Билет №1

Преимущества и недостатки многопроцессорных систем. Области применения многопр. систем.

За послед. годы прочув-ть СРИ выросла в 1280 раз, но не за счет частоты, а за счет пр-ти ядра и кол-ва ядер. Требуется уметь делать несколько независ. действий за 1 такт.

- Нужно оценивать, какая система может быть исполь-з. и насколько алгоритм „ложится" на эту систему

* Вычислит. мощность суперкомп. Огромна, но:
  - образована суммой мощностей множества узлов
    ⇒ опасность, что в процессе рассчетов одно выйдет из строя (не зависит от решаемой задачи)
* Когда ядро одно, рост пр-ти ⇒ уменьш. времени. Теперь это не так:
  - последов. прогр-мы не могут работать быстрее
    ⇒ требуются спец. алгоритмы (зав. от задачи)

Области применения многопроц. систем:

1) Сокращ. времени реш-я выч-но-сложн. задач
   необходимы алгоритмы, обладающ. достат. высокими показат. ускорения и эффективности исп-я процессорной мощности

2) Сокращ. времени обработки больш. об. данных
   Задача интерактивной визуализации, основная цель – быстрая реакция на запросы пользователя
   ⇒ основное требование: высокоскор. обраб. запроса (высокая эффективн. отходит на второй план)

3) Решение задач реального времени

решения принимаются в жестко установленные сроки ⇒ допустимы системы, часть которых простаивает кроме малых промежутков времени ⇒ предъявляются требования по пр-ти, но не по эффективности

4) Создание систем высокой надежности

традиционный способ – полное дублирование вычислений, результаты сравниваются между собой ⇒ обеспечивается живучесть системы; производительность поддерживается на неизменном уровне, требования к эффект. не предъявляются

- вычисл. газовая динамика
- молекулярн. динамика
- символьные вычисления
- квантовая хромодинамика
- термоядерный синтез

"

* потребность в суперкомп. высока
* эффективность исп-я суперкомп. мала
  - последовая пр-ма не используется
  - обмены, синхр-я

* число ядер определ. минимальным объемом вычислений на одном ядре

* Сложно сокращать время моделирования физ. процесса, но можно повышать сложность решаемых задач

Билет №2

- Отличия между параллельн. и послед-ми вычислит. арх-ми с точки зрения принципов фон Неймана

У нас есть ноутбук (многоядерный), мы говорим, что это машина фон Неймана. Есть суперкомпьютер, говорим, что это не машина фон Неймана.

- Как правильно провести границу между понятиями?
- Грань пробегает по двум принципам: (некоторые из принципов фон Неймана)

* Принцип последоват. программного управления
  - прогр. состоит из набора команд, которые вып-ся процессором одна за другой в опред. посл-ти

  (вообще говоря, изначально - м. фон Неймана - параллельная, пример - сложение чисел - разряды обрабат. параллельно)

  ⇓

  по одному этому принципу разделить не удается

* Принцип адресуемости памяти
  - память состоит из пронумерованных ячеек, процессору в произв. момент времени доступна любая ячейка

- Эти принципы не всегда нарушаются сразу оба.

Принцип послед. пр. упр. наруш., если больше 1 вычислительного устройства: нет общего времени

Принцип адресуемости может выполняться на системах с общей памятью, но он нарушается при распределенной организации памяти

