1. **Единицы и методы измерения производительности**

FLOPS - единица, используемая для измерения производительности компьютеров, показывающая, сколько операций с плавающей запятой в секунду выполняет данная вычислительная система.

Значение FLOPS, опубликованное для конкретной системы, – это характеристика прежде всего самого компьютера, а не программы. Данная величина определяется путём запуска на испытуемом компьютере тестовой программы, которая решает задачу с известным количеством операций и подсчитывает время, за которое она была решена. Наиболее популярным тестом производительности на сегодняшний день являются тесты производительности LINPACK, используемый при составлении рейтинга суперкомпьютеров TOP500.

1. **Производительность наиболее мощных вычислительных систем и динамика изменения показателей и лидирующих архитектур.**



**Применение суперкомпьютеров**

* **Sunway TaihuLight**

Суперкомпьютер Sunway TaihuLight предназначен для сложных расчётов, требуемых в производстве, медицине, добывающей промышленности, для прогнозирования погодных условий и анализа «больших данных». Он расположен в национальном суперкомпьютерном центре в Уси, провинция Цзянсу.

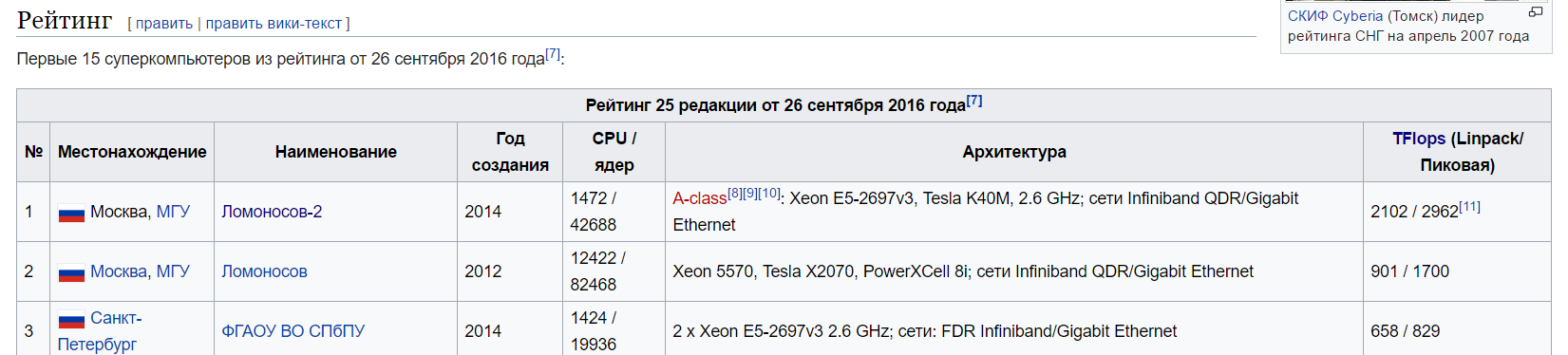
* **Titan**

*Процессы сгорания топлива*: моделирование турбулентного сгорания различных видов и составов топлива. Результаты исследований позволят создавать высокоэффективные, экономичные, оптимальные двигательные системы, почти не загрязняющие окружающую среду.

*Атомная энергия*: моделирование поведения нейтронов в ядерном реакторе. Результаты исследований позволят выяснить, как более эффективно использовать стареющие ядерные реакторы США при сохранении их безопасности.

*Изменения климата*: симулирование долгосрочных глобальных изменений климата. Результаты вычислений позволят исследователям вычислить качество воздуха в ближайшем будущем, а также влияние его состава на климат.

1. **Производительные платформы в России.**



**Применение**

Предполагается использовать суперкомпьютер для решения ресурсоёмких вычислительных задач в рамках фундаментальных научных исследований, а также для проведения научной работы в области разработки алгоритмов и программного обеспечения для мощных вычислительных систем.

1. **Какими методами повышается производительность приложений.**

* Оптимизация должна быть переносимой
* Оптимизация должна допускать безболезненное внесение изменений
* Прежде чем оптимизировать код, обязательно следует иметь надёжно работающий неоптимизированный вариант.
* Основной прирост оптимизации даёт не учёт особенностей системы, а алгоритмическая оптимизация.
* Для повышения производительности приложений можно использовать оптимизирующие компиляторы. Он автоматически задействует функциональные возможности процессора и стратегии оптимизации.
* Можно также оптимизировать какую-либо функцию путем вставки ассемблерного кода с определенными командами, имеющимися только на конкретных процессорах. (но этот метод не кроссплатформенный).
* Эффективный метод повышения производительности, это использование параллелизма.
* Также для повышения производительности нужно писать эффективный код: продумывать эффективные алгоритмы, стараться не использовать медленные операции, писать код так, чтобы его можно было распараллелить.

1. **Методы распараллеливания обработки данных в компьютерных архитектурах.**

Параллелизм является важной технологией повышения производительности системы за счет выполнения более одного действия одновременно.

Параллелизм на уровне команд позволяет выполнять более одной команды одновременно. Но такой метод становится возможным только тогда, когда определенные команды выполняются в определенном порядке.

Для многопроцессорных систем параллелизм достигается путем использования нескольких программных потоков в одном приложении.

Есть пять вариантов параллелизма:

1. Параллелизм на уровне команд: процессор выполняет несколько команд одновременно.
2. Гиперпоточность: процессор выполняет работу двух программных потоков.
3. Два ядра, одна микросхема: выполняет два программных потока с разделением ресурсов микросхемы.
4. Многопроцессорная обработка: два процессора, один компьютер.
5. Распределенная обработка: несколько компьютеров.

Можно также написать про OpenMP. Или впарить еще и про алгоритмы, гранулярность.

Нужно при распараллеливании распределять равномерно на все потоки.

1. **Алгоритмические методы повышения производительности:**

**а) задачи обработки числовой информации;**

**б) задачи реализации логических функций.**

Правильный выбор алгоритма играет важную роль в достижении высокой производительности.

Например, нужно добиваться, чтобы зависимость данных друг от друга была минимальной, чтобы процессор мог выполнять одновременно несколько команд.

Разрабатывать такие алгоритмы, которые могут быть легко адаптированы для поточной обработки, хорошо масштабируются на несколько процессоров, эффективно используют кэши процессоров, не ограничены некоторыми медленными командами.

Если, например, в алгоритме требуется деление, то можно избежать его путем вычитания, сдвигов, или применения справочных таблиц.

Также частым применяемым подходом с целью избежать выполнения медленных функций является использование справочной таблицы для хранения предварительно посчитанных результатов.

Для избежания медленных функций можно также писать их самим. Так как большинство функций, предназначены для общих случаев.

Также strlen, также многие общие функции слишком точные, также нужно использовать не if else, а switch, развертывать циклы. Также индексация.

**а)** Про числовую обработку, тут как раз про точность, что не нужно со слишком большой точностью все обрабатывать со стандартными функциями, и также про операции с ними, что можно и сдвигами обойтись, а не делением.

**б)** При реализации логических функций, нужно ставить на первые места те варианты, которые с большей вероятностью будут истиной.

1. **Вариации программно-технических решений с целью уменьшения времени исполнения программных функций.**

Тут рассказать про оптимизацию в коде.

* Если в switch используются последовательные значения параметров case (case 0: case 1: case 2:), то switch значительно эффективнее чем if-else. Это происходит за счет того, что при if-else будет вычисляться значение каждого условия, а в случае таких параметров в конструкции switch значение будет вычислено один раз, а затем будет переход сразу к нужному пункту.
* В некоторых случаях, для увеличения быстродействия кода, можно разворачивать циклы. Это особенно эффективно, если они короткие. Когда цикл развернут, отпадает потребность в индексной переменной и ее проверках при каждой итерации.
* Объявление переменных там, где они будут использоваться, давая им наименьшую область видимости.

