## 1. Причины возникновения многопроцессорных систем (МВС). Преимущества и недостатки МВС. Области применения многопроцессорных систем.

Причины:

- Необходимость в снижении удельного энергопотребления вычислительных систем, а следовательно остановка роста частоты процессоров
- Когда тактовую частоту процессора и ширину конвейра стало невозможно увеличивать, появилась необходимость в переходе к параллельным вычислениям

Преимущества:

- Позволяет увеличить скорость вычислений, не увеличивая частоту процессора

Недостатки:

- Не даёт выигрыша в скорости при вычислении последовательных алгоритмов

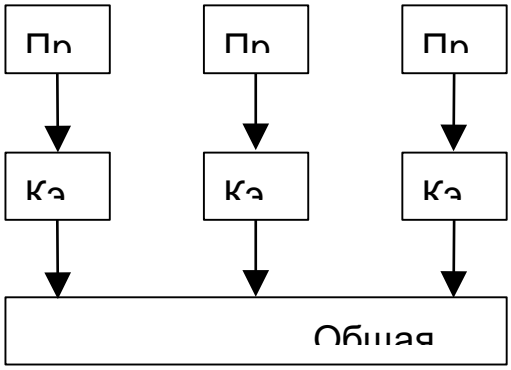
Области применения:

- Сокращение времени решения вычислительно сложных задач. **Пример применения: системы для научных вычислений.**
- Сокращение времени обработки больших объёмов данных. **Пример применения: обработка банковских данных.**
- Решение задач реального времени (Например: один процессор может б**о**льшую часть времени находится в режиме ожидания, но оперативно производить вычисление, когда к нему обратились. Т.е. упор на скорость, но не на эффективность) **Пример применения: системы с необходимым оперативным пользовательским интерфейсом.**
- Создание систем высокой надёжности (Например: если отказал или ошибся один из процессоров, то он перезапускается) **Пример применения: на космическом оборудовании, где вычислительная ошибка может стоить потери аппарата.**
    - Примечание: если вероятность ошибки = β, то вероятность ошибки в n-процессорной системе = (1-β)n

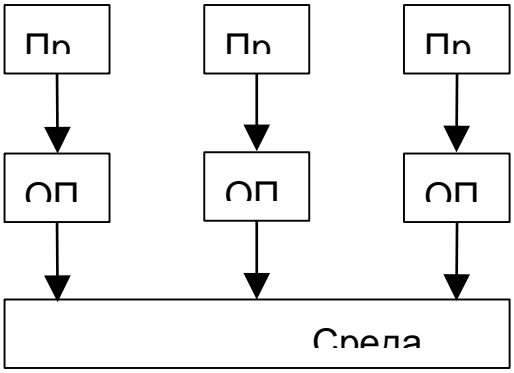
## 2. Отличия между параллельными и последовательными вычислительными архитектурами с точки зрения принципов фон Неймана.

Два основных вида параллельных архитектур:

- Вычислительные системы с общей памятью




- Вычислительные системы с распределённой памятью




Различия в соответствии принципам архитектуры фон Неймана:

- **Принцип программного управления -** программа выполняется последовательно инструкция за инструкцией
    - Не выполняется ни в одном виде параллельных архитектур, так как отсутствует понятие единого времени. Инструкции выполняются не последовательно, а параллельно.
- **Принцип адресности -** Структурно основная память состоит из пронумерованных ячеек, причем *процессору в произвольный момент доступна любая ячейка*.
    - Нарушается в системах с распределённой памятью, так как процессоры не имеют доступа к оперативной памяти других процессоров

## 3. Виды многопроцессорных систем. МВС с общей памятью UMA, NUMA, ccNUMA. МВС с распределенной памятью, гибридные, векторные.

**С общей памятью:**

- **UMA** (**Uniform Memory Access**) - Все процессоры обращаются к одной оперативной памяти, но у каждого процессора может быть свой кэш.
- **NUMA** (**Non-uniform memory access**) - Системы, состоящие из нескольких узлов, где каждый узел - это блок ОП и несколько процессоров, работающих на ней. Блоки ОП соединены коммутаторами памяти. В отличие от *среды передачи данных*, *коммутатор памяти* обеспечивает то, что любой блок ОП внутри NUMA системы доступен любому процессору. Пример использования: системы, где определённые данные используются почти только одними и тему же процессами.
- **ccNUMA** (**Cache coherent NUMA**) - NUMA с когерентностью кэш-памяти.Системы NUMA состоят из однородных базовых узлов, содержащих небольшое число процессоров с модулями основной памяти. Практически все архитектуры ЦПУ используют небольшое количество очень быстрой неразделяемой памяти, известной как кеш, который ускоряет обращение к часто требуемым данным. В NUMA поддержка когерентности через разделяемую память даёт существенное преимущество в производительности.
- **Основные преимущества**:
    - Высокая скорость

**С распределённой памятью:**

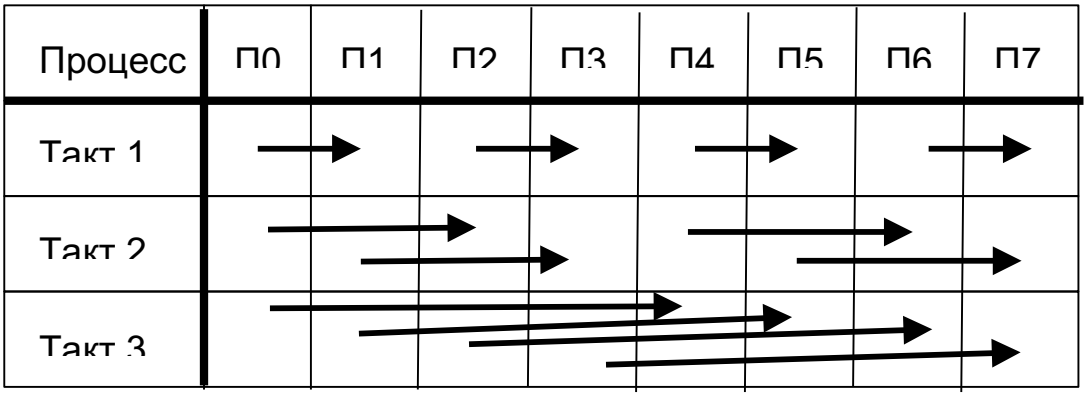
- **Основной принцип**: каждый процессор обращается к своей оперативной памяти. Узлы связаны друг с другом отдельными каналами передачи данных(типа point to point) или одной общей средой передачи данных.
- **Основные преимущества**:
    - Низкая стоимость
    - Высокая масштабируемость
    - Возможность динамически расширять уже существующие системы, подключая к ним новые узлы
- **Гибридные системы**: Системы объединяющие несколько подсистем разного типа. Например системы с распределённой памятью, каждый узел который имеет группу процессоров, работающих на одном блоке ОП.
- **Векторные системы:** Состоят из большого числа идентичных процессорных элементов, имеющих собственную память. Все процессорные элементы в такой машине выполняют одну и ту же

программу. Такая машина, составленная из большого числа процессоров, может обеспечить очень высокую производительность только на тех задачах, при решении которых все процессоры могут делать одну и ту же работу. Например: работа с графикой

## 4. Топологии вычислительных систем.

Под топологией вычислительной системы понимается способ связи отдельных узлов этой системы, то есть правило, по которому решаются, будут ли соединены два узла.
Различные параллельные алгоритмы в ходе работы обмениваются данные с различными узлами, поэтому нельзя сказать, что какая-то топология является оптимальной.
Чем больше связей мы разрешаем, тем быстрее будет происходить обмен данными, но тем дороже будет получившееся решение.
Можно выделить некоторые типичные виды топологий:

- “Звезда”: выбирается некоторый основной узел, все остальные соединяются с ним и только с ним. Такая топология хороша для параллельных программ с центральным планированием выполнения
- “Кольцо”: все узлы нумеруются числами от 0 до n-1, i-ый узел соединяется с i+1 и i-1-ым узлом. При этом первый узел так же соединяется с последним. Такая топология часто используется, например, при использовании геометрического параллелизма.
- “Шина”: все узлы соединены с общей шиной, занимающейся передачей данных. Это -- самая распространённая схема соединения для систем общего назначения. Каждый узел может отправить сообщение каждому узлу за время, не зависящее от расстояния между узлами. Но это время зависит от текущей нагрузки на сеть.
- “Полная”: каждый узел связан с каждым, фактически не используется
- Гиперкуб: Узлы нумеруются от 0 до n-1, i-ый узел связан с i⊕1, i⊕2, i⊕4, i⊕8, …, i⊕(2^k), где ⊕ -- это операция xor. Такая топология использовалась для быстрой работы барьера типа “бабочка”.



**5. Свойства канала передачи данных. Оценка времени передачи данных через канал. Латентность и пропускная способность.**

Для взаимодействия программ, использующих распределённую память, необходимо пользоваться функциями передачи данных между процессорами. Работа этих функций сильно зависит от используемых технологий разработки и оборудования, но возможно ввести некоторые общие характеристики, достаточно хорошо описывающие поведение этих функций:

Пропускная способность сети: время, необходимое на передачу 1 полезного байта между двумя узлами сети

Латентность: время, требующееся для начала осуществления данных. Она обусловлена служебной информацией, используемой для связи, расстоянием между узлами, работой промежуточного сетевого оборудования, и т.д.

Таким образом, можно ввести формулу

Тпередачи(n) = n * Тбайта + Тлатентности

Как правило, Тлатентности >>> (очень много больше) Тбайта. Например, для Gigabit Ethernet, Тлатентности ~ 50мкс, Тбайта ~ 0.015мкс.

Для современных Infiniband, Тлатентности ~ 0.5мкс, Тбайта ~ 3 * 10^-4мкс.

Таким образом, пересылка 1 байта и 100 байт занимают почти одинаковое время.

Следовательно, важно стараться отправлять данные большими кусками, тогда их передача будет на порядок быстрее, чем отправка по отдельности.

## 6.Когерентность кэш-памяти.

При работе машины с памятью возникает узкое место: время доступа процессора к памяти. Поэтому каждый процессор имеет кеш памяти, а именно некоторый небольшой, встроенный в процессор, участок очень быстрой памяти, доступ к которому происходит на порядок быстрее доступа к основной памяти. Современные процессоры имеют различные кеши для инструкций и данных, кроме того, имеют кеши различных уровней (например, индивидуальный для ядра или общий для всего процессора). Однако, наличие отдельного кеша у каждого процессора в многопроцессороной машине приводит к появлению вопросов:

- Как определить, что данные из кеша нужно записать в память?
- Как определить, что данные в кеше нужно обновить из памяти?

В случае, если эти вопросы не решены, разные процессоры могут видеть несогласованные данные, что приведёт к невозможности работы большинства параллельных алгоритмов.

Тривиально требование когерентности формулируется как “данные во всех кешах должны быть одинаковыми”.

Формально когерентности можно сформулировать следующим образом:

- Если программа, выполняющаяся на каком-то процессоре, выполнила запись в ячейку памяти, то через некоторое время программа, выполняющаяся на другом процессоре, должна увидеть в той ячейке записанное значение
- Для каждой ячейки памяти есть последовательность записей, которая наблюдаема программами, выполняющимися на любом процессоре. То есть, если в одну ячейку произошла запись значений A и B, то если программа на одном процессоре прочитала сначала A, а потом B, то программа на другом процессоре не может прочитать сначала B, а потом A. Кроме того, записи, сделанные одним процессором, должны быть видны именно в том порядке, в котором он их делал.

Как правило, для поддержания когерентности используется отдельный протокол взаимодействия кешей. Каждый процессор по общей шине сообщает другим процессорам, что он помещает в кеш, чтобы они могли проверить наличие той же ячейки памяти в своем кеше и пометить её, как требующую особого внимания. Тогда при записи в эту ячейку процессор будет сообщать всем другим процессорам, что нужно либо выкинуть эту ячейку из кеша, либо заменить её значение на новое.

Наличие единственной шины и необходимость её захвата для передачи сообщения как раз гарантирует выполнение второго свойства.

Если запись в общую память с разных процессоров происходит редко (а при большинстве правильно организованных параллельных процессов с общей памятью это так), то расходы на поддержание когерентности кеша низки.

7. Внутренний параллелизм, степень параллелизма.

Не любая задача подается распараллеливанию, даже если для нее известен вполне хороший последовательный алгоритм. Сегодня разработка параллельных программ является скорее искусством, чем технологией. Она предполагает поиски и создание методов, обладающих значительным внутренним параллелизмом. Алгоритм обладает **внутренним параллелизмом**, если в нем присутствуют действия, для которых допустимо одновременное выполнение. Это свойство именно алгоритма, оно не имеет отношения ни к типу вычислительной системы, ни к виду используемого языка программирования. Если на всех его этапах достаточно много взаимозависимых операций, то есть вероятность успешного создания параллельного алгоритма. Если таких операций нет, то для решения задачи на многопроцессорный системе необходимо либо преобразовать алгоритм, либо искать другой метод решения.