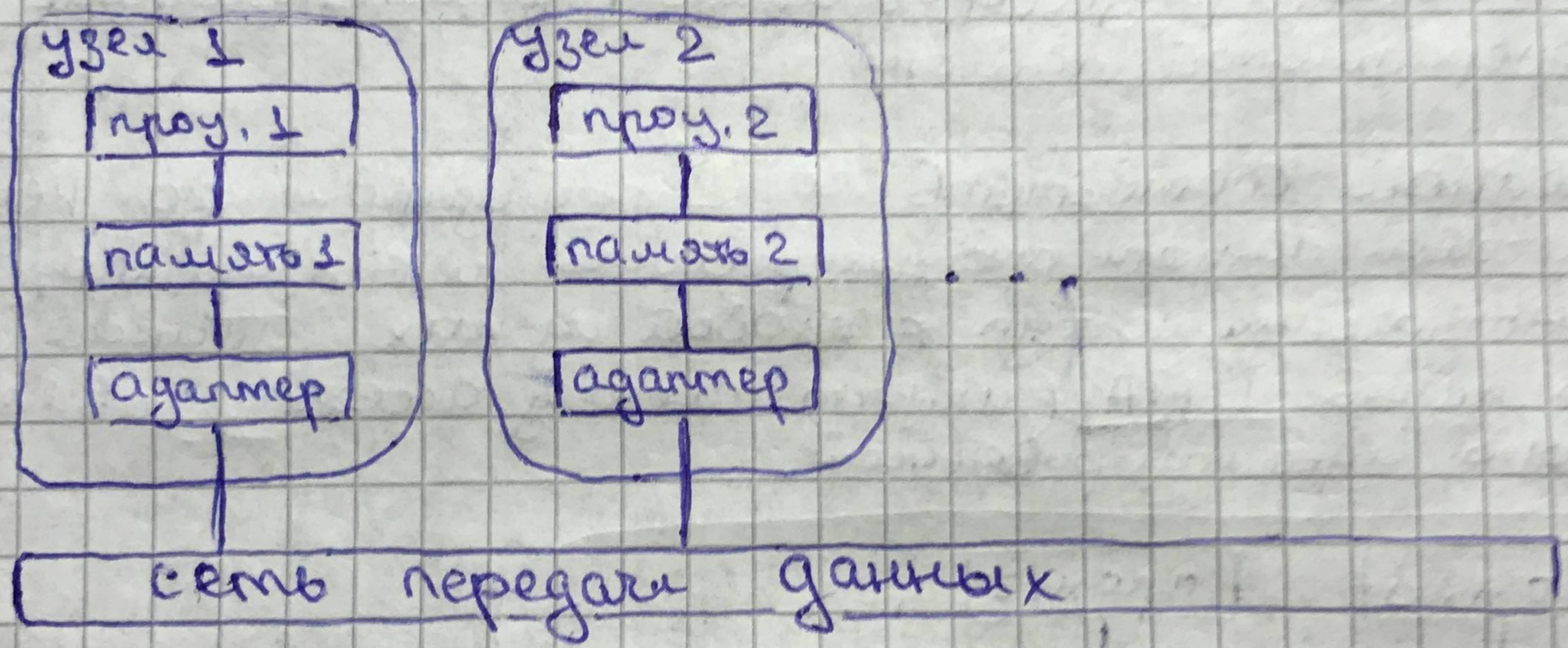
(см. Билет №3)

Билет №3 лист №1 из 2

- Виды многопроцессорных систем. Многопроц. сис-мы с общей памятью: UMA, NUMA, ccNUMA. Многопроц. сис-мы с распр. памятью. Гибридные, Векторные.

I Вычисл. системы с распред. памятью

Системы конструируются на основе объединения процессорных узлов каналами передачи данных. В каждом узле есть процессор, доступная ему опер. память и сетевой адаптер. Опер. память узла недоступна процессорам из других вых. узлов



- Взаимод. процессов, вып-ся на разных узлах, возможно лишь путем передачи данных через комм. сети.

В таких системах нарушаются оба принципа ф. Неймана:

- принцип послед. пр. упр.: нарушается, поскольку взаимный порядок вып-я операций процессами, вып-ся на разных узлах, не определен. (не существ. единого времени)

- принцип адресуемости нарушается, т.к. одному узлу не доступна память на других узлах

Преимущества:

1) низкая стоимость : высок. отношение цены к производительности

2) Высокая масштабир. : практически неограничт.