1. **Влияние структур данных и технологии программирования на производительность.**

Способы хранения данных

Фиксированная точка - формат хранения данных, выравнивания порядков нет, поэтому быстрее работает.

В Джаве создается копия строки, при конкатенации, а в СИ напрямую в память.

структуры — списки, деревья O(log n), стеки, хэши - разные скорости

Динамические массивы.

1. **Профилирование программ для оценки производительности.**

Профилировщики функционируют путем периодического прерывания работы системы для записи информации о производительности (время исполнения, указатель команд процессора, идентификатор потока, идентификатор процесса и счетчика событий). Собрал необходимую информацию, можно получить точное представление о том, что именно делала программа в сеансе сбора данных.

Пример - vTune;

1. **Анализ продуктов трансляции программ и машинно-ориентированные методы повышения производительности.**

Анализ продуктов трансляции программ и машинно-ориентированные методы повышения производительности. критерия оптимизации, это:    
  
1) размер кода программы,    
2) время исполнения программы.    
  
Оптимизационная техника (прием) – это такое преобразование кода программы, которое приводит к улучшению некоторых его свойств (уменьшение размера, уменьшение времени исполнения).

Примеры оптимизационных техник:    
• планирование размещения переменных на регистрах процессора,    
• изменение порядка инструкций в коде программы,    
• замена долгих инструкций на более быстрые,    
• устранение избыточного кода,    
• оптимизация циклов и условных переходов,    
• использование команд предвыборки данных,  использование специальных расширений архитектуры.    
  
Результатом оптимизации является более эффективный код (т.е. более быстрый или меньший по объему), дающий при выполнении такой же результат. Наиболее важным критерием обычно считается время исполнения программы.

1. **Использование многопоточности для распараллеливания вычислений на многоядерных архитектурах с целью повышения производительности.**

Тоже самое что и выше.

1. **Уровни параллелизма и гранулярность параллельных вычислений.**

**Уровни параллелизма:**

1. Уровень заданий – несколько независимых заданий одновременно выполняются на разных процессорах.
2. Уровень программ – части одной задачи выполняются на множестве процессоров.
3. Уровень команд – разные фазы нескольких команд выполняются одновременно на различных стадиях конвейера.
4. Уровень данных машинной команды – обработка нескольких операндов в одной команде.

Гранулярность – отношение объема вычислений к объему коммуникаций между параллельными ветвями.

* Крупнозернистый параллелизм – слабая зависимость между ветвями параллельных вычислений (тысячи исполненных команд на одну операцию обмена).
* Среднезернистый параллелизм – средняя зависимость (сотни команд на одну операцию обмена).
* Мелкозернистый параллелизм – единицы и десятки команд обработки на одну операцию обмена между параллельными ветвями.

1. **Метрики параллельных вычислений**

Метрики параллельных вычислений - это система показателей, позволяющая оценивать преимущества, получаемые при параллельном решении задачи на *n* процессорах, по сравнению с последовательным решением той же задачи на единственном процессоре.   
Базисом для определения метрик являются следующие характеристики вычислений:

* ***n*** - количество процессоров, используемых для организации параллельных вычислений;
* ***O(n)*** - объем вычислений, выраженный через количество операций, выполняемых *n* процессорами в ходе решения задачи;
* ***Т(n)*** - общее время вычислений (решения задачи) с использованием *n* процессоров.

В однопроцессорной системе ***T(1)=O(1)***. В общем случае ***Т(n)<О(n)***, если в единицу времени *n* процессорами выполняется более чем одна команда, где ***n>2***. Последнее соотношение формулирует утверждение: время вычислений можно сократить за счет распределения объема вычислений по нескольким процессорам.  
Можно выделить четыре группы метрик.  
Первая характеризует скорость вычислений:

* Ускорение: ***S(n)=T(1)/T(n)***.

Вторую группу образуют метрики, дающие возможность судить об эффективности привлечения к решению задачи дополнительных процессоров. Эффективность n-процессорной системы - это ускорение, приходящееся на один процессор.

* Эффективность: ***E(n)=S(n)/n=T(1)/(nT(n))***.

Третья группа метрик характеризует эффективность параллельных вычислений путем сравнения объема вычислений, выполненного при параллельном и последовательном решении задачи.

* Избыточность: ***R(n)=O(n)/O(1)***.

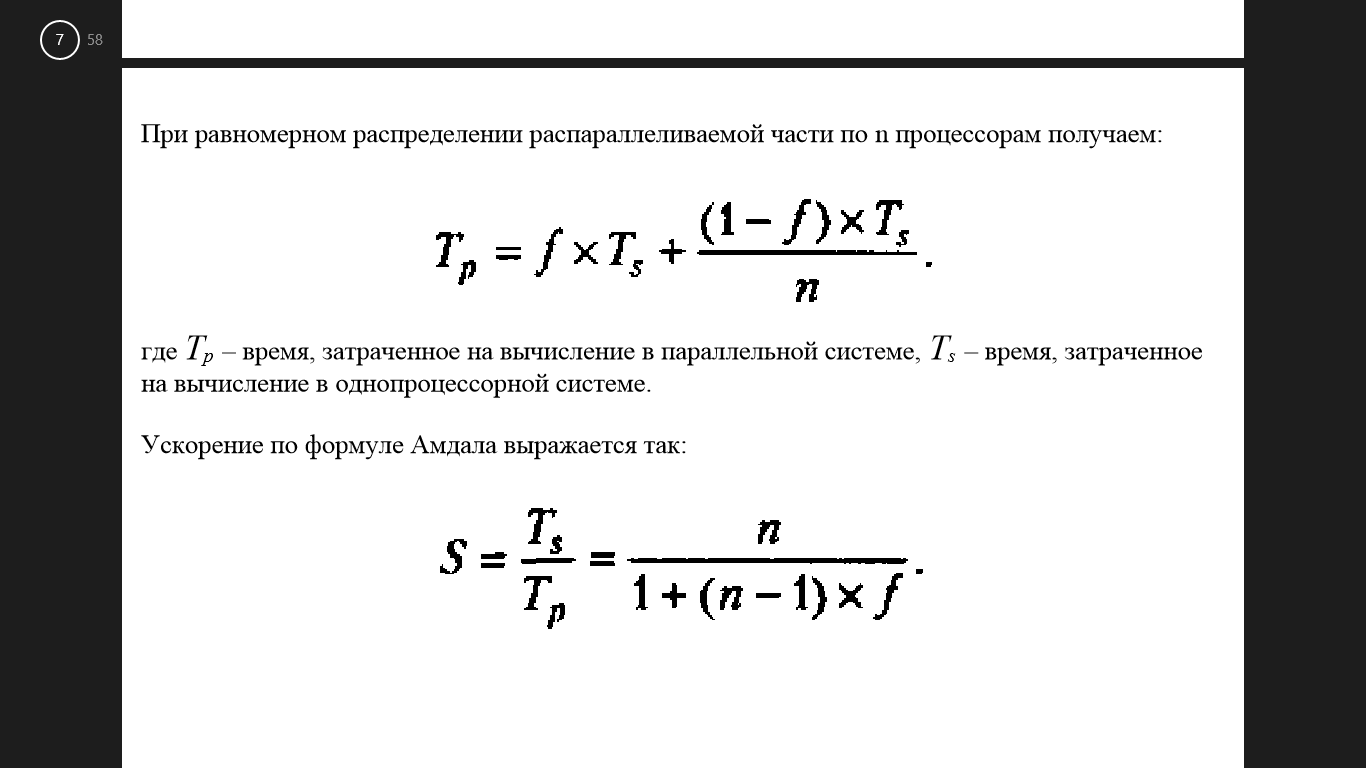
Четвертую группу образует метрика *качество*:

***Q(n)=S(n)E(n)R(n).***

1. **Законы Амдала и Густафсона**

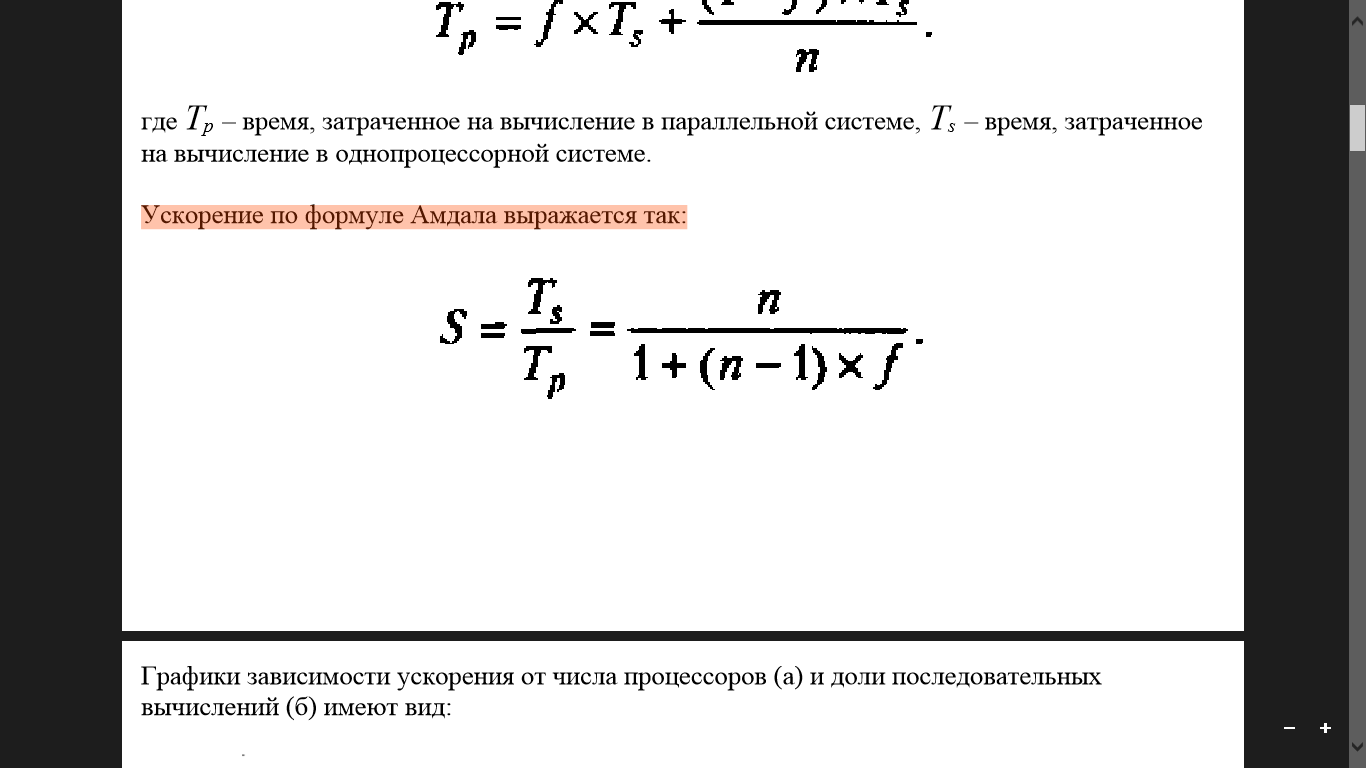
Джин Амдал предложил формулу для оценки ускорения в зависимости от соотношения двух долей программы – распараллеливаемой и чисто последовательной (не распараллеливаемой). Пусть доля последовательной части программы обозначена через f, где 0 ≤ f ≤ 1.

При равномерном распределении распараллеливаемой части по n процессорам получаем:



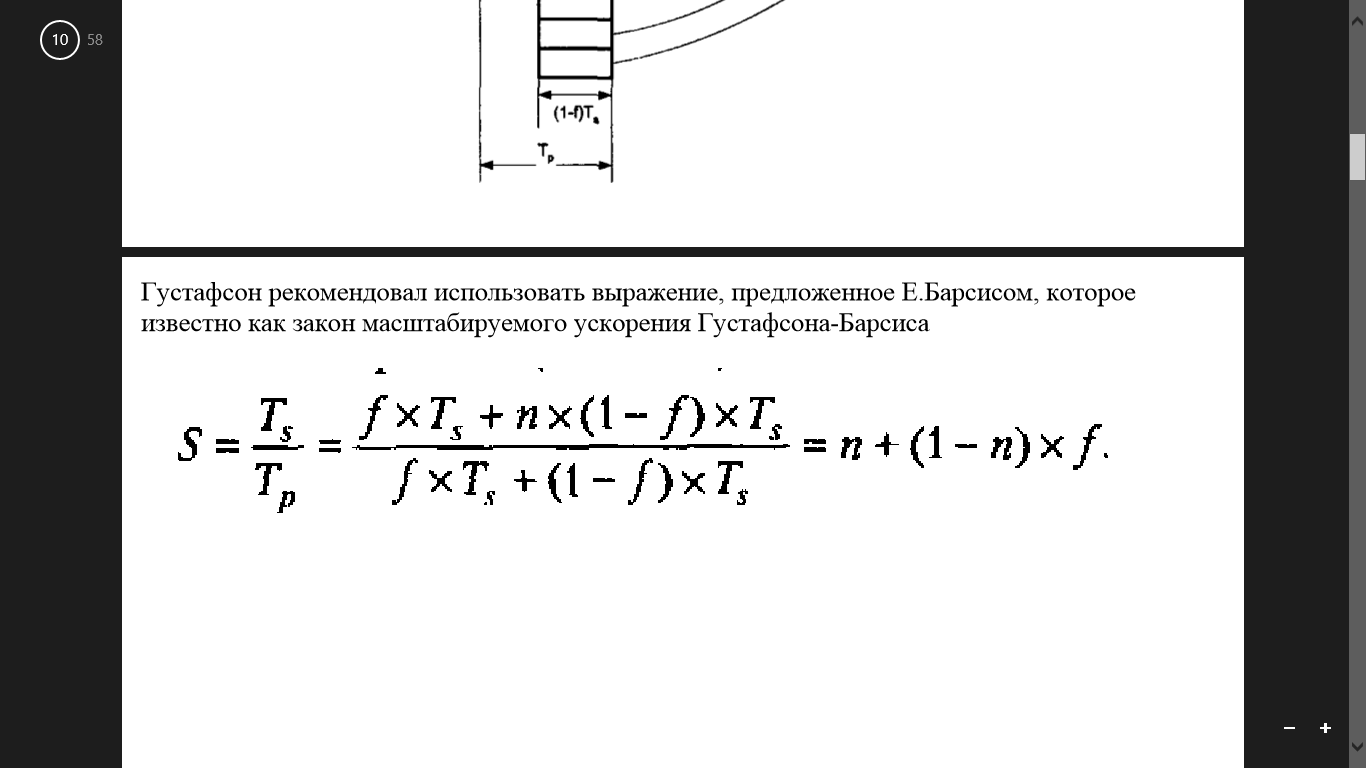
где Tp – время, затраченное на вычисление в параллельной системе, Ts – время, затраченное на вычисление в однопроцессорной системе.

Ускорение по формуле Амдала выражается так:



**Закон Густафсона**

Джон Густафсон обнаружил, что при увеличении числа процессоров пользователи увеличивают объем решаемой задачи. Причем, это увеличение касается главным образом распараллеливаемой части.