**Степень параллелизма алгоритма** - это число операций, выполнять которые можно в любом порядке.

Иногда употребляют термин степень внутреннего параллелизма, подчеркивая тем самым, что имеют ввиду именно внутренние свойства алгоритма, не зависящие от вида и особенностей вычислительных систем. На разных этапах выполнения алгоритма степень параллелизма может быть различной. Важно, чтобы высокой степенью внутреннего параллелизма обладали этапы, на выполнение которых тратится основное время.

8.Ускорение, эффективность, масштабируемость. Закон Амдаля.

Ускорение и эффективность - две основные характеристики, используемые для количественной оценки параллельных алгоритмов.

**Ускорение параллельного алгоритма** - это отношение времени выполнения последовательного алгоритма T1 ко времени выполнения параллельного алгоритма Tp на заданном числе процессоров p.

$S_p = T_1/T_p$

Что подразумевается под временем выполнения? На практике используется два способа оценки этого времени:

1. Определяется число выполняемых алгоритмом операций и полагается, что время выполнения пропорционально этому числу.
2. Заменяется время выполнения программы, реализующий алгоритм на некоторой вычислительной системе.

Недостатки первого. Команды в алгоритме зачастую более высокоуровневые, чем команды процессора. Не всегда можно с достаточной точностью оценить время их выполнения. Также сложно оценить время выполнения *последовательности* этих операций. На процессорах, оснащенных кэш-памятью и конвейерами выполнения команд, время выполнения последовательности операций может быть существенно меньше суммы времен выполнения каждой операции по отдельности, а на вычислительной системе с общей памятью время выполнения параллельного выполнения двух независимых последовательностей команд на двух процессорах может быть существенно больше времени последовательного выполнения этих же последовательностей на одном процессоре (например, зачет конфликтов кэш-памяти).

Недостатки второго. Идет речь не о времени работы алгоритма, а о времени работы готовой программы. Время выполнения может зависеть от параметров вычислительной системы, например, объема ОП.

**Эффективность параллельного алгоритма** - это отношение ускорения к числу процессоров, на которых оно достигнуто:

$E_p = S_p / p$

**Масштабируемость параллельного алгоритма** - наименьшее число процессоров, на которых достигается максимальное ускорение.

m = min(p*): для любого p, Sp* >= Sp

Чем лучше масштабируем алгоритм, тем для большего числа процессоров будет наблюдаться сокращение времени решения задачи. Максимально ускорение, таким образом, равно Sm*.

**Закон Амдаля** утверждает, что максимально достижимое ускорение, независимо от числа используемых процессоров и ограничено следующей величиной:

Sp < 1/lambda,

где lambda - доля операций алгоритма, выполнение которых невозможно одновременно с другими операциями.

9.Сверхлинейное ускорение, возможные причины. Формальное преобразование параллельного алгоритма в последовательный.

**Сверхлинейное ускорение** - ускорение, превышающее число используемых для расчета процессов: Sp > p. Причины:

1. Особенности используемых вычислительных систем. Пример. Запросы к базе данных. Размер базы десятикратно превышает размер ОП. При обработке одним процессором при каждом запросе просматривается вся база, читаются записи с медленного внешнего носителя. Два процессора - база делится пополам, и каждый процессор отвечает за свою половину. Это приведет к ускорению в два раза. Пока объем данных, обрабатываемых каждым из процессоров, превышает объем ОП, рост ускорения будет линейным, поскольку время работы каждого процессора будет определяться временем чтения данных с внешнего носителя. Ситуация меняется, ка только совокупный объем ОП превысит размер базы. После однократного прочтения каждым процессором своей части базы, обращения к внешним медленным носителям больше не потребуется. Весь поиск процессоры будут проводить, используя данные в своей ОП. Поэтому происходит сверхлинейное ускорение.

2. Неудачный выбор последовательного алгоритма. Слишком медленный последовательный алгоритм, с которым происходит сравнение.

**Формальное преобразование параллельного алгоритма в последовательный** (2 процесса):

1. Выбираем для выполнения первый процесс. Выполним все действия, предшествующие операции взаимодействия между процессами.
2. Выбираем для выполнения второй процесс. Выполним все действия, предшествующие операции взаимодействия между процессами.
3. Выполним операции, соответствующие требуемому взаимодействию. Например, если предполагалась передача сообщения, установим значение принимаемой переменной равным значению передаваемой переменной.
4. Перейдем к пункту 1.

10.Накладные расходы в параллельных алгоритмах.

Оценим накладные расходы (total overhead), которые имеют место при выполнении параллельного алгоритма

$T_0 = p*T_p-T_1$
Накладные расходы появляются за счет необходимости организации взаимодействия процессоров, выполнения некоторых дополнительных действий, синхронизации параллельных вычислений и т. п. Используя введенное обозначение, можно получить новые выражения для времени параллельного решения задачи и соответствующего ускорения:

$T_p = (T_1+T_0)/p,$
$S_p = T_1/T_p = p*T_1/(T_1+T_0)$
Используя полученные соотношения, эффективность использования процессоров можно выразить как:

$E_p = S_p/p = T_1/(T_1+T_0) = 1/(1+T_0/T_1)$
Последнее выражение показывает, что если сложность решаемой задачи является фиксированной ($T_1=const$), то при росте числа процессоров эффективность, как правило, будет убывать за счет роста накладных расходов $T_0$.

**11.** Модель параллельной программы, выполняющейся на вычислительной системе с распределенной памятью.

Как правило, на вычислительной системе с распределённой памятью программисту доступно некоторое (заранее неизвестное) количество процессоров, каждый из процессоров имеет доступ к своей локальной памяти, и на каждом из процессоров запускается один и тот же код программы.
Вычислительная система не обязательно должна выполнять каждый экземпляр программы на отдельном узле: они могут выполняться на разных процессорах/ядрах одного узла, или даже в конкурентном режиме на одном ядре одного процессора.
Но с точки зрения программиста у каждого запущенного экземпляра (процесса) есть только изолированная от других процессов область памяти, с которой он работает. Способа непосредственно получить доступ к памяти другого процесса нет, даже если они выполняются на одном узле. Каждая копия запускается на “виртуальном процессоре”, не имеющей с логической точки зрения отличий от физического одноядерного процессора.
При запуске процессу в качестве аргументов передаётся 2 числа: общее число процессов, которые будут запущены, и номер текущего процесса.
Для взаимодействия друг с другом процессы могут использовать некоторые примитивы:
Send(data, n), Recv(data, n): синхронно отправить/получить данные на/от процесса №N
ASend (асинхронная отправка), ABSend (асинхронная буферизирующая отправка), ARecv (асинхронное получение)
Barrier(): Барьер (сначала все процессы входят в эту функцию, затем одновременно выходят)

# 12. Методы передачи данных.

1. Синхронный

При передаче данных адрес и размер передаваемых данных, номер процессора, с которым осуществляется взаимодействие
Send(номер процессора, адрес, размер) — функция синхронной передачи данных
Recv(номер процессора, адрес, размер) — функция синхронного приема данных
Обе функции - блокирующие (процесс переводится в состояние ожидания (пока другая функция не будет вызвана, очевидно)

2. Асинхронный

Не выполняет передачу, а лишь создает заказ на ее выполнение
Небуферизованные функции:
ASend(номер процессора, адрес, размер)
ARecv(номер процессора, адрес, размер)
ASync(номер процессора) — нужно вызывать а) после ASend перед изменением данных, б) после ARecv перед использованием полученных данных
Буферизованная функция:
ABSend(номер процессора, адрес, размер) — заносит данные в буфер и после нее вызов ASync не нужен

## 13. Барьер. Схемы реализации и оценки времени выполнения. Реализация барьера на основе синхронных обменов.

Тип синхронизации взаимодействующих процессов.

**Барьер** - это функция, вызываемая всеми процессами, участвующими в акте взаимной синхронизации. Ни один из процессов не завершает выполнение этой функции, пока все процессы не начнут её выполнение.

*Замечание:* является довольно затратной по времени функцией. Полезны в основном для отладки программ, обладающих сложной логикой выполняемых операций.

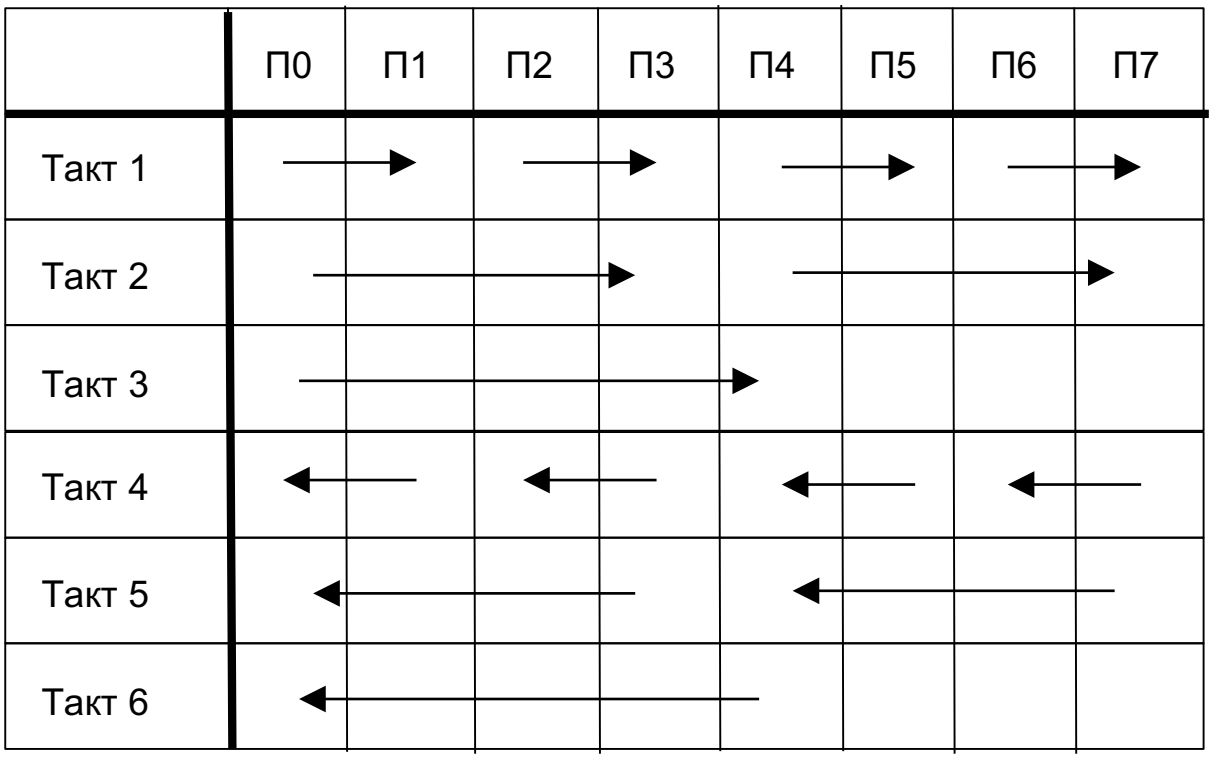
### *Реализация*

**Задача**: Используя передачу сообщений, оповестить все процессы о том, что каждый другой процесс зашёл в барьер.

**Оценка стоимости:** в качестве стоимости алгоритма реализации барьера оценивается минимальное количество тактов, которое проходит с того момента, как все процессоры уже вошли в барьер, до момента, когда все процессоры получили сообщения о том, что они могут выходить из барьера.

**Пример реализации с помощью синхронных обменов:**

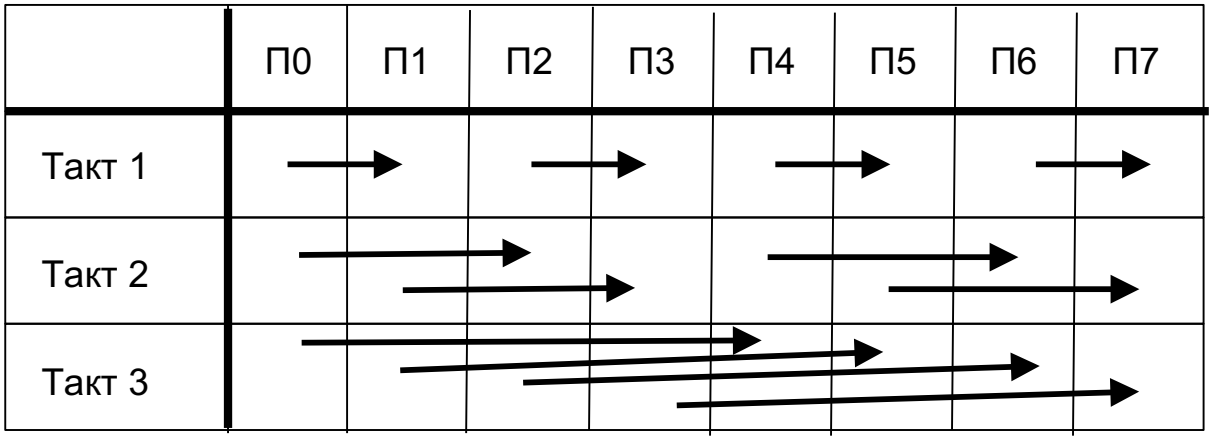
Каждая стрелка символизирует операцию send в процессоре на начале стрелки и операцию recv в процессоре на конце стрелки.