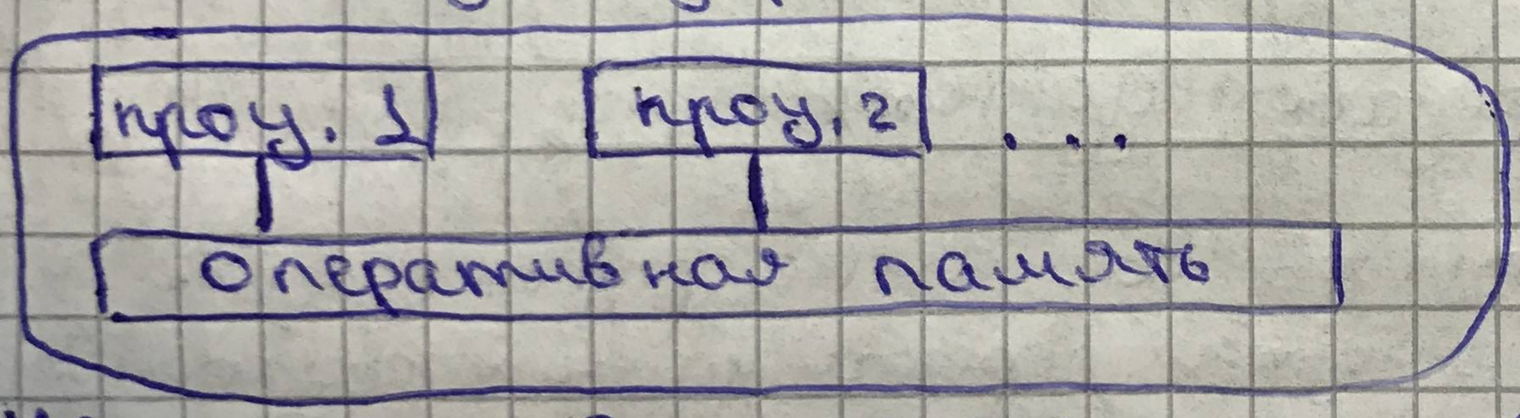
3) Возможность параллелиз. модулей уже функционир. системы

Недостаток: сложность разработки эффективных параллельн. алгоритмов

II. Вычислит. системы с общей памятью

Несколько процессоров имеют симметричный доступ к оперативной памяти

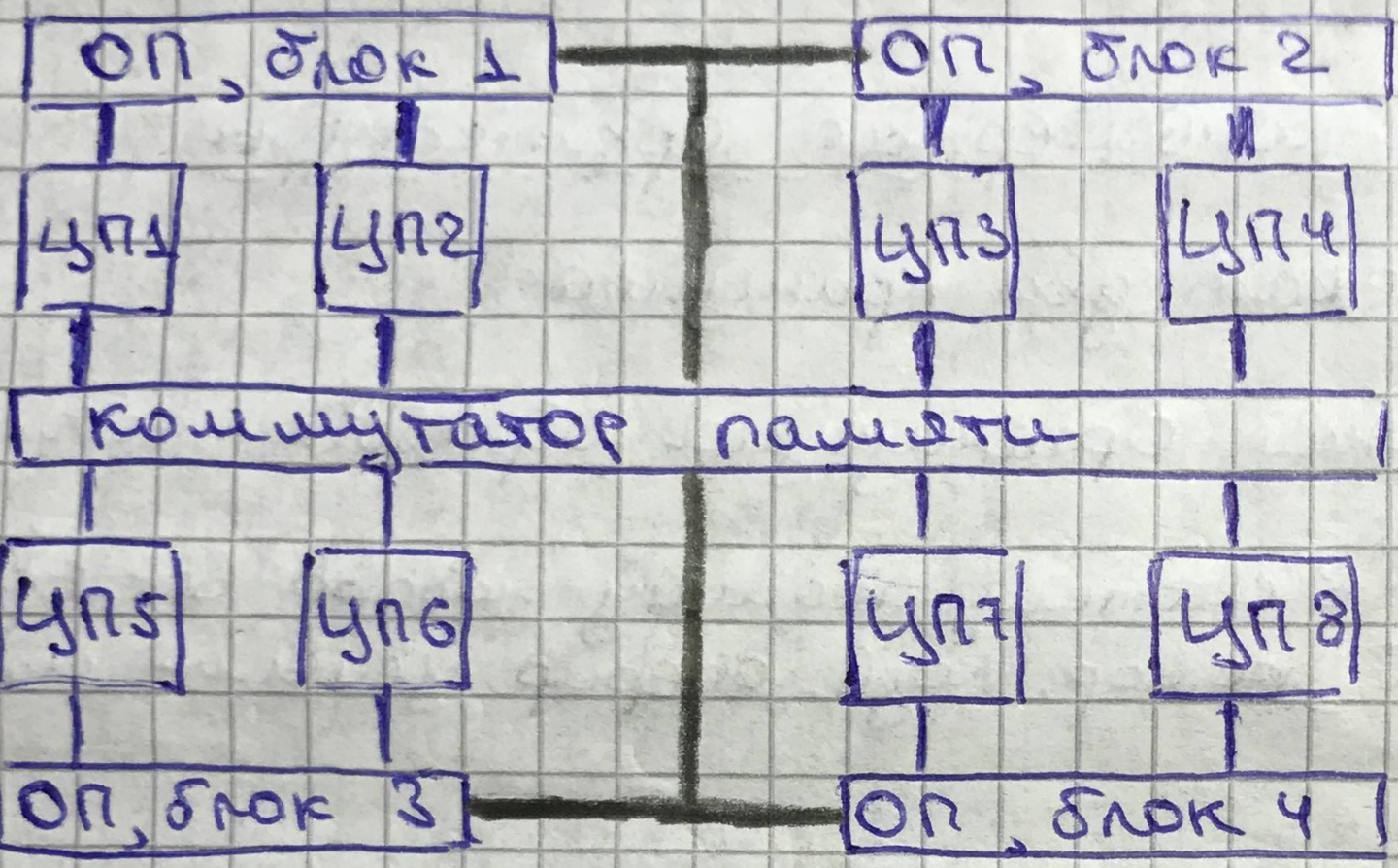
В полном соответств со 2ым принципом фон Неймана все ячейки памяти пронумерованы и каждый процессор имеет равный доступ к любой из них. Такая модель наз. UMA (uniform memory access); является сильно идеализированной