Таким образом, закон Густафсона характеризует ситуацию, при которой время вычислений с расширением системы не меняется, но увеличивается объем решаемой задачи. Цель такого подхода - за заданное время выполнить максимальный объем вычислений.

1. **Классификация параллельных систем**

Флинн в 1966 году предложил одну из самых распространенных классификаций.

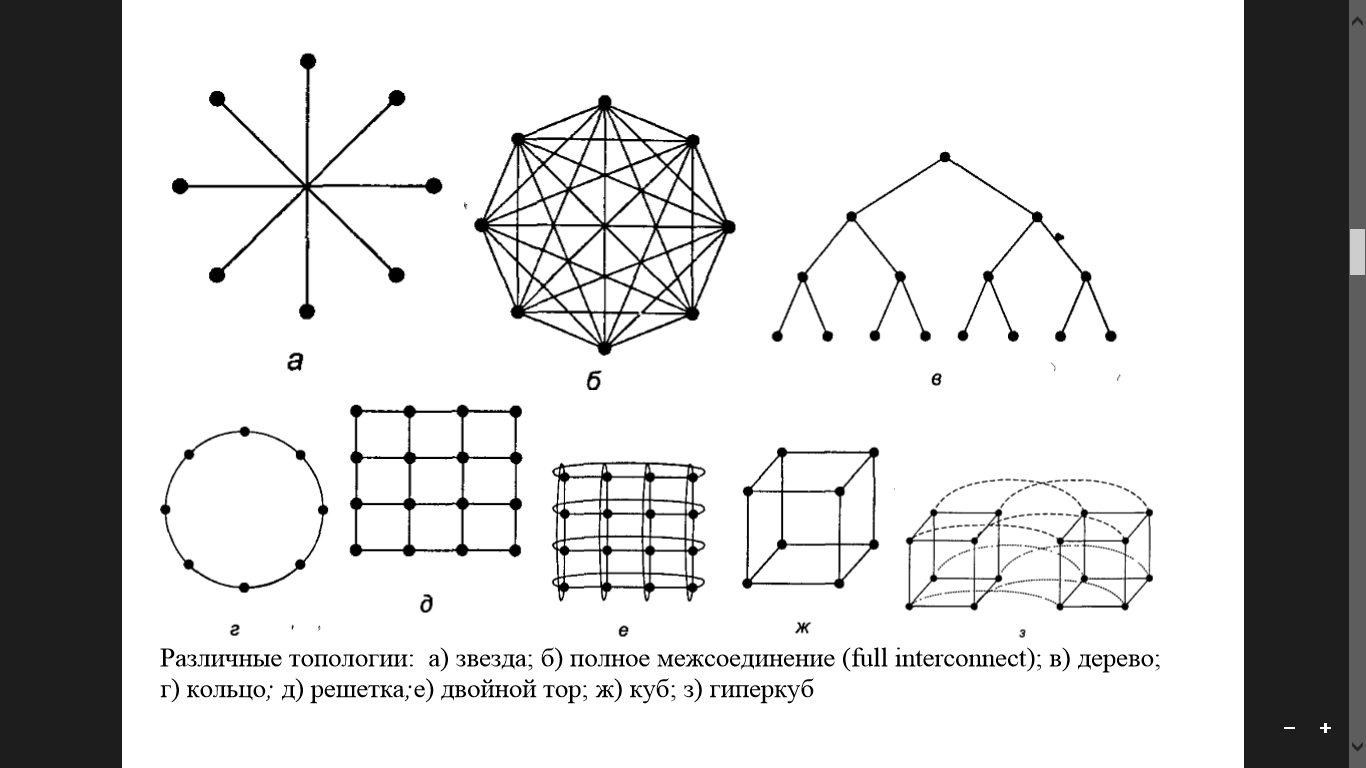
|  |
| --- |
| **SISD** (single instruction stream / single data stream) - одиночный поток команд и одиночный поток данных. К этому классу относятся, прежде всего, классические последовательные машины. В таких машинах есть только один поток команд, все команды обрабатываются последовательно друг за другом, и каждая команда инициирует одну операцию с одним потоком данных. |
| **SIMD** (single instruction stream / multiple data stream) - одиночный поток команд и множественный поток данных. В архитектурах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Это позволяет выполнять одну арифметическую операцию сразу над многими данными - элементами вектора. |
| **MISD** (multiple instruction stream / single data stream) - множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. |
| **MIMD** (multiple instruction stream / multiple data stream) - множественный поток команд и множественный поток данных. Этот класс предполагает, что в вычислительной системе есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каждое со своим потоком команд и данных. |

1. **Сети взаимодействия и их топологии**

Сети взаимодействия могут состоять максимум из пяти компонентов:

1) центральные процессоры;   
2) модули памяти;   
3) интерфейсы;   
4) каналы связи;   
5) коммутаторы.

Интерфейсы объединяют процессор, память и каналы связи между собой. Каналы связи обеспечивают передачу данных между центральными процессорами. Коммутаторы — это устройства с несколькими входными и несколькими выходными портами. Когда на входной порт приходит пакет, некоторые биты в этом пакете используются для выбора выходного порта, в который посылается пакет. Размер пакета может составлять 2 или 4 байта, но может быть и значительно больше (например, 8 Кбайт). Одним из важнейших свойств сети межсоединений является топология. Топология определяет, как расположены каналы связи и коммутаторы (это, например, может быть кольцо или решетка). Топологии можно изображать в виде графов, в которых дуги соответствуют каналам связи, а узлы — коммутаторам (рис. 3). С каждым узлом в сети (или в соответствующем графе) связан определенный ряд каналов связи. Математики называют число каналов степенью узла, инженеры — коэффициентом разветвления. Чем больше степень, тем больше вариантов маршрута и тем выше отказоустойчивость. Если каждый узел содержит k дуг и соединение сделано правильно, то можно построить сеть межсоединений так, чтобы она оставалась полносвязной, даже если k-1 из этих каналов повреждены. Следующее свойство сети межсоеди-нений — ее диаметр. Если расстоянием между двумя узлами мы будем считать число дуг, которые нужно пройти, чтобы попасть из одного узла в другой, то диаметром графа будет расстояние между двумя наиболее удаленными друг от друга узлами. Диаметр сети определяет самую большую задержку при передаче пакетов от одного процессора к другому. Среднее расстояние между двумя узлами определяет среднее время передачи пакета. Еще одно важное свойство сети межсоединений — это ее пропускная способность, то есть количество данных, которое она способна передавать в секунду. Очень важная характеристика — бисекционная пропускная способность. Чтобы вычислить это число, нужно мысленно разделить сеть межсоединений на две примерно равные (с точки зрения числа узлов) части путем удаления ряда дуг из графа так, что эти части перестанут быть связанными между собой. Затем нужно вычислить общую пропускную способность дуг, которые мы удалили. Предположим, что бисекционная пропускная способность составляет 800 кбит/с. Тогда если между двумя частями много взаимодействий, то общую пропускную способность в худшем случае можно сократить до 800 кбит/с. Сети межсоединений можно характеризовать по их размерности. Размерность определяется по числу возможных вариантов перехода из исходного пункта в пункт назначения. Если выбора нет (то есть существует только один путь из каждого исходного пункта в каждый конечный пункт), то сеть нульмерная. Если есть два возможных варианта (например, если можно пойти либо направо, либо налево), то сеть одномерна. Если есть две оси и пакет может направиться направо или налево либо вверх или вниз, то такая сеть двумерна и т. д.