Оценка времени такого метода $T = 2\,\tau_s\, log_2 n$, где $\tau_s$- время одного такта.

**Барьер типа “бабочка”:**

Оценка времени такого метода $T = \tau_s \, log_2 n$. Удобен для использования на топологии *Гиперкуб*.



**Билет №14**

*Модель параллельной программы, выполняющейся на вычислительной системе с общей памятью. Нити (легковесные процессы, threads).*

Изолированный последовательный процесс — последовательность действий, выполняемых автономно (независимо от окружающей обстановки).

Будем считать, что, кроме явных, достаточно редких моментов связи, процессы выполняются совершенно независимо друг от друга. Кроме того, будем считать, что такие процессы связаны слабо (за исключением моментов явной связи, эти процессы рассматриваются как совершенно независимые друг от друга). Будем называть такие процессы слабо связанными последовательными процессами.

Под параллельной программой будем понимать всю совокупность слабо связанных последовательных процессов, необходимых для выполнения требуемых вычислений. Учитывая, что параллельная программа выполняется на наборе процессоров, ей соответствует множество процессов, запущенных на одном или на нескольких процессорах.

**Суть билета:**

Примем следующую модель параллельной программы на общей памяти. Выполнение начинается с запуска единственного последовательно процесса. В тот момент, когда появляется возможность параллельного выполнения команд, вызывается функция порождения дополнительной нити кода (треда, легковесного процесса). Таким образом, появляется возможность параллельного выполнения команд. Число порождаемых нитей определяется требованиями алгоритма и числом доступных процессоров. В общем случае оно вполне может превышать число доступных процессоров, в этом случае нити будут выполняться не в параллельном, а в конкурентном режиме (режиме разделения времени). У каждой нити есть недоступные другим нитям локальные переменные, хранимые в стеке нити. Кроме локальных переменных, каждая из нитей имеет доступ к общим глобальным переменным. Они находятся в общем адресном пространстве. Каждая нить может читать и модифицировать глобальные данные.

15. Разделяемые ресурсы. Семафор, критическая секция, монитор.

**Семафор** — целочисленная неотрицательная переменная, над которой можно выполнять только две атомарные операции P и V
Атомарная операция — если некоторый процесс начал выполнение атомарной операции над переменной, то никакие другие процессы не будут иметь возможность обратиться к этой переменной до тех пор, пока эта операция не будет выполнена

V(S) — атомарная **неблокирующая** операция, которая увеличивает значение семафора S на 1.
P(S) — атомарная блокирующая операция, которая уменьшает значение семафора S на один, если S больше нуля, и блокирует процесс до тех пор, пока S не станет больше нуля, если S=0

Есть бинарные семафоры - которые принимают значение 0 и 1, и семафоры общего вида.

**Критическая секция** — последовательность команд, которая должна выполняться как одна непрерывная операция
По сути, все, что между P(S) и V(S) — и есть критическая секция
Для уменьшения ошибок вида отсутствия одной из операций используется монитор, или использование разных семафоров, когда подразумевается использование одного и того же.

**Монитор** — Набор подпрограмм, включающих в себя все обращения к защищаемому ресурсу.

16 Методы статической и динамической балансировки загрузки процессоров.

Балансировка нагрузки применяется для оптимизации выполнения распределённых (параллельных) вычислений с помощью распределённой (параллельной) ВС. Балансировка нагрузки предполагает равномерную нагрузку вычислительных узлов (процессора многопроцессорной ЭВМ или компьютера в сети). При появлении новых заданий программное обеспечение, реализующее балансировку, должно принять решение о том, где (на каком вычислительном узле) следует выполнять вычисления, связанные с этим новым заданием. Кроме того, балансировка предполагает перенос (migration – миграция) части вычислений с наиболее загруженных вычислительных узлов на менее загруженные узлы. Стратегия балансировки должна быть таковой, чтобы вычислительные узлы были загружены достаточно равномерно, но и коммуникационная среда не должна быть перегружена.

**Балансировка вычислительной нагрузки**
**Причины возникновения несбалансированной нагрузки**
Проблема балансировки вычислительной нагрузки распределённого приложения возникает по той причине, что:

- структура распределенного приложения неоднородна, различные логические процессы требуют различных вычислительных мощностей;
- структура вычислительного комплекса (например, кластера), также неоднородна, т.е. разные вычислительные узлы обладают разной производительностью;
- структура межузлового взаимодействия неоднородна, т.к. линии связи, соединяющие узлы, могут иметь различные характеристики пропускной способности.

**Статическая балансировка -** выполняется до начала выполнения распределенного приложения. Очень часто при распределении логических процессов по процессорам используется опыт предыдущих выполнений, применяются генетические алгоритмы. Однако предварительное размещение логических процессов по процессорам (компьютерам) не даёт эффекта.

Это объясняется тем, что:

- Может измениться вычислительная среда, в которой происходит выполнение приложения, какой либо вычислительный узел может выйти из строя.

- Вычислительный узел, на котором выполняется распределенное приложение, занят ещё и другими вычислениями, доля которых со временем может возрасти.

Методы:

- метод сдваивания
- геометрический параллелизм
- конвейерный параллелизм

**Динамическая балансировка** - предусматривает перераспределение вычислительной нагрузки на узлы во время выполнения приложения. Программное обеспечение, реализующее динамическую балансировку, определяет:

- загрузку вычислительных узлов;
- пропускную способность линий связи;
- частоту обменов сообщениями между логическими процессами распределенного приложения и др.

На основании собранных данных как о распределенном приложении, так и вычислительной среде) принимается решение о переносе логических процессов с одного узла на другой.

Методы

- коллективное решение
- диффузная балансировка загрузки

**Постановка задачи динамической балансировки**

Цель балансировки загрузки может быть сформулирована следующим образом:

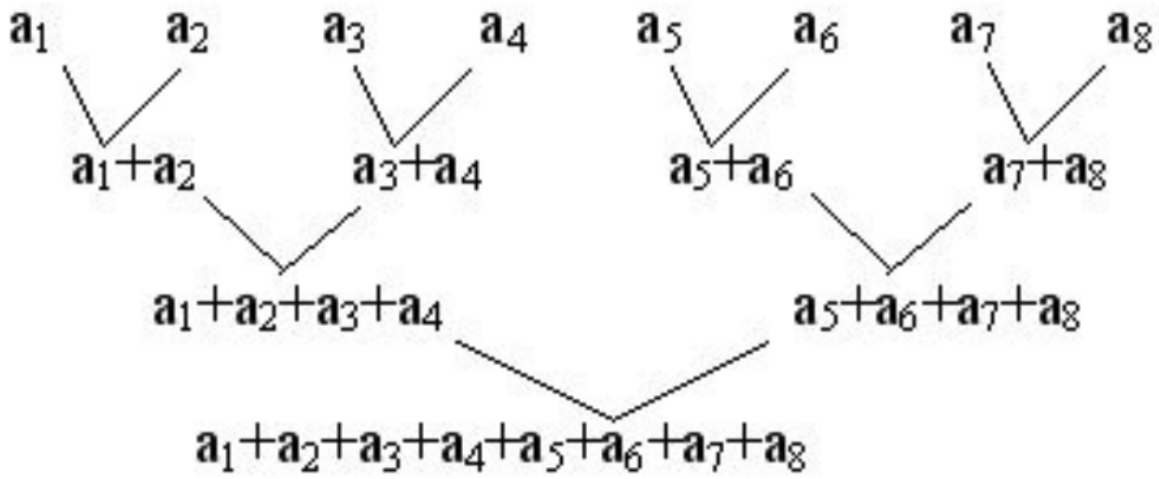
Исходя из набора задач, включающих вычисления и передачу данных, и сети компьютеров определенной топологии, найти такое распределение задач по компьютерам, которое обеспечивает примерно равную вычислительную загрузку компьютеров и минимальные затраты на передачу данных между ними.

Обычно практичное и полное решение задачи балансировки загрузки состоит из четырех шагов:

- Оценка загрузки вычислительных узлов.
- Инициация балансировки загрузки.
- Принятие решений о балансировке.
- Перемещение объектов.

17 Метод сдваивания.

Пусть требуется найти сумму восьми элементов массива, имея в распоряжении четыре процессора. На первом шаге на каждом из процессоров найдем сумму двух последовательно идущих элементов. На втором шаге на двух процессорах найдем суммы результатов, полученных на первом шаге. На третьем шаге на одном процессоре получим искомый результат (рисунок 1).




Если **n** - количество элементов массива, **p** - число процессоров,  - время выполнения одной операции (одной задачи), то получим следующие оценки ускорения, эффективности и времени выполнения алгоритма:

$$T_{p=n/2}(n) = \tau_c \log_2 n,$$

$$S_{p=n/2}(n) = \frac{n-1}{\log_2 n},$$

$$E_{p=n/2}(n) \approx \frac{1}{\log_2 n}.$$

Ускорение достаточно велико, но эффективность невелика, особенно для большого количества элементов массива, так как все процессоры в этой схеме задействованы только на первом шаге. При решении некоторых задач эффективность можно поднять, заменив обычную схему на модифицированную каскадную схему.

**Модифицированная каскадная схема**

В модифицированной каскадной схеме на первом этапе все суммируемые значения подразделяются на **n/log₂n** групп, в каждой из которых содержится **log₂n** элементов; для каждой группы вычисляется сумма значений при помощи последовательного алгоритма суммирования. На втором этапе для полученных **n/log₂n** сумм отдельных групп применяется обычная каскадная схема (рисунок 2).

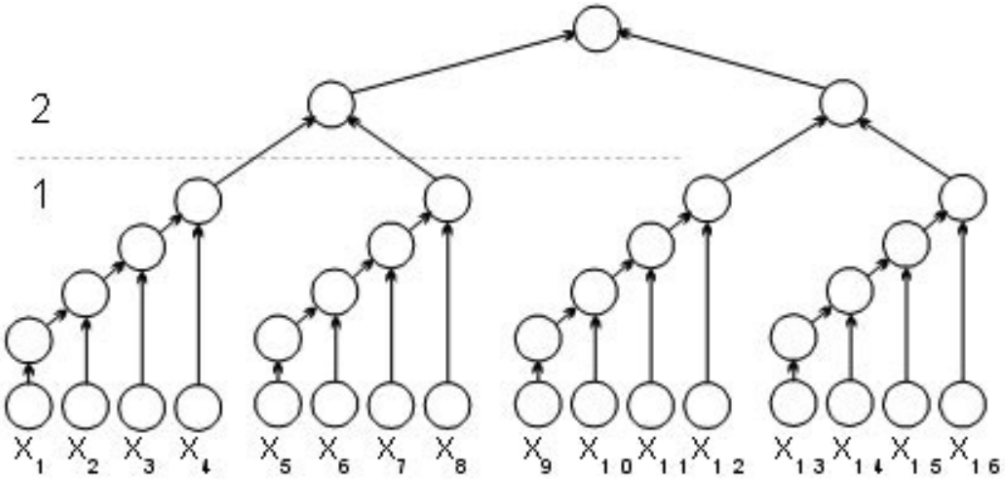



Рис.2. Модифицированная каскадная схема

Показатели ускорения и эффективности модифицированной каскадной схемы определяются соотношениями:

$$T_p = \frac{n}{p}\tau_c + (\tau_c + \tau_s)\log_2 p$$

$$T_{p=\frac{n}{\log_2 n}}(n) \approx 2\tau_c \log_2 n, \qquad \tau_s = 0$$

$$S_{p=\frac{n}{\log_2 n}}(n) \approx \frac{n}{2\log_2 n - \log_2 \log_2 n} \approx \frac{n}{2\log_2 n} = \frac{p}{2}$$

$$E_{p=\frac{n}{\log_2 n}}(n) \approx \frac{1}{2}$$

По сравнению с обычной каскадной схемой ускорение уменьшилось в 2 раза, но эффективность выросла почти до 0,5.

***18 Метод геометрического параллелизма.***

Распределим данные по процессорам равными частями и поручим обработку каждой части соответсвующему процессу. Подход эффективен при условии, что действия, выполняемые одним процессом, зависят лишь от небольшого, ограниченно объема данных, расположенных на других процессорах.

Рассмотрим на примере задачи построения ***стены Фокса***. Стена состоит из кирпичей, разбиваем кладку на равные по длине участки и поручаем постройку каждого участка отдельному каменщику(процессу). Каменщики могут начать укладку одновеременно, но перед укладываем очередного слоя, должны убедиться, что кирпичи предыдущего слоя уложены, не только на его участки, а также на соседних. Если еще не уложены - возникают паузы из-за синхронизации работы.