На практике времена доступа при большом числе процессоров не совпадают

Есть более реалистичная схема. Оперативная память разделена на независимые блоки. Каждый процессор имеет непосредственный доступ не ко всем блокам памяти, а только к одному. (несколько проц. могут иметь доступ к одному) Время обращ. процессора к своему блоку памяти меньше, чем к другим.

Обращение к другим блокам обеспечивается коммутатором оперативной памяти. Такая модель в зависимости от деталей реализации называется NUMA (non-uniform -//-) или ccNUMA (cache coherent -//-).



При равной производительности системы с общей памятью существ. дороже, чем сис. с распред. Они значительно хуже масштабируются.

Важная особенность: коммутатор памяти предоставляет непосредственный доступ к памяти любому процессору

III. Гибридные архитектуры

Нет систем, в точности соответств. арх-ре распред. памяти. Современный суперкомпьютер объединяет большое кол-во узлов, в пределах каждого из которых имеется общий доступ к памяти. (как правило NUMA или ccNUMA).

IV Векторно-конвейерные архитектуры

В основе лежат два принципа:

1) конвейерная организация обработки потока команд
2) введение в систему команд набора векторных операций, котор. позв. оперир. целыми массив. данных

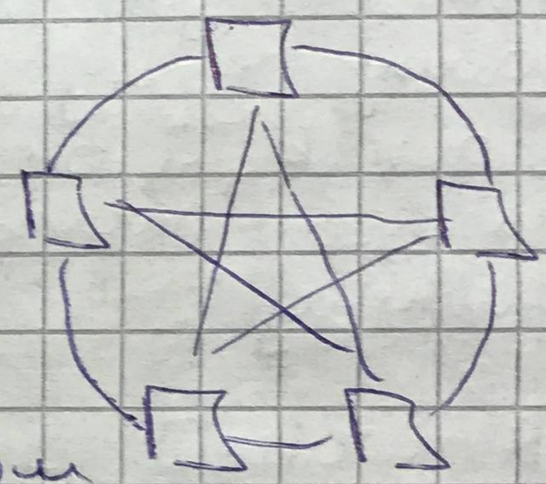
Билет №-4

Топологии многопроцессорных вычислит. систем

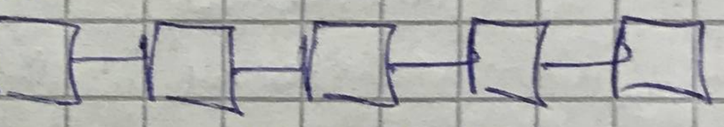
Топология - структура линий коммутации между процессорами вычислительной сис-мы. Она определяется с учетом возможностей эффективной технической реализации и анализа интенсивности информационных потоков при паралл. решении наиболее ресур. вычисл. задач.