На рис. показано несколько топологий. Жирные точки соответствуют коммутаторам. Процессоры и модули памяти не показаны. На рис. 4, а изображена нульмерная конфигурация звезда, где процессоры и модули памяти прикрепляются к внешним узлам, а коммутацию совершает центральный узел. Такая схема очень проста, но централизация коммутатора снижает производительность и надежность. На рис. 4, б изображена другая нульмерная топология — полное межсоединение(полносвязная). Здесь каждый узел непосредственно связан со всеми другими узлами. Пропускная способность максимальна, диаметр минимален, а отказоустойчивость очень высока. Однако для k узлов требуется k(k-l)/2 каналов, а это сильно удорожает сеть. На рис. 4, в изображена третья нульмерная топология — дерево. Здесь обычно у верхушки дерева наблюдается очень большой поток обмена информации, что заставляет иногда делать неоднородную сеть. Например, самые нижние каналы будут иметь пропускную способность Ь, следующий уровень — пропускную способность 2Ь, а каждый канал верхнего уровня — пропускную способность 4Ь. Такая схема называется толстым деревом (fat tree). Кольцо (рис. 4, г) — это одномерная топология, поскольку каждый отправленный пакет может пойти направо или налево. Решетка или сетка (рис. 4, д) — это двумерная топология, которая применяется во многих коммерческих системах. Она отличается регулярностью и применима к системам большого размера, а диаметр составляет квадратный корень от числа узлов (то есть при расширении системы диаметр увеличивается незначительно). Двойной тор (рис. 4, е) является разновидностью решетки. Это решетка, у которой соединены края. Она характеризуется большей отказоустойчивостью и меньшим диаметром, чем обычная решетка, поскольку теперь между двумя противоположными узлами всего два транзитных участка. Куб (рис. 4, ж) — это правильная трехмерная топология. На рисунке изображен куб 2x2x2, но в общем случае он может быть kxkxk. На рис. 4, з показан четырехмерный куб, полученный из двух трехмерных кубов, которые связаны между собой. Можно сделать пятимерный куб, соединив вместе 4 четырехмерных куба. Чтобы получить 6 измерений, нужно продублировать блок из 4 кубов и соединить соответствующие узлы и т. д.; n-мерный куб называется гиперкубом. Эта топология используется во многих компьютерах параллельного действия, поскольку ее диаметр находится в линейной зависимости от размерности. Чем меньше диаметр гиперкуба, тем больше разветвления и число каналов (и, следовательно, тем выше стоимость). Тем не менее в системе с высокой производительностью чаще всего используется именно гиперкуб.

1. **Векторные и векторно-конвейерные системы**

Векторные вычислительные системы - системы класса SIMD, в которых одна и та же заданная операция выполняется сразу над всеми компонентами векторов.

Векторные процессоры выполняют какую-то операцию над целыми массивами данных, векторами. По классификации Флинна, векторные процессоры относятся к **SIMD — (single instruction, multiple data — одиночный поток команд, множественный поток данных)**.

Вот как, например, выглядит умножение четырех пар чисел одной командой с применением SSE:  
  
float a[4] = { 300.0, 4.0, 4.0, 12.0 };  
float b[4] = { 1.5, 2.5, 3.5, 4.5 };  
\_\_asm {  
movups xmm0, a ; // поместить 4 переменные с плавающей точкой из a в регистр xmm0  
movups xmm1, b ; // поместить 4 переменные с плавающей точкой из b в регистр xmm1  
mulps xmm1, xmm0 ; // перемножить пакеты плавающих точек: xmm1=xmm1\*xmm0  
movups a, xmm1 ; // выгрузить результаты из регистра xmm1 по адресам a  
};

Таким образом, вместо четырех последовательных скалярных умножений мы сделали только одно — векторное.  
Векторные процессоры могут значительно ускорить вычисления над большими объемами данных, но сфера их применимости ограничена, далеко не везде применимы типовые операции над фиксированными массивами.

В задачах моделирования реальных процессов и объектов, для которых характерна обработка больших массивов чисел в форме с плавающей запятой, массивы представляются матрицами и векторами, а алгоритмы их обработки описываются в терминах матричных операций. Как известно, основные матричные операции сводятся к однотипным действиям над парами элементов исходных матриц, которые, чаще всего, можно производить параллельно. В универсальных вычислительных системах, ориентированных на скалярные операции, обработка матриц выполняется поэлементно и последовательно. При большой размерности массивов последовательная обработка элементов матриц занимает слишком много времени, что и приводит к неэффективности универсальных ВС для рассматриваемого класса задач. Для обработки массивов требуются вычислительные средства, позволяющие с помощью единой команды производить действие сразу над всеми элементами массивов - средства *векторной обработки*.

1. **Матричные системы**

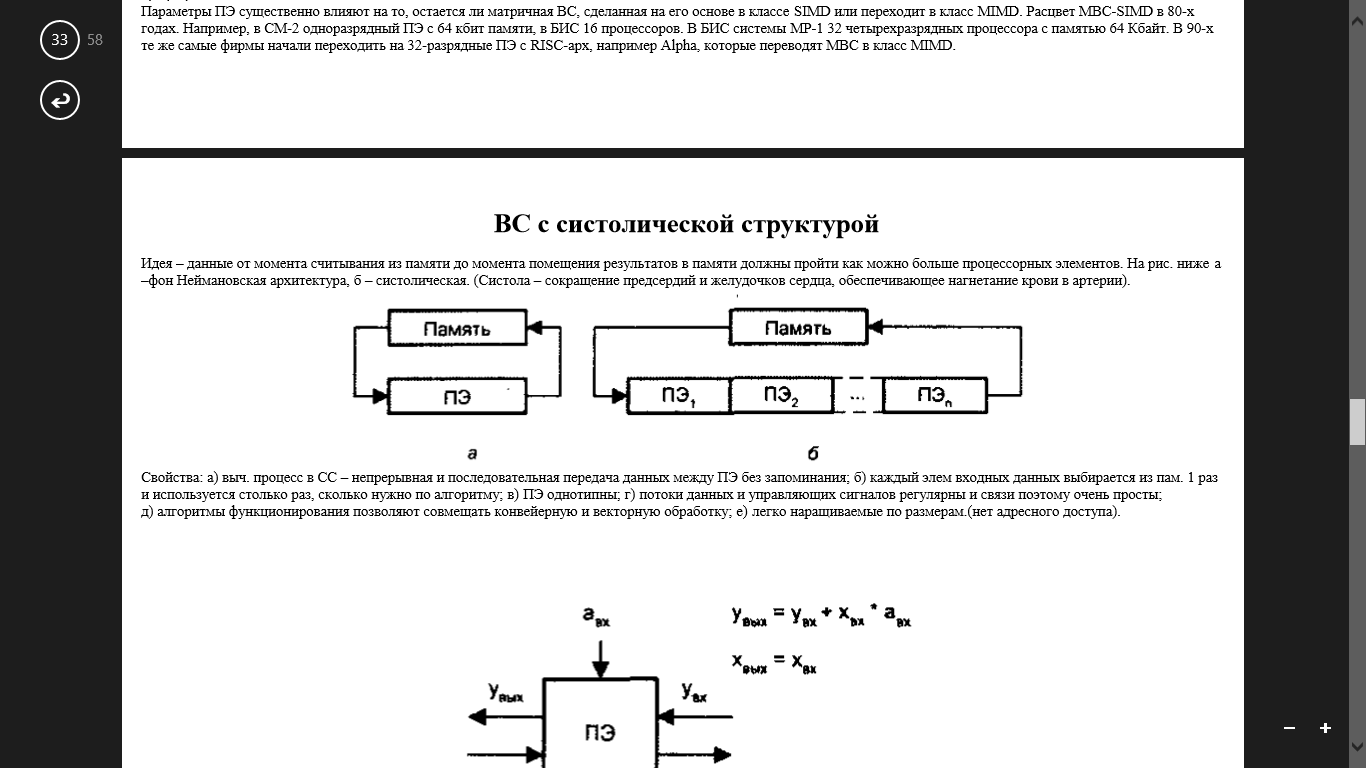
*Массив процессоров работает синхронно под управлением контроллера массива (КМП). КМП генерирует единый поток команд и через шину широковещательной рассылки передает команды всем процессорам, которые должны участвовать в выполнении этой команды.*

*Матричные вычислительные системы* - наиболее распространенные представители класса SIMD, лучше всего приспособленные для решения задач, характеризующихся параллелизмом независимых данных. Матричная система состоит из множества процессорных элементов, работающих параллельно и обрабатывающих свой поток данных.