Пусть **n** - ширина стены (число кирпичей в нижнем ряду), **k** - высота стены, $\tau_c$ — время укладки одного кирпича, $T_p$ — время решения задачи с помощью $p$ процессов

Тогда $T_1$(kn) = $\tau_c$kn - время укладки kn кирпичей одним процессом

Для определенности, пусть процессы выполняются на вычислительное системе с распр. памятью. У каждого процесса свой участок кирпичей длинной n/p. Каждый процесс на каждом слое должен передать и получить сообщения от левого и правого соседа поэтому добавляется 4k$\tau_s$. Значение ускорения и эффективности не зависят от высоты стены, а только от ширины, точнее от $\gamma = 4\frac{p\,\tau_s}{n\,\tau_c}$. Уменьшение $\gamma$ ведет к увеличения ускорения практически до p, а эффективность практически до 100%. Даже при относительно большом времени на передачу данных, можно найти такое n, при котором $\gamma$будет существенности меньше 1, и эффективность достаточно высокой. Поэтому метод геометр параллелизма позволяет эффективно использовать многопроцессорные системы, содержание большое число процессов. Метод является статическим, так как до начала выполнения расчетов определяется, какую часть работы будет выполнять каждый процесс и потом это значение не меняется. Оценки справедливы только при выполнении ряда условий:

1) кирпичи распределены строго поровну между процессами
2) число операций, необходимых для каждого кирпича не зависит от его положения в стене
3) все процессы выполняют вычислительные операции с одинаковой скоростью
4) все каналы передачи данных имеют одинаковы характеристики.

$$T_1(kn) = \tau_c kn,$$

$$T_p(kn) = \tau_c \frac{kn}{p} + 4k\tau_c,$$

$$S_p(kn) = p \frac{1}{1 + 4\frac{p}{n}\frac{\tau_s}{\tau_c}},$$

$$E_p(kn) = \frac{1}{1 + 4\frac{p}{n}\frac{\tau_s}{\tau_c}}.$$

При увеличении числа процессоров сокращается время выполнения вычислений, но увеличивается время передачи данных.

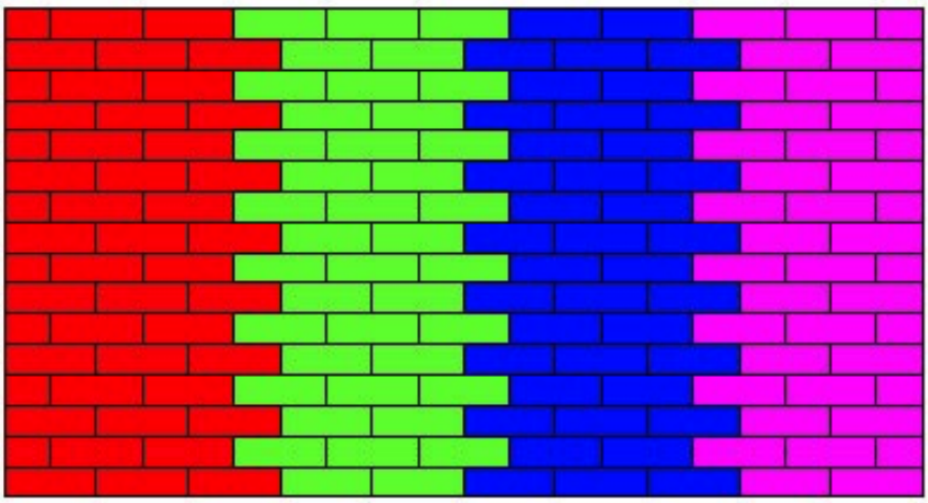


Рис.2. Метод геометрического параллелизма

Метод позволяет получать высокие ускорения, если все процессоры работают синхронно, с одинаковой скоростью. Широко применяется при решении задач математической физики, обработке изображений и во многих других. Ориентирован на выполнение множества однотипных действий над однородными взаимосвязанными данными.

**19 - Метод коллективного решения.**

**Суть:** разбиваем задачу на множество равных блоков и назначаем один из процессов управляющим. Управляющий процесс динамически распределяет блоки между остальными процессами. То есть каждый обрабатывающий процесс после вычисления своего блока обращается к управляющему за новым блоком. При этом невозможно заранее предсказать, какие блоки будут обработаны тем или иным процессом, это определяется непосредственно в ходе решения задачи.

Основные причины, приводящие к простою процессов при таком подходе:

1. Требуется синхронизация между вправляющим и обрабатывающими процессами: при выборе номера очередного обрабатываемого блока и при передаче процессу информации о выбранном блоке. На выполнение эти действие требуется время $\tau s1$, поэтому в начале работы все процессы, кроме первого, будут простаивать в ожидании своего первого задания.
2. Каждый выполнивший очередное задание процесс затратит время $\tau s2$ на передачу результата управляющему процессу. Если несколько процессов закончат работу одновременно, почти все они также будут находиться в состоянии ожидания готовности управляющего процесса к приему данных.
3. Вероятен простой части процессов в конце работы, когда все задания уже распределены по процессам. Процессы, окончившие работу раньше других, новой работы не получат, и их вычислительная мощность не будет использована в это время для решения задачи.

Оценим эффективность метода при следующих предположениях:

- Все процессоры имеют одинаковую производительность
- Каждое задание обрабатывается за время $\tau c$
- Время на передачу данных не зависит ни от номера задания, ни от номера процесса и составляет $\tau s=\tau s1+\tau s2$
- Работа выполняется на p обрабатывающих процессах и одном управляющем.

При сделанных предположениях каждому процессу будет направлено на обработку Np заданий, где N=kn. Время, затраченное на получение данных о каждом задании и на его обработку, составит $(\tau c+\tau s)$. Будем считать $p \ll N$ и пренебрежем простоями в начале и в конце работы. Тогда:

$$Tp=Np(\tau c+\tau s), Sp=p\tau c\tau c+\tau s,\ Ep=1/(1+\tau s\tau c)$$

Если $\tau s \ll \tau c$, то метод обладает высокой эффективностью. В противном случае следует изменить стратегию распределения заданий, увеличив объем

работы, передаваемой обрабатывающему процессу при каждом обращении к управляющему процессу. Пусть за одно обращение передается информация о r заданиях. Назовем такое задание, содержащее r заданий, расширенным. Тогда время на получение данных и обработку одного расширенного задания составит rτc+Ts, где Ts — время передачи данных о r заданиях. Число расширенных заданий, обрабатываемым каждым из процессов составит N/rp.

$$Tp=Nrp(r\tau c+Ts),Sp=p/(1+Tsr\tau s),\ Ep=1/(1+Tsr\tau s)$$

При этом эффективность возрастет только при условии, что Ts не зависит от числа элементарных заданий, передаваемых на обработку за одно обращение к серверу (управляющему процессу). Для ряда задач это действительно так, например, для задачи вычислений интеграла объем передаваемых данных не зависит от трудоемкости задания (передаются только координаты начала и конца обрабатываемого отрезка + результаты вычисления). При этом стоит помнить, что с ростом величины r падает степень параллелизма.

Кроме того, нужно помнить, что в системе присутствует не масштабируемый элемент — управляющий процессор, таким образом не имеет смысла иметь более, чем pmax обрабатывающих процессов, в противном случае они будут простаивать в ожидании задания от управляющего процесса.

Метод коллективного решения необходим именно тогда, когда время обработки заданий различно т априори неизвестно. Тогда, при использовании больших значений величины r может возникнуть ситуация, в которой некоторому процессу назначено расширенное задание, содержащее большое число трудоемких элементарных заданий. В результате, возможен существенный простой всех остальных процессоров в конце работы. Они могли бы принять участие в обработке сложных заданий, если бы было выбрано меньшее значение r. Таким образом, конкретное значение r при распределении каждого очередного задания следует выбирать, опираясь на дополнительную информацию о свойствах решаемой задачи и о числе еще не обработанных заданий.

20 - Метод конвейерного параллелизма.

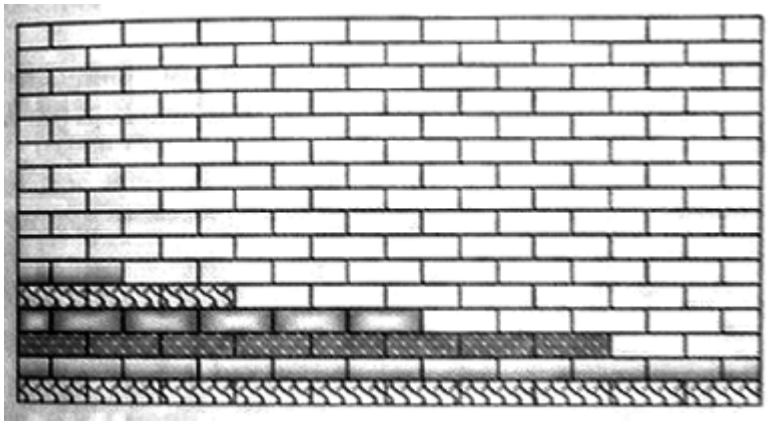
Пусть требуется вычислить:

```
for (step=1;step<=k;step++) {
        fnew[0]=g(step);
        for (i=1;i<n;i++)
                fnew[i]=fnew[i-1]+f[i];
        for (i=1;i<n;i++)
                f[i]=fnew[i]
}
```

Результат зависит от значения с индексом i-1 на текущем слое (step) и от значения с индексом i на предыдущем.

Геометрический параллелизм здесь не подходит (возможно только последовательно определение величин fnew). Можно организовать вычисление с помощью конвейерного метода.

Первый процесс вычисляет значения на первом слое. Второй – на втором и т.д. После того, как первый завершает вычисление на первом слое, он переходит к слою $p + 1$. Очевидно, что перейти к вычислению слоя $p + 1$ может только первый процесс, поскольку остальные вынуждены ожидать, пока будут вычислены нижние слои.



Для конвейерного параллелизма существенно условие одинаковости времени вычисления каждого шага. Если это время будет значительно различаться, эффективность будет низкой. Если процессоры имеют разную производительность, то общая производительность вычислений будет определяться самым медленным из них.

Эффективность данной стратегии не так высока, как у метода геометрического параллелизма. Главная причина низкой эффективности кроется в необходимости получения перед вычислением очередного шага

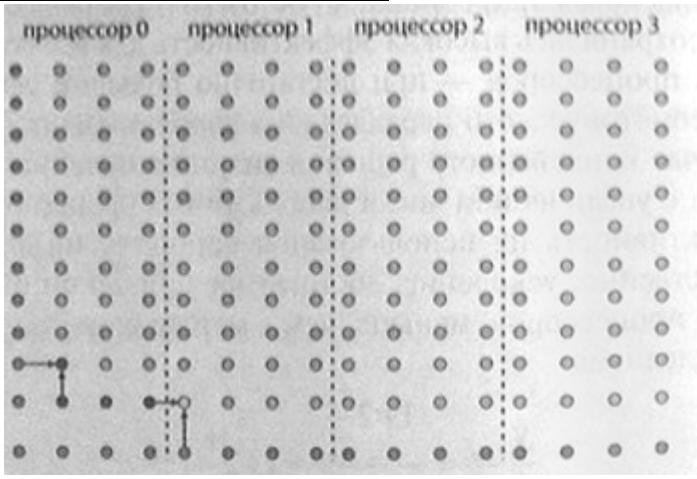
необходимости подтверждения того, что данный шаг на предыдущем слое уже завершен.

В параллельной версии алгоритма центральный цикл содержит по две операции передачи данных при обработке каждого из узлов (прием значения с предыдущего слоя и отправка вычисленного значения на следующий). Следовательно, основные характеристики можно оценить следующим образом:

$$T_p = \tau_A \cdot \frac{kn}{p} + 2\tau_s \cdot \frac{kn}{p},\ S_p = p\frac{1}{1+2\frac{\tau_s}{\tau_c}},\ E_p = \frac{1}{1+2\frac{\tau_s}{\tau_c}}.$$

Основная причина низкой эффективности использования процессоров – высокие накладные расходы на передачу данных – информация о каждом из узлов обрабатываемой сетки мигрирует с процессора на процессор.

Рассмотри другую стратегию распределения данных при решении с помощью метода конвейерного параллелизма. Распределим их так же, как и в методе геометрического параллелизма.




Вначале вычисления выполняются только на процессоре 0, остальные простаивают. Как только он обработает $n/p$ точек (с номерами $[0;\frac{n}{p}-1]$), информация о точке $\frac{n}{p}-1$ передается процессору 1. Теперь работу продолжают два процессора. Процессор 0 вычисляет значения второго слоя в узлах $[0;\frac{n}{p}-1]$, а процессор 1 – значения первого слоя в узлах $[\frac{n}{p};\frac{2n}{p}-1]$. В результате на каждом слое каждый процессор только один раз принимает данные и только один раз передает их. Оценка характеристик:

$$T_p = \tau_A \cdot \frac{kn}{p} + 2k\tau_s,\ S_p = p\frac{1}{1+2\frac{p}{n}\frac{\tau_s}{\tau_c}},\ E_p = \frac{1}{1+2\frac{p}{n}\frac{\tau_s}{\tau_c}}.$$

Следует заметить, что данная оценка не учитывает влияние начального и конечных простоев и справедлива только при $p \ll k$.

