, оно принимается равным единице.

`runtime` – способ распределения итераций выбирается во время работы программы в зависимости от значения переменной OMP_SCHEDULE.

`#pragma omp parallel for schedule (dynamic, 3) { … }` – динамическое распределение блоками по 3 итерации.

Примеры работы циклов (число обрабатываемых элементов - 48, число потоков - 4):

`static`

t0	00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
t1	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
t2	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
t3	48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

`static 2`

t0	00 01 08 09 16 17 24 25 32 33 40 41 48 49 56 57
t1	02 03 10 11 18 19 26 27 34 35 42 43 50 51 58 59
t2	04 05 12 13 20 21 28 29 36 37 44 45 52 53 60 61
t3	06 07 14 15 22 23 30 31 38 39 46 47 54 55 62 63

`static 32`

t0	00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
t1	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
t2	
t3	

dynamic

t0	10 12 14 18 20 22 24 26 28 30 32 34 36 38 40 42 43 45 47 49 52 54 56 58 59 61
t1	01 03 04 06 08 11 13 15 16 17 19 21 23 25 27 29 31 33 35 37 39 41 44 46 48 50 51 53 55 57 60 62 63
t2	00 02 05 07 09
t3	

dynamic 8

t0	00 01 02 03 04 05 06 07 16 17 18 19 20 21 22 23 40 41 42 43 44 45 46 47 56 57 58 59 60 61 62 63
t1	08 09 10 11 12 13 14 15 24 25 26 27 28 29 30 31 48 49 50 51 52 53 54 55
t2	32 33 34 35 36 37 38 39
t3	

dynamic 4

t0	00 01 02 03 12 13 14 15 20 21 22 23 24 25 26 27 32 33 34 35 40 41 42 43 52 53 54 55 56 57 58 59
t1	
t2	04 05 06 07 44 45 46 47
t3	08 09 10 11 16 17 18 19 28 29 30 31 36 37 38 39 48 49 50 51 60 61 62 63

guided

t0	00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 44 45 46 47 48 49 50 51 52 56 57 58 59 60 61 62 63
t1	28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 53 54 55
t2	16 17 18 19 20 21 22 23 24 25 26 27
t3	

guided 8

t0	00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 37 38 39 40 41 42 43 44 53 54 55 56 57 58 59 60 61 62 63
t1	28 29 30 31 32 33 34 35 36
t2	16 17 18 19 20 21 22 23 24 25 26 27 45 46 47 48 49 50 51 52
t3	

guided 4

t0	00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 37 38 39 40 41 42 43 44 45 46 47 48 57 58 59 60
t1	
t2	28 29 30 31 32 33 34 35 36 49 50 51 52 53 54 55 56 61 62 63
t3	16 17 18 19 20 21 22 23 24 25 26 27

Параллелизм на уровне независимых секции.

```
#pragma omp sections
{
        #pragma omp section
        {
                …
        }
        #pragma omp section
        {
                …
        }
        …
}
```

Все секции внутри `#pragma omp sections` можно исполнять параллельно, каждая из секций будет выполнена только один раз каким-либо одним потоком. Если число потоков больше числа секций, то какие потоки задействовать, а какие не использовать, решается конкретной реализацией OpenMP.

`#pragma omp barrier`
Потоки, дойдя до этой директивы, останавливаются и ждут, пока оставшиеся потоки не дойдут до нее.