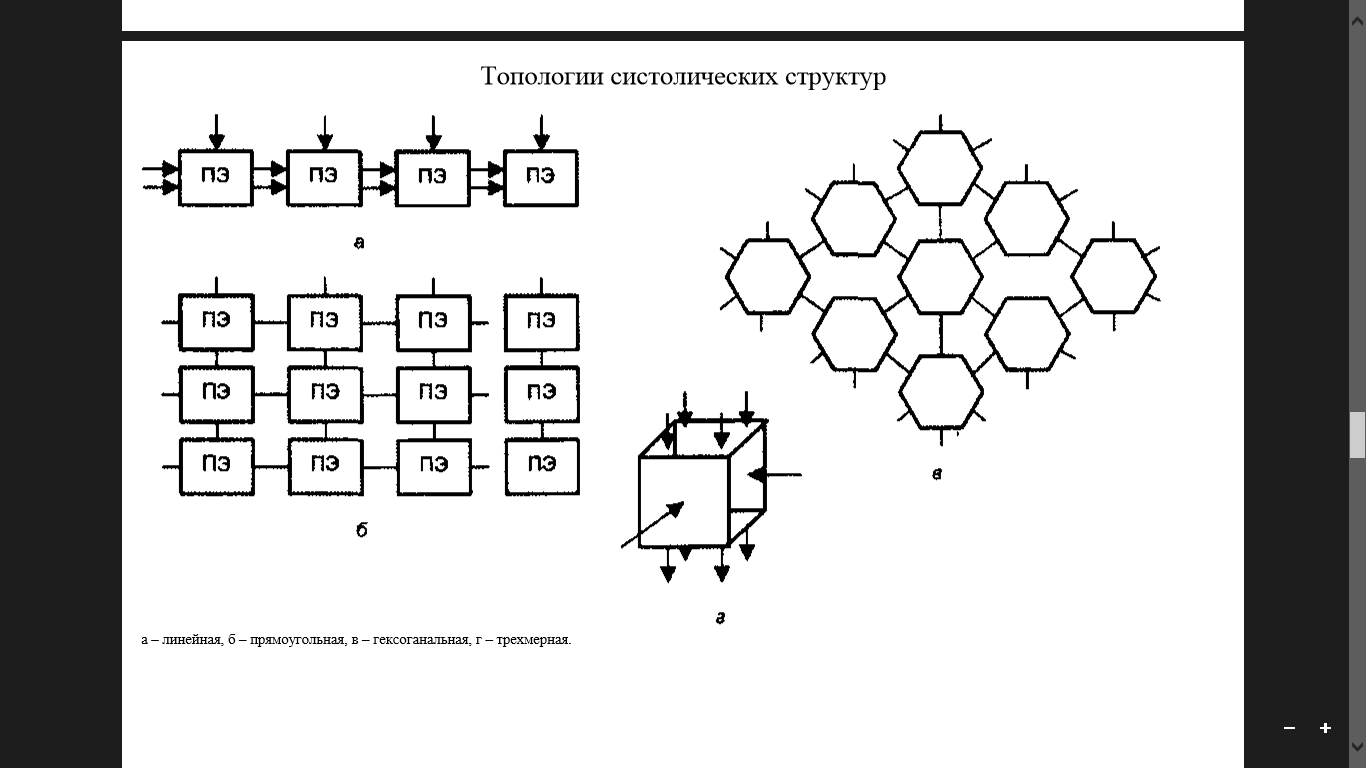
Назначение *матричных вычислительных систем* - обработка больших массивов данных. В основе матричных систем лежит *матричный процессор* (array processor), состоящий из массива процессорных элементов (ПЭ). Такие системы имеют общее управляющее устройство, генерирующее поток команд, и большое число процессорных элементов, работающих параллельно и обрабатывающих каждый свой поток данных. *Однако на практике, чтобы обеспечить достаточную эффективность системы при решении широкого круга задач, необходимо организовать связи между процессорными элементами так, чтобы наиболее полно загрузить процессоры работой. Именно характер связей между ПЭ и определяет разные свойства системы. Подобная схема применима и для векторных вычислений.*Между матричными и векторными системами есть существенная разница. Матричный процессор интегрирует множество идентичных функциональных блоков (ФБ), логически объединенных в матрицу и работающих в SIMD-стиле. Не столь существенно, как конструктивно реализована матрица процессорных элементов - на едином кристалле или на нескольких. Важен сам принцип - ФБ логически скомпонованы в матрицу и работают синхронно, то есть присутствует только один поток команд для всех. Векторный процессор имеет встроенные команды для обработки векторов данных, что позволяет эффективно загрузить конвейер из функциональных блоков.