Полная оценка, учитывающая все простои, выводится на 106–106 книги Якобовского.

21.Метод диффузной балансировки загрузки.

Метод диффузной балансировки нагрузки - динамический.

Рассмотрим причины дисбаланса:
1)Не всегда можно априори указать трудоемкость обработки узлов сетки.
2)Процессоры могут обладать разный, неизвестной заранее производительностью.
3)Трудоемкость обработки каждого из узлов расчетной сетки может отличаться на разных моментах модельного времени.
4)Эффективная производительность может меняться с течением времени.

Рассмотрим вариант, когда дисбаланс вызван пунктами 2, 4.**Предполагается одинаковая трудоемкость обработки любого узла сетки.**
Пусть в сетке n узлов, и на шаге j процессор i обработал$n_i$ точек за время$t_i$. $t_i$ не включает в себя затраты на ожидание других процессов.
Тогда, в соответствии с основной идеей метода диффузной балансировки загрузки, можно определить новое распределение $n_i'$ узлов сетки по процессорам, например, следующим образом:

$$n_i' = N \frac{\frac{n_i}{t_i}}{\sum_{i=0}^{p-1} \frac{n_j}{t_j}}$$

Этот подход обеспечивает равномерное распределение нагрузки только в том случае, если трудоемкость обработки всех узлов сетки на каждом из шагов по времени равны между собой. пропорционально производительности процессоров.

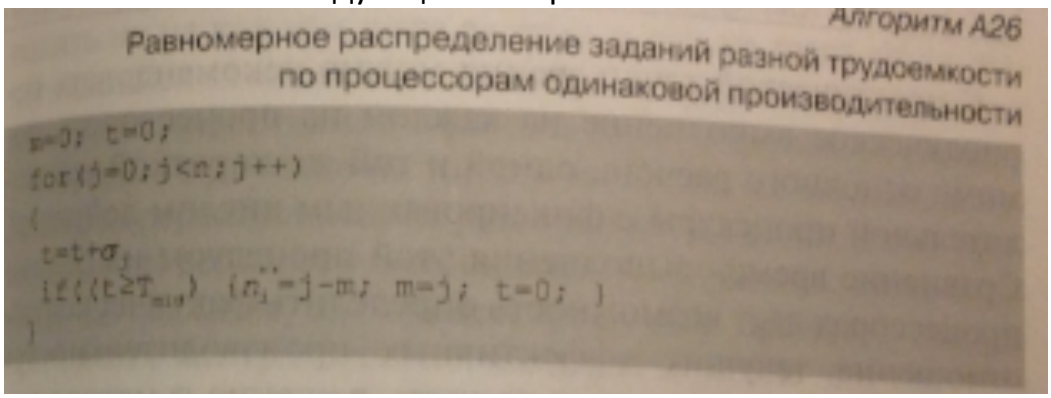
Рассмотрим вариант, когда дисбаланс вызван пунктами 1,3, а процессоры имеют одинаковую производительность.
Тогда подход к перераспределению должен быть другим. Общая вычислительная нагрузка $T = \sum_{k=1}^{p} t_i$ . для достижения идеальной балансировки на каждый процессор нужно назначит столько вершин, сколько на нем может быть обработано за время $T_{mid} = T/p$
Предположим, что на сетке единая нумерация и каждый процесс обрабатывает непрерывный диапазон узлов + затраты на все узлы в интервале у каждого процессора одинаковы.
Пусть $\tau_i = t_i/n_i$ - трудоемкость расчета каждой из точек, обработанных на процессоре i. Тогда $\sigma_j = \tau_{proc(j)}$ время обработки узла j, где proc(j) - номер процесса, который обрабатывал j.

Целью балансировки нагрузки в данных условиях является минимизация $\theta = min\ max \sum_{j \in proc_i} \sigma_j$

Суммирование тут выполняется по всем узлам сетки, назначенным на процессор i согласно новому распределению.

Предполагается, что изменение трудоемкости обработки узлов сетки происходит медленно.

Для приближенного определения нового распределения $n_i''$ можно воспользоваться следующим алгоритом.



Алгоритм А26

Равномерное распределение заданий разной трудоемкости по процессорам одинаковой производительности

```
s=0; t=0;
for(j=0;j<n;j++)
{
 t=t+σ_j
 if((t≥T_min) {n_i''=j-m; m=j; t=0; }
}
```

# 22. Метод серверного параллелизма.

***Постановка задачи динамической балансировки:***

Требуется разработать алгоритм, обеспечивающий обработку за наименьшее время совокупности заданий, при следующих условиях:

- в начале вычислений блока на каждом из процессоров хранится некоторое количество заданий $n_i$;
- время, требуемое для выполнения каждого из заданий (в нашем случае - решения одной системы ОДУ, относящейся к одному узлу расчетной сетки) $t^j_i$, j=0, …., $n_i$-1, заранее не известно;
- по завершении вычислений результаты расчета каждого из заданий должны быть размещены на том процессоре, на котором задание располагалось в начале вычислений.

**=>** очевидно, что минимальное время достигается при равномерном распределении всей вычислительной нагрузки по доступным процессорам.

**Метод серверного параллелизма:**

На каждом процессоре одновременно выполняется ***управляющий процесс*** и ***обрабатывающий процесс***. Обрабатывающий процесс занимается вычислением точек, а управляющий перераспределением “горячих” точек от процессоров, на которых их избыток к тем, на которых их нет.

Алгоритм работы обрабатывающего процесса:

- Ожидание сообщения от управляющего процесса своего процессора
- Если пришло сообщение с указание на завершение работы, то завершить работу
- Если пришло сообщение, содержащее данные, то обработать данные и вернуть результат

Алгоритм работы управляющего процесса:

1. **Если** есть необработанные точки (локальные или внешние) **и** процесс свободен, **то** установить флаг обрабатываемой точки **и** передать одну из необработанных точек обрабатывающему процессу
2. **Если** нет локальных необработанных точек **и** нет внешних точек **и** нет обрабатываемых точек **и** флаг запроса на получение необрабатываемых точек не установлен **и** есть процессоры, которые еще не ответили что не могут предоставить точки для обработки (соответствующий флаг запрета обменов не установлен), **то** послать запрос на получение необработанных точек одному из таких процессоров **и** установить флаг запроса на получение необработанных точек
3. **Иначе если** все переданные точки получены обратно необработанными **и** от всех процессоров получено сообщение о

том,что точки для обработки предоставлены быть не могут **и** всем процессорам послано сообщение о том, что точки для обработки предоставлены быть не могут, **то** завершение работы
4. Получить любое сообщение от любого процессора или от своего обрабатывающего процесса
5. Обработать полученное сообщение
6. Перейти к п.1

**Замечания**: нужно гарантировать отсутствие никем не принятых сообщений, поэтому на каждое сообщение следует слать ответ.

## 23. Отладка параллельных приложений, выполнение которых сопровождается недетерминированным потоком сообщений.

Алгоритмические ошибки можно разделить на

- сильные – в этом случае программа проходит логическую последовательность состояний, которая приводит к результату, отличному от ожидаемого, или к тому, что программа неспособна выполнять свои функции;
- слабые, которые не являются как таковыми ошибками в смысле данного выше определения, а представляют собой причины неэффективного поведения программы пользователя (например, неполное использование вычислительных ресурсов).

Среди сильных алгоритмических ошибок выделяют:

- локальные – для их обнаружения каждому процессу не требуется информация от других процессов;
- глобальные, которые включают 2 и более процессов.

Глобальные ошибки, возникающие при обмене данных между ветвями параллельного приложения, можно разбить по следующим категориям.

1)Ошибки синхронизации:
а) дедлоки
- потенциальные;
- реальные;

б) гонки данных.
2) Ошибки несоответствия.

Дедлоки разделяют на реальные, которые случаются всегда, и потенциальные, которые не проявляются на данной архитектуре или реализации MPI, но могут возникнуть при переносе приложения на другую платформу.

Гонки данных — ошибка проектирования многопоточной системы или приложения, при которой работа системы или приложения зависит от того, в каком порядке выполняются части кода

В MPI Standard определено, что аргументы, переданные в коммуникационную операцию каждым из процессов, должны

соответствовать аргументам, переданным другими процессами. В простейшем случае под «соответствующими» аргументами имеют в виду идентичные, но возможны и более сложные случаи. Так, абсолютно корректно послать два элемента составного типа (MPI_INT, MPI_DOUBLE) и получить один (MPI_INT, MPI_DOUBLE, MPI_INT, MPI_DOUBLE). MPI-реализации обычно прерывают приложение, когда есть несоответствие типов данных, но никакой точной информации о несоответствии пользователю не выдается.

**Методы и программные средства обнаружения алгоритмических ошибок**

1) **Диалоговая отладка**. Параллельные отладчики обеспечивают обычные функциональные возможности отладчиков, типа выполнения в пошаговом режиме, установки контрольных точек, оценки переменных, и т. д.

2) **Верификация модели программы.** Суть этого метода заключается в следующем. Для заданной анализируемой программы строится ее абстрактная формальная модель. Затем производится спецификация свойств, которыми должна обладать система. Проверяемые свойства или требования выражаются на формальном математическом языке. После этого верификация программы сводится к проверке выполнимости формализованного требования (спецификации) на абстрактной модели. При этом ведется перебор возможных состояний по разным маршрутам в графе.

3) **Сравнительная отладка.** Основная идея данного подхода заключается в том, чтобы сравнить поведение отлаживаемой версии программы с поведением эталонной и выдать пользователю информацию о расхождениях. Часто за эталонную версию принимают последовательный вариант параллельного приложения. Для этого выбираются определенные точки программы и в них сравниваются значения переменных. Выбор точек может осуществляться как пользователем, так и самой программной системой поддержки параллельной отладки. Для сравнения двух выполнений программы можно сначала накопить трассы, фиксирующие выполненные операторы и значения переменных, а затем сравнивать эти трассы, либо сначала собрать трассу при одном выполнении программы, а затем динамически сравнивать с ней другое выполнение.

4) **Автоматический анализ корректности.** Данный метод подразумевает собой выявление поведения параллельной программы, приводящего или способного привести к логическим ошибкам. Чаще всего анализируются аргументы и последовательность MPI-вызовов, сравни- ваются с информацией о вызовах в других процессах, а затем на основании некоторых

алго- ритмов делается вывод о существовании ошибок. Для сбора информации об MPI-вызовах применяется профилировочный интерфейс MPI.

5) **Анализ по трассе.** Автоматический анализ корректности разделяют на анализ по трассе (посмертный анализ) и динамический анализ. При применении посмертного анализа каждый процесс параллельного приложения записывает данные об MPI-вызовах в локальный файл трассы, после работы файлы с каждого процесса собираются в одну трассу, которая на сле- дующем шаге анализируется отдельной утилитой.

6) **Анализ во время исполнения.** Отличие данного подхода от предыдущего заключается в том, что анализ вызовов MPI-функций про- изводится не по собранной трассе после завершения параллельной программы, а динамиче- ски по ходу работы.

**24.Параллельные алгоритмы сортировки данных. Слияние методом сдваивания и двустороннее слияние. Масштабируемость алгоритмов сортировки. Сети сортировки.**

Задача - расположить в порядке неубывания N элементов массива числе, используя p процессов.

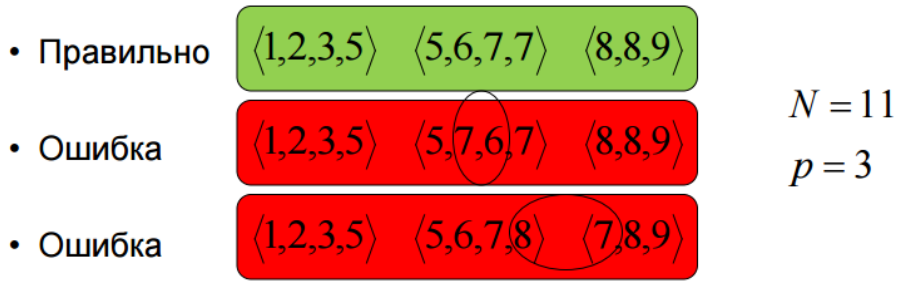
Две задачи сортировки массива чисел:
A. Объем оперативной памяти одного процессорного узла достаточен для одновременного размещения в ней всех элементов массива.
B. Объем оперативной памяти одного процессорного узла мал для одновременного размещения в ней всех элементов массива.