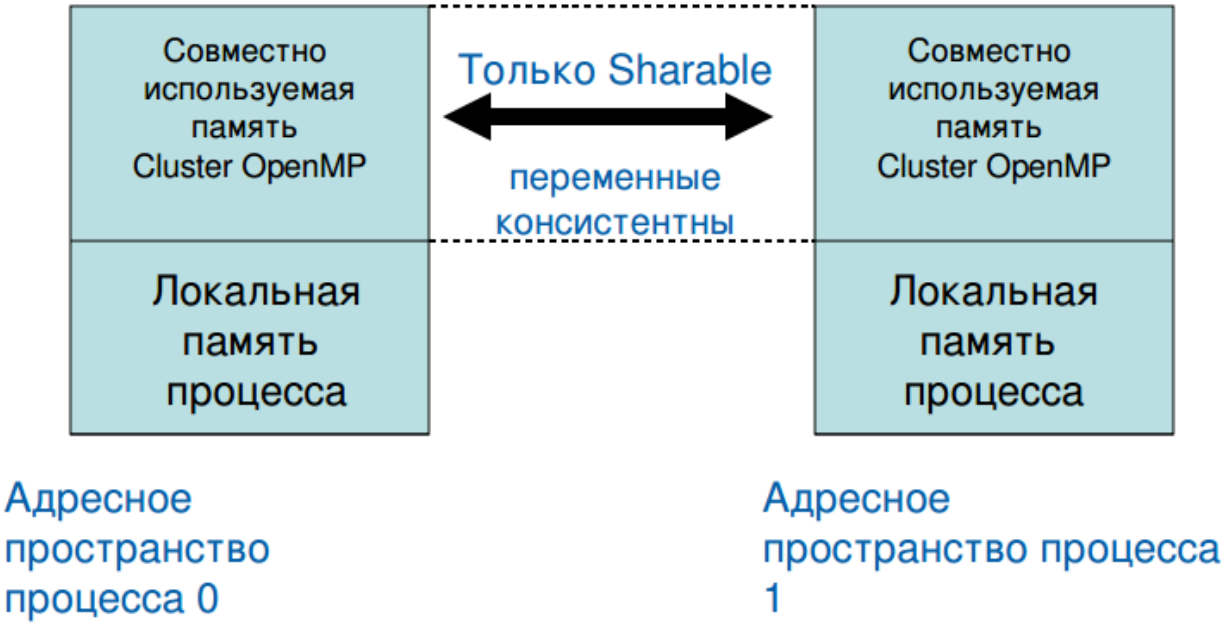
`#pragma omp master { … }`
Директива выделяет участок кода, который будет выполнен только мастер-потоком, остальные потоки пропускают этот участок.

`#pragma omp critical { … }`
Директива определяет критическую секцию, которую в некоторый момент времени выполняет только один поток. Другие потоки при достижении директивы засыпают, если секция занята. Затем при освобождении секции очередной поток для входа в секцию выбирается случайно.

`#pragma omp atomic`
Директива обеспечивает атомарность выполнения идущего вслед за ней оператора

`#pragma omp flush <список переменных>`
Значения всех переменных, временно хранящихся в регистрах, будут занесены в основную память, все изменения переменных, сделанные потоками во время их работы, станут видны другим потокам. Это делается для гарантии того, что в нужный момент времени каждый поток будет видеть единый согласованный образ памяти.

Cluster OpenMP - простое средство, требующее незначительного изменения кода для расширения параллелелизма OpenMP на кластерные системы. Высокая стоимость больших систем с общей памятью была главным ограничением для использования OpenMP, поэтому для совместного использования нескольких SMP-систем зачастую прибегали к помощи MPI. Cluster OpenMP избавляет от смешиванию этих двух парадигм. В Cluster OpenMP поддерживается система распределенной общей памяти. Cluster OpenMP позволяет синхронизировать общие переменные только в случае необходимости. При разработке программы следует строить ее так, чтобы максимизировать число обращений к локальной памяти узла и уменьшать общее число синхронизаций.



Стоимость вычислений на кластерных системах:

Тип кластера	Аппаратная составляющая расходов	Программная составляющая расходов
Большие SMP системы с общей памятью и OpenMP		
Кластерные системы с распределенной памятью и MPI		
Кластерные системы с распределенной памятью и Cluster OpenMP		

Преимущества OpenMP:

1. Идея "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.

2. Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки (stubs), текст которых приведен в спецификациях.

Настройка OpenMP в проекте

1. Создаем проект
2. Project -> properties (alt + f7) -> configuration properties -> c/c++ -> Language -> Enable OpenMP support: yes
3. Обязательно `#include <omp.h>`
4. Проверяем:

```
#include <omp.h>
#include <iostream>
int main()
{
    int test = 1;
    omp_set_num_threads( 4 );
    #pragma omp parallel
    {
        #pragma omp critical
        std::cout << "thread " << omp_get_thread_num();
    }
}
```

Лабораторная работа

1. Реализовать метод Гаусса для систем произвольной размерности на OpenMP. Показать ускорение.

2. Распараллелить алгоритм чет-нечетной перестановки. Показать ускорение. Сделать двумя способами – просто распределять итерации между потоками (при сравнении пар), и с помощью поблочного распределения массива между потоками и чередованием сортировки внутри блока и сравнением-разделением между потоками.

3. Придумать пример (можно искусственный), в котором динамическое планирование итераций имело бы преимущество в производительности над статической.

4. Придумать пример (можно искусственный), в котором множество потоков имеют доступ к разделяемому ресурсу через критическую секцию.

Распараллеленый метод Гаусса

Прямой ход.
Итерации i∈[0,n-1) Главный поток выбирает строку с максимальным i-ым элементом. Затем потоки, получив i-ую строку от главного потока, вычитают их из «своих» строк для исключения из них i-го элемента.

Обратный ход.
Подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, i∈[0,n-1), определяет значение своей переменной x_i, это значение должно быть разослано всем подзадачам с номерами k, k<i. Далее подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора b.

Алгоритм чет-нечетной перестановки

Алгоритм пузырьковой сортировки в прямом виде достаточно сложен для распараллеливания – сравнение пар значений упорядочиваемого набора данных происходит строго последовательно. В связи с этим для параллельного применения используется не сам этот алгоритм, а его модификация, известная в литературе как метод чет-нечетной перестановки.
Суть модификации состоит в том, что в алгоритм сортировки вводятся два разных правила выполнения итераций метода: в зависимости от четности или нечетности номера итерации сортировки для обработки выбираются элементы с четными или нечетными индексами соответственно, сравнение выделяемых значений всегда осуществляется с их правыми соседними элементами.
Таким образом, на всех нечетных итерациях сравниваются пары

`(a1, a2), (a3, a4), ..., (an-1, an)` (при четном n),

а на четных итерациях обрабатываются элементы

`(a2, a3), (a4, a5), ..., (an-2, an-1)`.

После n-кратного повторения итераций сортировки исходный набор данных оказывается упорядоченным.

Первый способ распараллеливания – распараллелить сравнение пар значений массива ((a1, a2) – первый поток, (a3, a4) – второй поток и т.д.).

При втором способе распараллеливания сначала массив поблочно распределяется между потоками. Затем на каждой итерации сначала сортируется каждый блок массива (с помощью любого алгоритма сортировки), затем происходят слияния соседних блоков, в результате которых в блоках с меньшими номерами оказываются меньшие части из двух блоков, с большими номерами – большая часть. Сортировка в блоках сохраняется.

№ и тип итерации	Потоки			
	1	2	3	4
Исходные данные	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 43
1 нечет (1, 2), (3, 4)	13 55 59 88	29 43 71 85	2 18 40 75	4 14 22 23
	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
2 чет (2, 3)	13 29 43 55	59 71 85 88	2 4 14 18	22 40 43 75
	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
3 нечет (1, 2), (3, 4)	13 29 43 55	2 4 14 18	59 71 85 88	22 40 43 75
	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
4 чет (2, 3)	2 4 13 14	18 29 43 55	22 40 43 59	71 75 85 88
	2 4 13 14	18 22 29 40	43 43 55 59	71 75 85 88