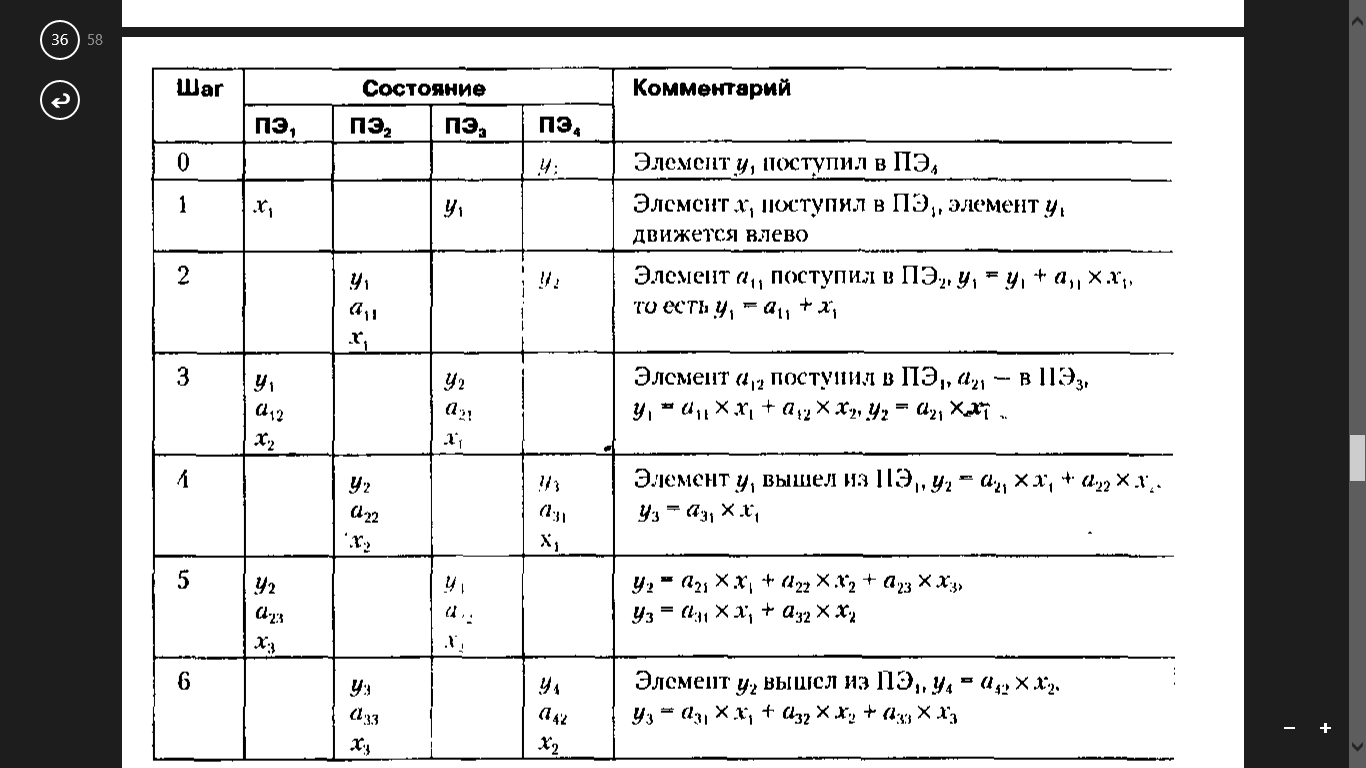
1. **Систолические системы**

Идея – данные от момента считывания из памяти до момента помещения результатов в памяти должны пройти как можно больше процессорных элементов. На рис. a) ниже –фон Неймановская архитектура, б – систолическая. (Систола – сокращение предсердий и желудочков сердца, обеспечивающее нагнетание крови в артерии).



Свойства: а) выч. процесс в СС – непрерывная и последовательная передача данных между ПЭ без запоминания; б) каждый элем входных данных выбирается из пам. 1 раз и используется столько раз, сколько нужно по алгоритму; в) ПЭ однотипны; г) потоки данных и управляющих сигналов регулярны и связи поэтому очень просты; д) алгоритмы функционирования позволяют совмещать конвейерную и векторную обработку; е) легко наращиваемые по размерам. (нет адресного доступа).





1. **ВС с командными словами сверхбольшой длины (VLIW)**

**Параллелизм на уровне инструкций**  
Возьмем простенькую программу:   
a = 1  
b = 2  
c = a + b  
  
Первые две инструкции вполне можно выполнять параллельно, только третья от них зависит. А значит — всю программу можно выполнить за два шага, а не за три.  
  
**Процессор, который умеет сам определять независимые и непротиворечащие друг другу инструкции и параллельно их выполнять, называется суперскалярным.**   
Но есть и другой путь: упростить процессор и возложить поиск параллельности на компилятор. Процессор при этом выполняет команды «пачками», которые заготовил для него компилятор программы, в каждой такой «пачке» — набор инструкций, которые не зависят друг от друга и могут исполняться параллельно. Такая архитектура называется **VLIW (very long instruction word — «очень длинная машинная команда»).**

Одна команда такой архитектуры задает сразу несколько двухместных и/ил одноместных операций. Компилятор переводит исходный текст в промежуточный код, где фигурируют операции, каждую из которых может выполнить функциональный блок системы. Затем исследуются возможности такого распараллеливания, при котором несколько операций объединяются в пакет, задаваемый одним длинным командным словом (обычно от 256 до 1024 бита). При этом обеспечиваются три главных свойства у набора операций, кодируемых одной командой: а) возможность исполняться без конфликтов(обязат. условие); б) число объединяемых операций должно быть близко к числу функциональных блоков (желательное); в) набор операций должен покрываться функциями блоков. Число полей VLIW-команды от 3 до 20. Большинство сверхпроизводительных сигнальных процессоров и мультимедийных процессоров строятся по архитектуре VLIW.

1. **ВС с явным параллелизмом (EPIC)**

**То же что и выше**

Архитектура *EPIC* имеет следующие особенности для устранения недостатков VLIW:

* Каждая группа из нескольких инструкций называется бандлом (*bundle*). Каждый бандл может иметь стоповый бит, обозначающий, что следующая группа зависит от результатов работы данной. Такой бит позволяет создавать будущие поколения архитектуры с возможностью параллельного запуска нескольких бандлов. Информация о зависимостях вычисляется компилятором, и поэтому аппаратуре не придется проводить дополнительную проверку независимости операндов.
* Для предподкачки данных используется инструкция программной подкачки (software prefetch). Предподкачка увеличивает вероятность того, что к моменту исполнения команды загрузки данные уже будут в кеше. Также в этой инструкции могут быть дополнительные указания для выбора различных уровней кеша для данных.
* Инструкция спекулятивной загрузки используется для загрузки данных до того, как станет известно, будут ли они использованы (bypassing control dependencies), или будут они изменены перед использованием (bypassing data dependencies).
* Инструкции проверки загрузки (check load instruction) помогают инструкциям спекулятивной загрузки при помощи проверок, зависела ли инструкция загрузки от последующей записи. В случае наличия подобной зависимости спекулятивная загрузка должна быть повторена.

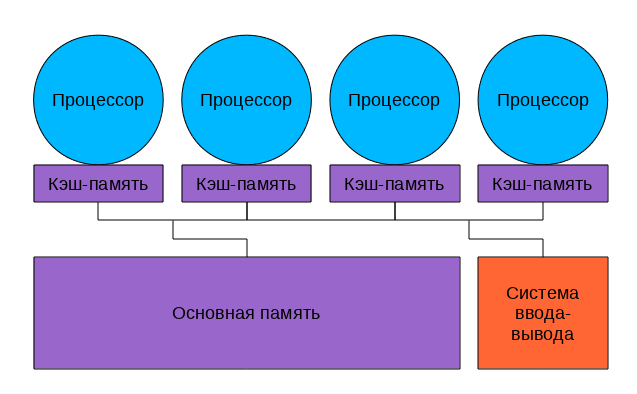
Архитектура *EPIC* также включает в себя несколько концепций (*grab-bag*) для увеличения *ILP* (параллелизма инструкций):

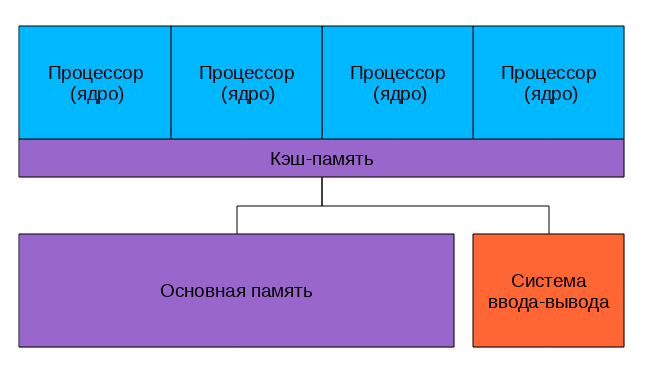
* [Предсказание ветвлений](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D0%BA%D0%B0%D0%B7%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2%D0%B5%D1%82%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9) используется, чтобы снизить частоту переходов и для увеличения [спекулятивности исполнения](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%BF%D0%B5%D0%BA%D1%83%D0%BB%D1%8F%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) инструкций. В последнем случае условное ветвление преобразуется в заполнение предикатных регистров, затем выполняются обе ветви. Результат той ветви, которая не должна была выполняться, отменяется по значению предикатного регистра.
* Отложенные исключительные ситуации, использующие бит [Not a thing](https://ru.wikipedia.org/w/index.php?title=Not_a_thing&action=edit&redlink=1" \o "Not a thing (страница отсутствует)) в регистрах общего назначения. Они позволяют продолжать спекулятивное исполнение даже после исключительных ситуаций.
* Крайне большой [регистровый файл](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB), чтобы избежать необходимости в [переименовании регистров](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%B8%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D1%80%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D0%BE%D0%B2).