Для A:

$$a_i <= a_{i+1}$$

Для B:

1. Части массива хранятся на нескольких процессорах
2. На процессорах с бОльшими номерами должны быть элементы массива с бОльшими значениями




Наилучший последовательный алгоритм, который рассматривался dhsort:
1.Короткие фрагменты сортируем с помощью hsort(пирамидальная сортировка)
2.Затем слияние алгоритмом dsort этих упорядоченных фрагментов.

Hsort плох на больших объемах, тк не последовательный доступ и кэш промахи = доступ в медленную оперативную память.
Dsort - алгоритм сортировки слиянием:

Рекурсивный алгоритм:

```
сортировать ( массив mas, число элементов n )
  {
  если (n > 1)
    {
    // сортировка первой половины массива
    сортировать ( mas, n/2);
    // сортировка второй половины массива
    сортировать ( mas+n/2, n-n/2);
    // слияние отсортированных половинок массива
    слияние ( mas, n/2, mas+n/2,n-n/2);
    }
  }
```

Нерекурсивный алгоритм:

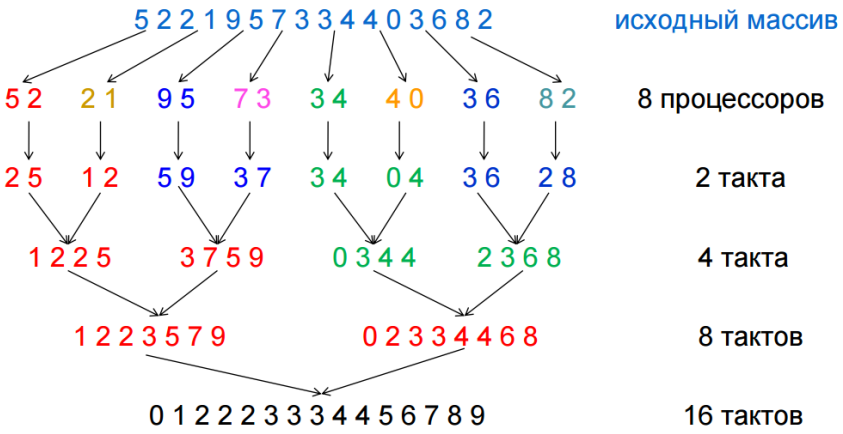
```
Dsort(intsort *array, int n)
{
 a=array;     // сортируемый массив
 b=array_second;     // вспомогательный массив

 for(i=1;i<n;i=i*2) // размер объединяемых фрагментов
  {
  for(j=0;j<n;j=j+2*i) // начало первого из объединяемых
                       // фрагментов
      {
      r=j+i; // начало второго из объединяемых фрагментов
      n1=max(min(i,n-j), 0);
      n2=max(min(i,n-r), 0);

      // слияние упорядоченных фрагментов
            b = a[r…r+n1] & a[j…j+n2]
      }
  c=a;a=b;b=c;
  }
```

Слияние можно делать двумя способами:

1)Стандартно одним процессом, когда 1 процесс проходит от меньших элементов к большим и склеивает 2 упорядоченных массива по k элем. за 2k тактов:

2)Двумя процессами, один проходит от меньших к большим элементам, другой наоборот. Это называется **Алгоритм двустороннего слияния** . Тактов в 2 раза меньше.

**Слияние методом сдваивания**: Нерекурсивный алгоритм dsort допускает параллельную обработку на основе метода сдваивания.

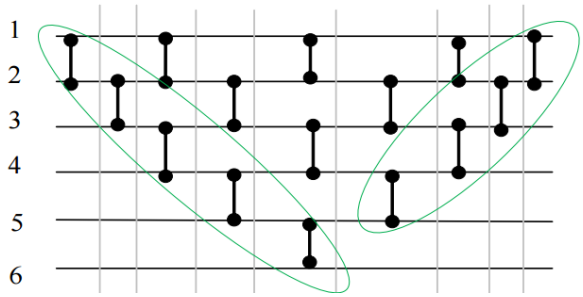



! Слияние методом сдваивания и двустороннее слияние являются двумя параллельными алгоритмами сортировки, которые эффективно используют большое количество процессоров, **но не обеспечивают возможность обработки данных, размер которых превышает объем оперативной памяти одного узла.**

Поэтому рассмотрим подход, основанный на понятии “Сеть сортировки”

***Сеть сортировки*** - это вид алгоритмов сортировки, в которых порядок выполнения операций сравнения и их количество не зависит от значения элементов сортируемого массива. Каждый элемент массива последовательно обрабатывается ***компараторами сравнения\перестановки***. Принято изображать сортируемые элементы массива горизонтальными линиями данных, а компараторы - вертикальными отрезками. Каждый компаратор соединяет две линии данных. Таким образом у него 2 входа и 2 выхода. Компаратор принимает на вход 2 элемента массива. В компараторе элементы сравниваются между собой и, при необходимости переставляются местами таким образом, чтобы на верхнем выходе всегда был меньший из двух, а на нижнем больший.

Сеть сортировки (пузырёк) *n=6 s=2n-3=9*

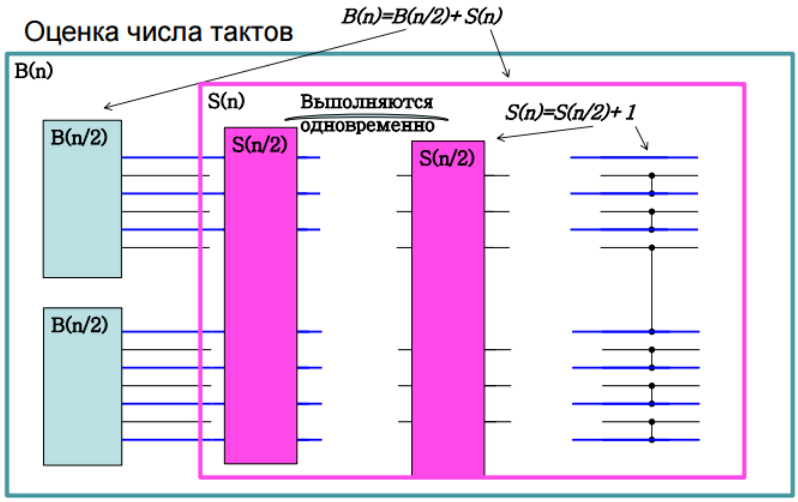



$$O\left(\frac{n}{p}\left[\log_2\frac{n}{p}+p\right]\right)=O\left(\frac{n}{p}\log_2\frac{n}{p}+n\right)\approx O(n)$$

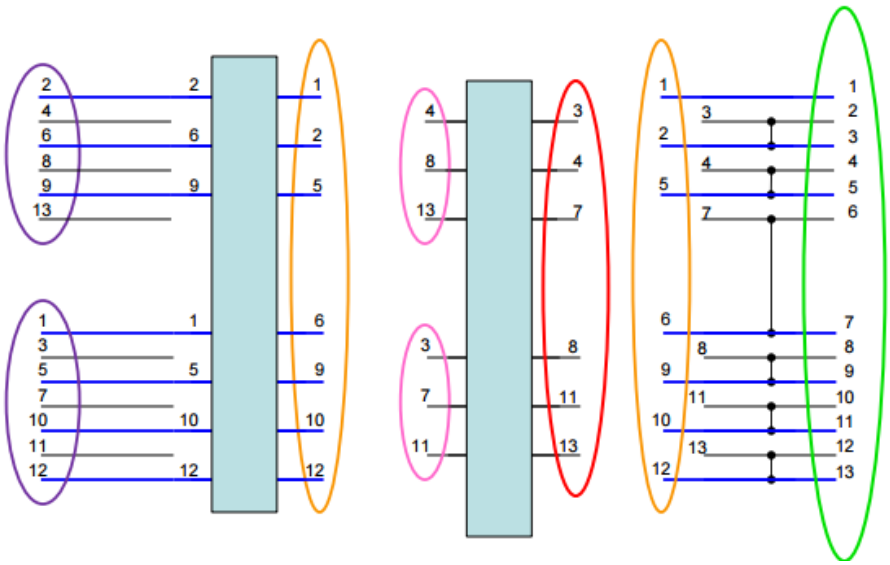
**Сеть обменной сортировки со слиянием Бэтчера.**

Наиболее быстрая из рассматриваемых масштабируемых сетей сортировки. Для сортировки массива из n элементов следует разделить его на 2 части, отсортировать каждую из частей и объединить результаты с помощью (n,m) сети нечетно-четного слияния Бэтчера. В сети четно-нечетного слияния отдельно объединяются элементы массивов с нечетными номерами и отдельно с четными, после чего, с помощью заключительной группы компараторов, обрабатываются пары соседних элементов с номерами (2i, 2i+1), где i от 1 до n/2-1.

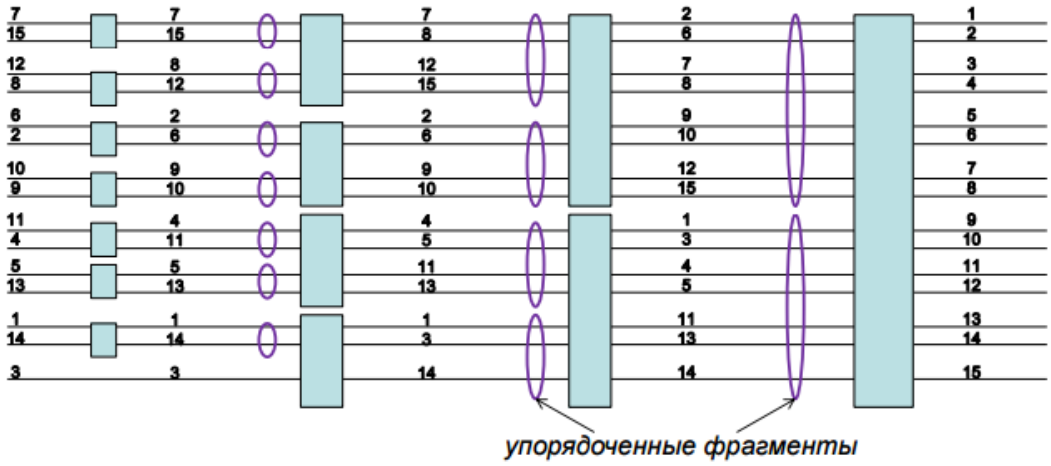
Общая структура алгоритма:




Чётно-нечётное слияние Бэтчера:



Сортировка массива из 15 элементов:




 - сеть четно-нечетного слияния Бетчера

Оценка числа шагов:

Пусть Qb(p) - число шагов работы сети сортировки.

Qs(p) - число шагов работы сети слияния упорядоченных групп нечетных и четных линий в каждой из которых p/2 линий.

$Qb(p) = Qb(p/2) + Qs(p)$

$Qs(p) = Qs(p/2) + 1$

$Qb(p) = \ln(p) * (\ln(p) + 1) / 2.$

Оценка числа операций для рассмотренных методов:

Зависимость количества операций от размера массива и числа процессоров

| Алгоритм | Число шагов сети сортировки $Q(p)$ | Объем памяти процессора | Оценка числа операций |
|---|---|---|---|
| Параллельная сортировка сдваиванием | — | $2n$ | $O\left(\frac{n}{p}\log_2\left(\frac{n}{p}\right)+n\right)$ |
| Сеть простой вставки | $2p-3$ | $3\frac{n}{p}$ | $O\left(\frac{n}{p}\log_2\left(\frac{n}{p}\right)+n\right)$ |
| Сеть четно-нечетной перестановки | $p$ | $3\frac{n}{p}$ | $O\left(\frac{n}{p}\log_2\left(\frac{n}{p}\right)+n\right)$ |
| Сеть обменной сортировки со слиянием Бэтчера | не больше $\frac{\lceil\log_2 2p\rceil\lceil\log_2 p\rceil}{2}$ | $3\frac{n}{p}$ | $O\left(\frac{n}{p}\left[\log_2\frac{n}{p}+\frac{\lceil\log_2 p\rceil^2}{2}\right]\right)$ |

25. Параллельные алгоритмы генерации псевдослучайных чисел. Требования к генераторам псевдослучайных чисел. Линейно-конгруэнтные генераторы. М-последовательности.