К числу типовых топологий обычно относят:

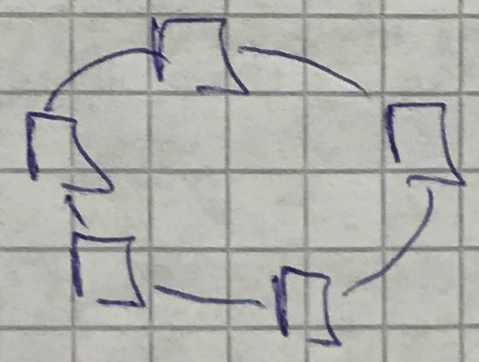
1) <u>полный граф</u> - система в которой между любой парой процессоров существует прямая линия связи ⇒ минимальные затраты при передаче данных, но сложно реализуется при большом числе процессоров



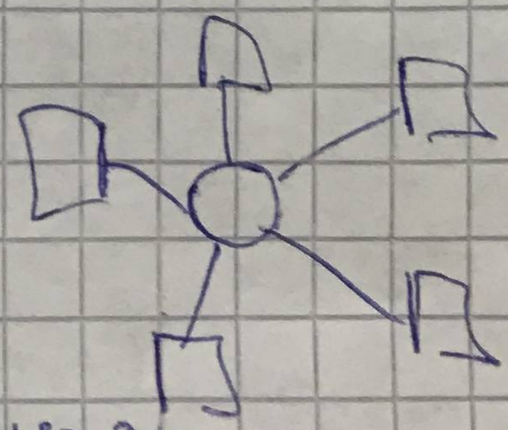
2) <u>линейка</u> - система, в которой каждый процессор имеет линии связи только с двумя соседними ⇒ просто реализуется, соответствует структура передачи данных в конвейерной обработке



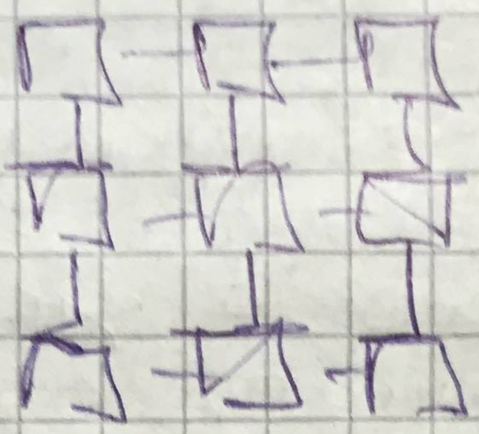
3) <u>кольцо</u> - получается из линейки соединением первого и последнего



4) <u>звезда</u> - система, в которой все пр-ры имеют линии связи с некоторым управляющим; эффективно при централизов. вы-ях



5) <u>решетка</u> - система, в которой граф линий образует прямоугольную сетку (n-мерную). может быть достаточно просто реализована и, кроме того, эфф. используется при параллельных методах



6) <u>k-ичный n-куб</u>

реализуется куб, имеющий n измерений, причем каждое измерение содержит k узлов. Каждому узлу назначен n-разрядный номер в СС с осн. k, и он связан с узлом, номер которого отличается только в одной цифре и только на единицу

k-ичный 1-куб - кольцо
k-ичный 2-куб - двумерный тор
k-ичный 3-куб - трехмерный тор
4-ичный 2-куб - плоская решетка 4×4
2-ичный n-куб - гиперкуб

Билет №5

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

Для взаимод. программ, запущенных на сист. с распред. памятью, необх. воспользоваться ф-ями передачи данных между процессорами. Эти ф-и могут быть самыми разными, но среди них есть общее.

- ⇒ можно ввести некоторые характеристики

- 1) <u>Пропускная способность сети</u>

  говорит, сколько времени затрачивается на передачу 1 байта между двумя процессор.