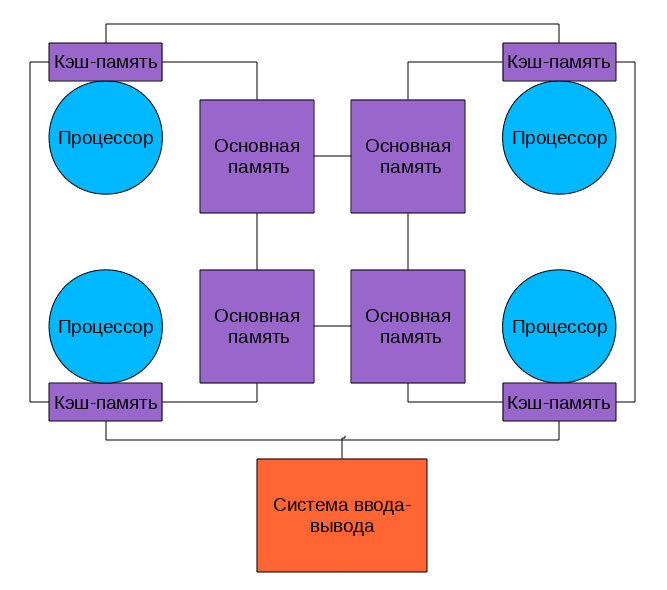
В архитектуре [Itanium](https://ru.wikipedia.org/wiki/Itanium" \o "Itanium) также был добавлен [вращающийся регистровый файл](https://ru.wikipedia.org/w/index.php?title=%D0%92%D1%80%D0%B0%D1%89%D0%B0%D1%8E%D1%89%D0%B8%D0%B9%D1%81%D1%8F_%D1%80%D0%B5%D0%B3%D0%B8%D1%81%D1%82%D1%80%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB&action=edit&redlink=1)[[3]](https://ru.wikipedia.org/wiki/EPIC_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BC%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0)#cite_note-3), необходимый для упрощения [программной конвейеризации циклов](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%B0%D1%8F_%D0%BA%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)(*software pipelining*). При наличии такого файла исчезает необходимость в ручной [раскрутке циклов](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D1%81%D0%BA%D1%80%D1%83%D1%82%D0%BA%D0%B0_%D1%86%D0%B8%D0%BA%D0%BB%D0%BE%D0%B2&action=edit&redlink=1) и ручного переименования регистров.[[4]](https://ru.wikipedia.org/wiki/EPIC_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BC%D0%B8%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80%D0%B0)#cite_note-anand-4)

1. **Симметричные мультимикропроцессорные системы**

##### **Мультипроцессор**

**Мультипроцессор** — это компьютерная система, которая содержит несколько процессоров и одно видимое для всех процессоров. адресное пространство.  
Мультипроцессоры отличаются по организации работы с памятью.  
  
**Системы с общей памятью**  
В таких системах множество процессоров (и процессорных кэшей) имеет доступ к одной и той же физической оперативной памяти. Такая модель часто называется симметричной мультипроцессорностью (SMP). Доступ к памяти при таком построении системы называется UMA (uniform memory access, равномерный доступ) т.к. любой процессор может обратиться к любой ячейке памяти и скорость этого обращения не зависит от адреса памяти. Однако каждый микропроцессор может использовать свой собственный кэш.  
  
Несколько подсистем кэш-памяти процессоров, как правило, подключены к общей памяти через шину  
  
Посмотрим на рисунок.   
Что у нас хорошего?  
Любой процессор обращается ко всей памяти и вся она работает одинаково. Плохо то, что все процессоры обращаются к памяти через шину, и с ростом числа вычислительных ядер пропускная способность этой шины быстро становится узким местом.  
Добавляет головной боли и проблема обеспечения когерентности кэшей.

**Когерентность кэша**  
Допустим, у нас есть многопроцессорный компьютер. Каждый процессор имеет свой кэш, ну, как на рисунке вверху. Пусть некоторую ячейку памяти читали несколько процессоров — и она попала к ним в кэши. Ничего страшного, пока это ячейка неизменна — из быстрых кэшей она читается и как-то используется в вычислениях.   
Если же в результате работы программы один из процессоров изменит эту ячейку памяти, чтоб не было рассогласования, чтоб все остальные процессоры «видели» это обновление придется изменять содержимое кэша **всех** процессоров и как-то тормозить их на время этого обновления.  
Хорошо если число ядер/процессоров 2, как в настольном компьютере, а если 8 или 16? И если все они обмениваются данными через одну шину?  
Потери в производительности могут быть очень значительные.  
  
**Многоядерные процессоры**  
Как бы снизить нагрузку на шину?  
Прежде всего можно перестать её использовать для обеспечения когерентности. Что для этого проще всего сделать?  
Да-да, использовать общий кэш. Так устроены большинство современных многоядерных процессоров.  
  
Посмотрим на картинку, найдем два отличия от предыдущей.  
Да, кэш теперь один на всех, соответственно, проблема когерентности не стоит. А еще круги превратились в прямоугольники, это символизирует тот факт, что все ядра и кэши находятся на одном кристалле. В реальной действительности картинка несколько сложнее, кэши бывают многоуровневыми, часть общие, часть нет, для связи между ними может использоваться специальная шина, но все**настоящие** многоядерные процессоры не используют внешнюю шину для обеспечения когерентности кэша, а значит — снижают нагрузку на нее.  
Многоядерные процессоры — один из основных способов повышения производительности современных компьютеров.  
Уже выпускаются 6 ядерные процессоры, в дальшейшем ядер будет еще больше… где пределы?  
Прежде всего «ядерность» процессоров ограничивается тепловыделением, чем больше транзисторов одновременно работают в одном кристалле, тем больше этот кристалл греется, тем сложнее его охлаждать.  
А второе большое ограничение — опять же пропускная способность внешней шины. Много ядер требуют много данных, чтоб их перемалывать, скорости шины перестает хватать, приходится отказываться от SMP в пользу  
  
**NUMA**

**NUMA (Non-Uniform Memory Access — «неравномерный доступ к памяти» или Non-Uniform Memory Architecture — «Архитектура с неравномерной памятью») — архитектура, в которой, при общем адресном пространстве, скорость доступа к памяти зависит от ее расположения**Обычно у процессора есть " своя" память, обращение к которой быстрее и «чужая», доступ к которой медленнее.  
В современных системах это выглядит примерно так  
  
  
  
Процессоры соединены с памятью и друг с другом с помощью быстрой шины.  
Т.к. нет общей для всех шины то, при работе со «своей» памятью, она перестает быть узким местом системы.   
NUMA архитектура позволяет создавать достаточно производительные многопроцессорные системы, а учитывая многоядерность современных процессоров получим уже очень серьезную вычислительную мощность «в одном корпусе», ограниченную в основном сложностью обеспечения кэш-когерентности этой путаницы процессоров и памяти.

1. **Кластерные вычислительные системы и их программирование**

Мультикомпьютер — вычислительная система без общей памяти, состоящая из большого числа взаимосвязанных компьютеров (узлов), у каждого из которых имеется собственная память. При работе над общей задаче узлы мультикомпьютера взаимодействуют через отправку друг другу сообщений.   
Современные мультикомпьютеры, построенные из множества типовых деталей, называют вычислительными кластерами.  
Большинство современных суперкомпьютеров построены по кластерной архитектуре, они объединяют множество вычислительных узлов с помощью быстрой сети (Gigabit Ethernet или InfiniBand) и позволяют достичь максимально возможной при современном развитии науки вычислительной мощности.   
Проблемы, ограничивающие их мощность, тоже немаленькие  
Это:  
1) Программирование системы с параллельно работающими тысячами вычислительных процессоров  
2) Гигантское энергопотребление  
3) Сложность, приводящая к принципиальной ненадежности