Есть два вида ГПЧ (Генератора псевдослучайных чисел):

1. Основанный на некоторых физических принципах
   - Формируют произвольно длинные последовательности случайных чисел (+++)
   - Различные последовательности на различных процессорах (+)
   - Не обеспечивают воспроизводимость результатов (+/-, в зависимости от задачи, тяжело отлаживать, плюс для криптографии)
   - Зависят от внешних условий (-)
   - Не поддаются априорному (предварительному) тестированию (-)
   - Аппаратно зависимые (--, сложная переносимость)
2. Основанный на детерменированных алгоритмах
   - Ограничены периодом (-, т.к. он может быть очень большим)
   - Различные последовательности на различных процессорах (+)
   - Есть алгоритмы генерации единой последовательности на различных процессорах (++)
   - Могут быть предварительно протестированы (+)

1 — используют когда воспроизводимость результатов не требуется, и качество и количество физических ГПЧ достаточно для программы

Программные ГПЧ разделают на два типа:

1. Своя последовательность на каждый процесс (+)
2. Каждый i+k*p mod T берется i-м (из p процессоров, T - период последовательности) (leapfrog method) (--, разные результаты на разном числе процессоров, подпоследовательность может быть даже линейной функцией)
3. Элементы от i*N до (i+1)*N-1 элементов последовательности (N - какое-то большое число), (skip-ahead method) (++, считается самым предпочтительным)

**Требования для ГПЧ:**

1. Возможность формирования последовательности достаточно большой длины
2. Определение любого элемента последовательности быстро, без вычисления предыдущих

**Линейно-конгруэнтные генераторы**

$u_{n+1} = (a*u_n+c) \bmod m$ — линейно-конгруэнтный генератор

Может вычислять элементы по сокращенным формулам:

$u_n = (a^n * u_0 + c*(a^n-1)/(a-1)) \bmod m$ — за o(logn) (через известную формулу $a^{2t}=a^t(a^t-1) + a^t$
$u_{k(n-1)} = (a^k * u_{kn} + c*(a^k-1)/(a-1)) \bmod m$

Минусы:

- малая длина последовательности
- k-мерные точки лежат не более чем в $(k! * m)^{1/4}$ плоскостях размерности (k-1)

Длина последовательности меньше m, и равна m ⇔

1. c и m взаимно простые
2. a-1 кратно p для любых простых p являющихся делителем m
3. a-1 кратно 4, если m кратно 4

**M-последовательности**

$U_n = a_{n-1}U_{n-1} + a_{n-2}U_{n-2} + \ldots + a_{n-r}U_{n-r} \bmod 2$
$U_0=1, U_1,U_2,\ldots U_{r-1} = 0$
Например:
$U_n = U_{n-20}+U_{n-33}$ — генерирует однобитовую псевдослучайную последовательность с периодом $2^{33}-1$
Если предварительно вычислить матрицу смещений (требует $r^2$ памяти и вычисление элемента за $O(r^2 \log n)$ операций, однако, он применим для общих ГПЧ, но сузив класс рассматриваемых генераторов можно оптимизировать эти значения до $O(r \log r \log n)$ и затраты по данным до $O(r \log r)$ (через эквивалентность такой последовательности и последовательности, генерируемой $t_i = x^i \bmod G(x)$ )

n-разрядные числа вычисляются по формуле:
$A = SUM(i=0, i<n; i++)\{2^i * t_i \bmod G(x)\}$
$t_{i+1} = xt_i \bmod G(x)$
$t_0= 1$ (обычно)

**26. Декомпозиция расчетных сеток. Критерии декомпозиции.**

Расчетная сетка - совокупность точек (сеточныйх узлов), заданных в области определения некоторой функции.

Критерии декомпозиции:

1. Классический критерий
- равномерно распределение по доменам суммарного веса узлов(ребер)
- минимизация максимального веса исходящих из домена ребер

Пусть задан граф $G^0 = (V, E)$, $V\{v_i\}$, $|V| = n$. Пусть вершины $v_i$ и ребра $e_{ij}$ имеют веса $w(v_i)$ и $w(v_i, v_j)$ соответственно. Тогда требуется найти такое разбиение $R(V) = \{V1, \dots, Vp\}$ вершин на заданное число доменов p, при котором J — взвешенная сумма весов вершин и разрезанных ребер — принимает минимальное значение:

*формула со страницы 183*

1. Выделение обособленных доменов
- исключение связей между доменами
- минимизация суммарного веса разрезанных ребер

1. Минимизация максимальной степени домена
- Степень домена — число граничащих с ним доменов.
- Ведет к снижению актов обмена данными между процессорами на каждом шаге работы

1. Обеспечение связности графов каждого из доменов
- При других критериях в рамках одного домена часто оказываются несколько несвязных компонент. Это увеличивает степень домена и снижает удобство работы.

**27. Иерархические алгоритмы разбиения графов. Локальное уточнение.**

Иерархический подход предполагает огрубление сетки (построение приближенного образа сетки, содержащего меньше узлов), декомпозицию ограбленного образа и последующее уточнение разбиения на основе локальных критериев.

1. Огрубление графа

Алгоритм основан на идее итерационного стягивания ребер. На каждом шаге из всего множества ребер выбирается подмножество ребер покрытия так, чтобы никакая из вершин сетки не была инцидента более чем одному из выбранных ребер. Далее каждая пара вершин, соединяемая ребром покрытия, стягивается в одну новую вершину.

1. Декомпозиция
    1. Разбиение сразу на требуемое число доменов
    2. Рекурсивно — разбиение на каждом шаге рекурсии на небольшое число фрагментов (обычно 2, 4 или 8)

Самый привлекательный метод — спектральный, но он вычислительно затратный. Чаще всего используется метод координатной бисекции и метод заполняющих кривых.

1. Восстановление графа

Порождающие вершины (огрубления) распределяются в те же домены, что и образованные ими вершины.

Полученное начальное разбиение можно существенно улучшить перераспределением вершин между соседними доменами с помощью локального уточнения.

Локальное уточнение границ доменов.
— Позволяет выровнять веса доменов между собой и уменьшить число ребер, пересекающих границы доменов.

Обычно используется алгоритм Кернигана-Лина (KL-алгоритм).

- эвристически -> нет полного перебора
- проверяет выигрыш от перемещения вершины в соседние домены

Итеративно повторяются следующие шаги:

A. Для каждой вершины определяется выигрыш от перемещения в соседние домены.

B. Определяется множество пар (вершина, в какой домен перемещаем), для которых максимальный выигрыш.

C. Переносим вершину в множество перемещаемых, если выигрыш положительный.

D. Перемещение в новые домены вершин из множества перемещаемых.
E. Если величина J уменьшилась, то принимаем новое разбиение.
F. Если J не уменьшилась, то отменяем изменения и выбираем другое множество перемещаемых вершин.
G. Если несколько попыток выбрать множество перемещаемых вершин закончились неуспехом то разбиение считает оптимальным и процесс заканчивается.
H.

29. Адаптивное интегрирование. Метод локального стека. Метод глобального стека.

Вычислить с точностью $\varepsilon$ значение определенного интеграла: $J(A,B)=\int_A^B f(x)dx$

Пусть на отрезке [A,B] задана равномерная сетка, содержащая n+1 узел:

$$x_i = A + \frac{B-A}{n} i, \quad i = 0,\cdots,n$$

Тогда, согласно методу трапеций, можно численно найти определенный интеграл от функции на отрезке [A,B]:

$$J_n(A,B) = \frac{B-A}{n}\left(\frac{f(x_0)}{2} + \sum_{i=1}^{n-1} f(x_i) + \frac{f(x_n)}{2}\right)$$

Будем полагать значение J найденным с точностью $\varepsilon$, если выполнено условие:

$$\left|J_{n1} - J_{n2}\right| \le \varepsilon \left|J_{n2}\right|$$

$$n_2 > n_1$$

В качестве последовательных алгоритмов используются:

- Метод трапеций

  Неэффективен так как:

  - в некоторых точках значение подынтегральной функции вычисляется более одного раза

- на всем интервале интегрирования используется равномерная сетка, тогда как число узлов сетки на единицу длины на разных участках зависит от вида функции f(x).

- Метод рекурсивного деления -

разбиваем интервал интегрирования на 2 части и независимо проинтегрируем f(x) на каждой их них. Процедуру разбиения можно рекурсивно повторять до получения отрезков, на которых подынтегральная функция может быть с заданной точностью аппроксимирована отрезком. Таким образом выигрыш достигается за счет возможности использования сеток с разным числом узлов на разных участках интервала интегрирования. Поскольку в реальной системе число процессоров ограничено, и время, необходимое для порождения параллельного процесса ,значительно больше 0, такой подход неэффективен.

- Метод локального стека.

Метод позволяет распределить вычислительную работы между ограниченным числом процессов. Несколько ветвей программы одновременно обрабатывают разные части интервала интегрирования, а координаты концов этих интервалов хранятся в некоторого явно описанном массиве(организованный по принципу стека) и передаются по частям для обработки разным нитям программы.

Времена выполнения метода рекурсивного деления и метода локального стека отличаются примерно на 5% в пользу последнего, таким образом, алгоритм локального стека будем считать в качестве наилучшего.

Параллельные алгоритмы:

- метод геометрического параллелизма (статическая балансировка)

Отрезок интегрирования разбивается на p частей, p - число процессоров. Отрезки распределяеются по процессорам, каждый процессор вычисляет частисчную сумму на своем орезке, после чего все суммы складываются. Метод эффективен при условии равномерного распределения всего объема вычислений по отрезку интегрирования. Однако существует множество функций, при интегрировании которых указанное условие не соблюдается. На одном из отрезков может быть сосредоточенно основное вычисление и метод геометрического парллелизма становится не пригодным.

```
main() {
...
for(i=0;i<p;i++)
{StartParallelProcess(IntTrap04,A+(B-A)*i/p, A+(B-A)*(i+1)/p, &(s[i]) );}
```

```
WaitAllParallelProcess();
J=0
for(i=0;i<p;i++)
  J+=s[i]
}
```

- метод коллективного решения (динамическая балансировка)

```
main(){
// Порождение p параллельных процессов,
// каждый из которых выполняет процедуру slave
for(k=0;k<p;k++)
  StartParallel(slave #k)
i=0 // число переданных для обработки интервалов // n – число отрезков интегрирования

for(k=0;k<p;k++)
{ // Передача концов отрезков интегрирования
      Send(slave k, A+(B-A)*i/n, A+(B-A)*(i+1)/n) i++
}

// J - значение интеграла на всем интервале [A,B]
J=0
Пока есть отрезки, не переданные для отработки,
следует дождаться сообщения от любого из процессов slave,
вычислившего частичную сумму на переданном ему отрезке,
Получить значение этой суммы, прибавить к общему значению
Интеграла и передать освободившемуся процессу очередной
отрезок
while(i<n) {
      k = Recv(slave ANY, s)
      J+=s
      Send(slave k, A+(B-A)*i/n, A+(B-A)*(i+1)/n) i++
      }
// Получить результаты вычислений переданных отрезков
// и прибавить их к общей сумме

for(k=0;k<p;k++)
{
      Recv(slave any, s)
      J+=s
}}
//подчиненный процесс, вычисляющий значение интеграла на отрезке [a,b]
```

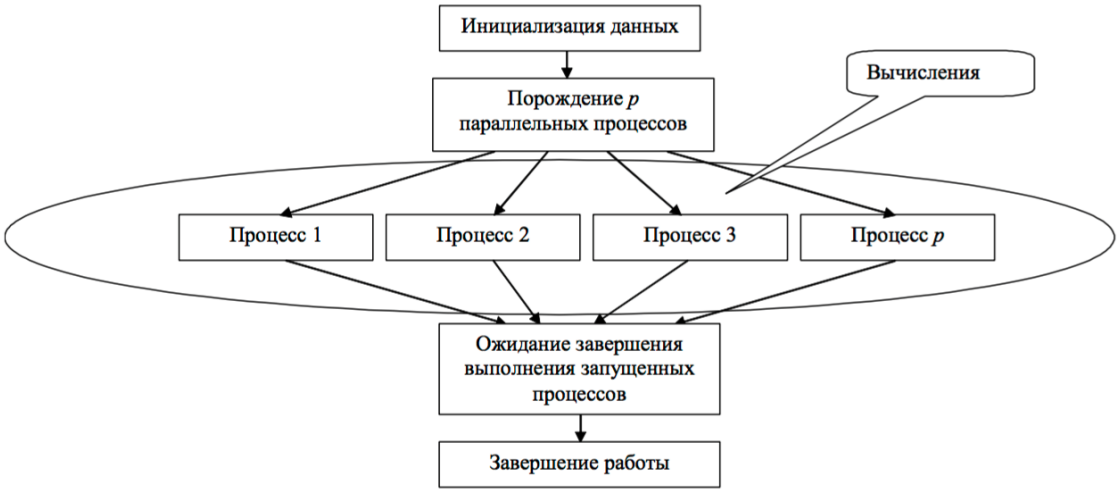
```
slave(){
        while(1){
                Recv(main,a,b)
            s=IntTrap04(a,b)
               Send(main,s)
             }
}
```

- Либо большой дисбаланс загрузки процессоров
- Либо большой объем лишних вычислений

Оба метода практически непригодны для решения поставленной задачи.

- метод глобального стека

Рассматривается система с общей памятью, используются треды (нити), для защиты распределяемых переменных - семафоры. Не требуется налачие управляющего процесса. Принципиальное отличие от коллективного решения : порожденные процессы совершенно равноправны в своих возможностях, и каждый из них в равной степени выполняет вычислительные и управляющие функции.




Пусть есть доступный всем параллельным процессам список отрезков интегрирования, организованный в виде стека. Назовем его глобальным стеком.

Пусть у каждого процесса есть свой, доступный только этому процессу локальный стек

Перед запуском параллельных процессов в глобальный стек помещается единственная запись (в дальнейшем "отрезок"):

- координаты концов отрезка интегрирования,
- значения функции на концах,

– приближенное значение интеграла на этом отрезке.

Тогда, предлагается следующая схема алгоритма, выполняемого каждым из параллельных процессов:
Пока в глобальном стеке есть отрезки:
1) взять один отрезок из глобального стека - выполнить алгоритм локального стека, но, если при записи в локальный стек в нем есть несколько отрезков, а в глобальном стеке отрезки отсутствуют, то переместить часть отрезков из локального стека в глобальный стек.
2) по исчерпанию локального стека добавить полученную частичную сумму к общему значению интеграла и вернуть к шагу 1

Отличия от метода локального стека:

1) в конце цикла интегрирования одного интервала присутствует фрагмент переноса отрезком из локального в глобльный стек
2) по окончинии цикла интегрирования выполняется суммирование частичных сумм, полученных всеми процессами

30. Визуализация сеток и сеточных данных.

Три основных этапа визуализации:

- передача данных системе визуализации
- преобразование их к виду, удобному для восприятия
- отображение данных на экране

Причины, по которым для анализа больших данных необходимы параллельные вычислительные системы и решения, ориентированные на удаленную визуализацию:

1. ограниченная пропускающая способность каналов связи между суперкомпьютером и рабочим местом пользователя не обеспечивает возможности быстрой передачи больших объемов данных
2. ограниченный объем оперативной памяти компьютера пользователя затрудняет обработку больших объемов данных, существенно снижая эффективность
3. ограниченный размер файловой системы компьютера пользователя не позволяет в полном объеме хранить результаты вычислений и не обеспечивает приемлемую скорость доступа к хранимым данным

Клиент-серверная технология

- серверная часть системы визуализации обеспечивает обработку больших объемов данных
- клиентская часть системы визуализации обеспечивает интерфейс для пользователя

Online-визуализация -- визуализация данных непосредственно при проведении расчета, используя данные в оперативной памяти.

+ экономия дискового пространства и времени на осуществление операций чтения и записи
+ потенциальная возможность активного влияния на ход вычислений (можно реализовать и при offline-визуализации)
- большие требования к оперативной памяти
- невозможность повторного изучения единожды визуализированных данных
- ограничены возможности визуализации динамики процесса
- ограничены возможности визуализации заданных моментов времени
- неконтролируемо ухудшается балансировка загрузки процессоров
- организационные трудности из-за коллективного доступа

Offline-визуализация -- характеризуется периодической записью данных на диск и их визуализацией с помощью отдельной программы.

Единственный основной минус -- необходимость записи на диск промежуточных данных, что ведет к  загруженности дискового пространства и отнимает время.

Этапы визуализации:

| Моделирование | Сервер визуализации | Клиент визуализации |
|---|---|---|
| **а)** декомпозиция сетки<br>**б)** запись структуры сетки<br>**в)** запись фрагментов сетки<br>**г)** запись фрагментов сеточной функции | **д)** чтение структуры сетки<br>**е)** декомпозиция сетки, назначение фрагментов сетки на процессоры<br>**ж)** чтение фрагментов сетки<br>**з)** чтение фрагментов сеточной функции<br>**и)** формирование данных, описывающих виртуальную сцену и их передача на клиент визуализации | **к)** прием данных от сервера визуализации и преобразование их к виду, пригодному для отображения<br>**л)** отображение<br>**м)** манипулирование образом объекта |

## 31 Стратегии ввода-вывода больших объёмов данных, предназначенных для визуализации

В обработке и визуализации больших сеток значительную часть времени сжирает процессы чтения и записи данных. Одним из способов уменьшения времени чтения является использование двухуровневой схемы обработки больших сеток (см. 32 билет). Это дает следующие преимущества:

- По сравнению с поэлементной обработкой, существенно повышается скорость чтения.
- Обеспечивается возможность независимого параллельного чтения процессорами микродоменов.
- При решении задач визуализации, появляется дополнительная возможность значительного уменьшения хранимых данных.

*Небольшое замечание. Сеточные данные в основном состоят из двух частей: сама расчетная сетка и сеточные функции. В расчетной сетке заложена топология узлов, поэтому эти данные сжимать с потерями нельзя, а вот сеточные функции - можно. Про это дальше и рассказывается.*

Для уменьшения времени записи данных перед непосредственной записью можно прибегнуть к сжатию данных, но использование стандартных методов сжатия без потерь применительно к вещественным числам неэффективно. Для увеличения степени сжатия данных, предназначенных для анализа методом визуализации, допустимо игнорировать часть младших разрядов мантиссы каждого из значений сеточной функции, что значительно увеличивает коэффициент компрессии. Относительная ошибка огрубления данных, полученная в результате отбрасывания *k* двоичных разрядов мантиссы, приближенно может быть оценена как $2^{k-24}$. Использование иерархической двухуровневой схемы хранения позволяет задавать параметр огрубления независимо для каждого из записываемых блоков, что позволяет записывать значения сеточной функции, соответствующие разным фрагментам сетки, с разной точностью.

Дополнительно увеличить коэффициент компрессии данных можно за счет перегруппировки разрядов вещественных чисел. Идея заключается в том, что соседние в массиве числа с большой вероятностью имеют одинаковый порядок и одинаковый знак. Выделим биты, отвечающие за сохранение порядка и знака каждого числа в один массив, а остальные биты - в другой. Раздельная компрессия получившихся массивов в общем случае обеспечит меньший объем сжатых данных, чем компрессия исходных чисел.

## 32 Двухуровневое хранение сеточных данных

Большой объем данных предполагает использование множества процессоров для фильтрации изучаемых данных, следовательно, необходим метод их рационального распределения по выделенным для обработки процессорам. Несмотря на высокую эффективность иерархических алгоритмов, разбиение нерегулярных расчетных сеток большого размера занимает значительное время. Для уменьшения потерь целесообразно использовать **двухуровневую стратегию хранения и обработки больших сеток.**

В соответствии с ней расчетная сетка предварительно однократно разбивается на множество блоков небольшого размера - микродоменов. В дальнейшем сетка хранится в виде набора микродоменов и графа, определяющего их взаимосвязи - макрографа. Каждая из вершин макрографа соответствует одному микродомену. Вершины каждого из микродоменов и информация об определенных на них элементах сетки сохраняются вместе как единое целое. При каждом сеансе расчета производится разбиение макрографа, и каждому процессору назначается список микродоменов. Поскольку в макрографе значительно меньше вершин, чем в исходной сетке, его разбиение требует меньшего времени и может быть выполнено последовательными алгоритмами. Дополнительный выигрыш времени достигается за счет возможности распределенного хранения больших графов как совокупности микродоменов, что значительно уменьшает накладные расходы на чтение и запись сеток большим числом процессоров.

**33 Измельчение расчетных сеток**

Доступные в настоящее время генераторы не позволяют непосредственно получить трехмерные пирамидальные сетки большого размера. В частности сетка, приведенная на Рис. 11 подготовлена с помощью созданного в Институте вычислительной математики РАН пакета и содержит 70 300 узлов. Данная сетка использовалась для моделирования обтекания сферы набегающим стационарным потоком. Сетка равномерно сгущается к поверхности сферы и к оси симметрии за сферой. На рисунке 11 показаны только треугольники сетки, лежащие на внешних границах расчетной области и непосредственно на сфере. После измельчения и корректировки фрагмента сетки, прилегающего к сфере, размер был увеличен примерно в 8 раз. Дальнейшее увеличение размера сетки возможно с помощью алгоритма измельчения, гарантирующего не более чем ограниченное, не зависящее от степени измельчения, снижение уровня качества сетки.

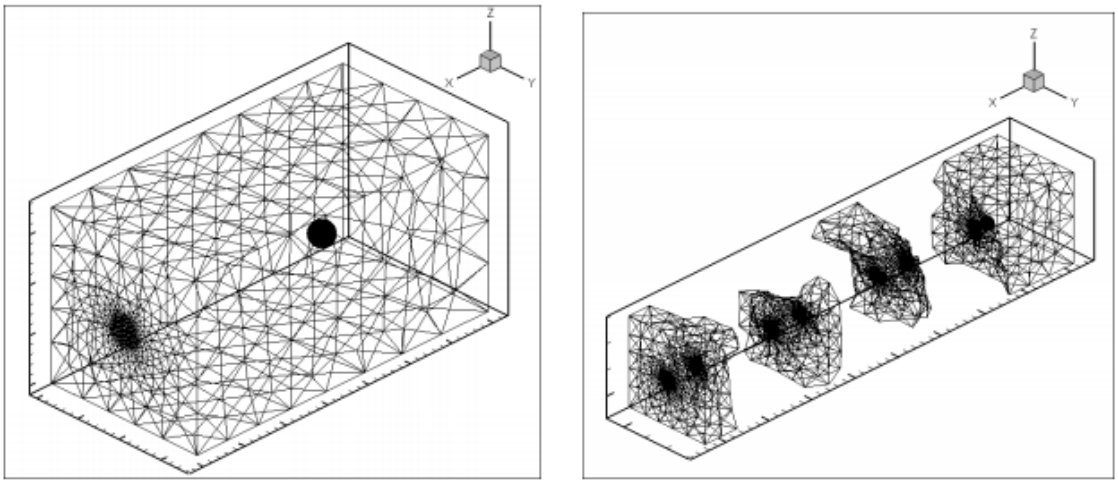


**Рис. 11. Разбиение на 4 части пирамидальной сетки: узлов 70300, тетраэдров 401418, ребер 1336881**

**Рис. 1. Равномерное измельчение двумерной сетки**

Качество сетки определяется геометрическими свойствами образующих сетку примитивов. Количественно его можно определить величиной взятого по всем пирамидам сетки максимума отношения самого длинного ребра к самому короткому в каждой из 4 пирамид. Чем ближе к единице эта величина, тем более «правильные» пирамиды образуют сетку. Равномерное измельчение двумерной сетки не представляет проблемы (Рис. 1), поскольку разбиение каждого треугольника на четыре, подобных исходному

треугольнику, решает задачу увеличения числа узлов, при сохранении геометрических свойств сетки (например, при таком измельчении не меняется отношение длин минимального и максимального ребер в треугольниках сетки).
В трехмерном случае аналогичный алгоритм не известен, тем не менее, равномерное измельчение пирамидальной сетки может быть выполнено следующим образом:

- на первом шаге каждая из пирамид разбивается на четыре подобных исходной пирамиде тетраэдра, и один октаэдр. Вершинами вновь образованных многогранников являются вершины и середины ребер исходной пирамиды (Рис. 2);
- на втором шаге каждая из образованных пирамид измельчается так же, как на шаге 1, а каждый из октаэдров разбивается на шесть октаэдров, подобных исходному октаэдру, и 8 тетраэдров, подобных исходной пирамиде (Рис. 3). Первые два шага повторяются необходимое число раз, до получения сетки заданного размера (точности);
- заключительным является третий шаг, он выполняется только один раз. Каждый из октаэдров разбивается на пирамиды, например на четыре (Рис. 4).

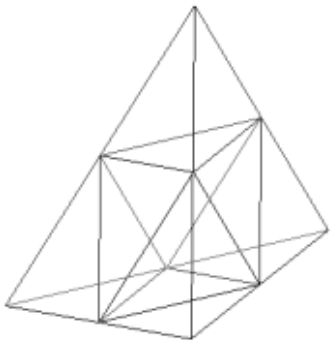


**Рис. 2. Первый шаг**

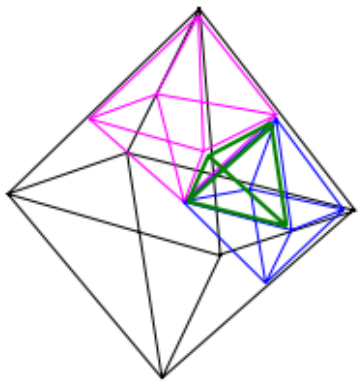


**Рис. 3. Второй шаг**

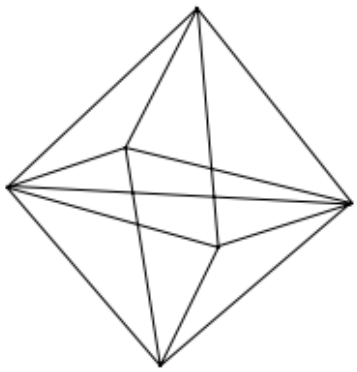


**Рис. 4. Третий шаг**

На первых двух шагах качество сетки не ухудшается, что позволяет повторить их произвольное число раз. Только на шаге 3, в момент разбиения октаэдров на пирамиды, происходит ухудшение качества сетки. Поскольку шаг 3 выполняется только один раз, итоговое ухудшение качества сетки не зависит от степени измельчения.