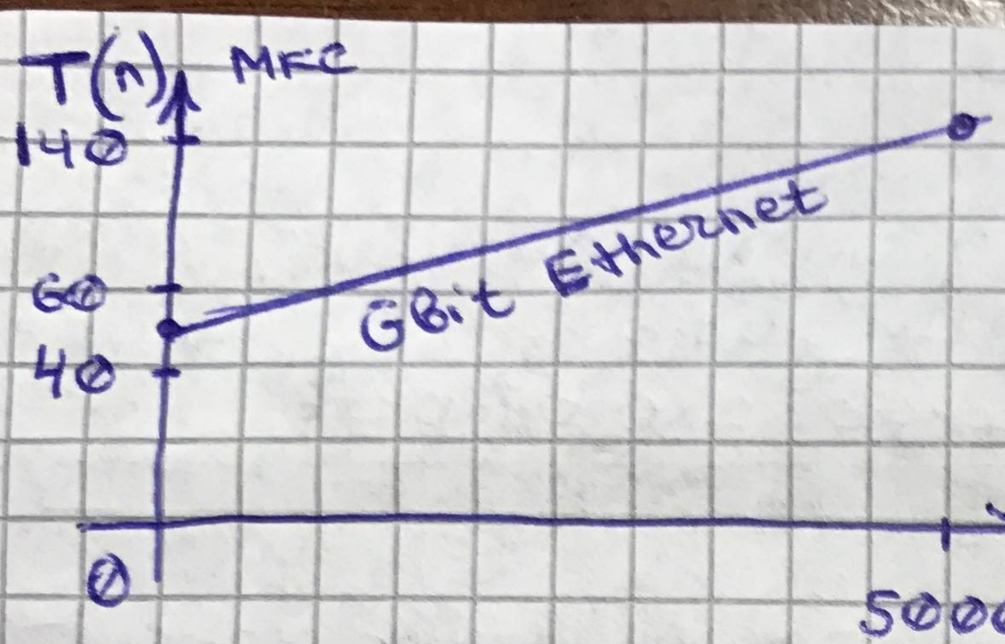
  2) <u>латентность</u>

  время, которое требуется, чтобы передача данных начала осуществляться

Т.о. время, требуемое для передачи данных между двумя процессорами:

- $$T(n) = n \cdot T_{1\text{байта}} + T_{\text{латентности}}$$

- Графики зависимости времени показывают, что время растет достаточно медленно, а начальный интервал латентности является значительным



Значит, на передачу 1 и 100 байт будет затрачено примерно одинаков. время. По возможности следует заменить множ-во передач коротких сообщ. передачей одного большого, иначе каждый раз добавляем латентность.

Вывод: следует стремиться к тому, чтобы данные передавались большими блоками.

Билет №6

- Кэш-память и ее влияние на эффективн. вып-я многопоточных приложений. Когерентность кэш.

Кэш-память - промежуточный буфер с быстрым доступом, содержащий копию той информации, кот. хранится в более медленной памяти, но с наибольшей вероятностью может быть отпущена запрошена.

На процессорах, оснащенных кэш-памятью, время выполнения послед-ти операций может быть значительно ниже суммы времен выполнения каждой операции по отдельности, за счет меньшего числа обращений к более медленной памяти

На вычислительной системе с общей памятью время параллельного вып-я двух независимых послед-ей команд на двух процессорах может быть существенно больше времени последовательного вып-я этих же команд на одном процессоре. Причина - конфликты кэш-памяти. Кэш-память может „устаревать" из-за изменений, производимых другим процессором.

Для управления такими конфликтами и поддержания соответствия между разными кэшами существует когерентность кэша.

Кэш называется когерентным, если вып-ся следующие условия

1) Если проц. P записывает знач-е в пер-ю X, то при след. считывании X он должен получить ранее записанное знач-е, если между его записью и чтением другой процессор не производил запись в X.

г) Операция чтения пер. X проц-ом P_1, следующая после того, как другой пр. P_2 осуществил запись в X, должна вернуть записанное зна-е, если другие пр-ры не изменяли X между двумя операциями

з) Записи в одну и ту же ячейку памяти должны быть последовательными.

Предполагается что чтение и запись происходит мгновенно. Вообще, это не так. За видимость записанного зна-я отвечает модель консистентности памяти

Кэш-память также может стать причиной скачка сверхлинейного ускорения многопроцесс. системы. При значительном увеличении числа процессоров все данные могут уместиться в их кэшах.

Билет №7

Внутренний параллелизм. Степень параллелизма.

Не любая задача поддается распараллел., даже если для нее известен хороший последовательный алгор.

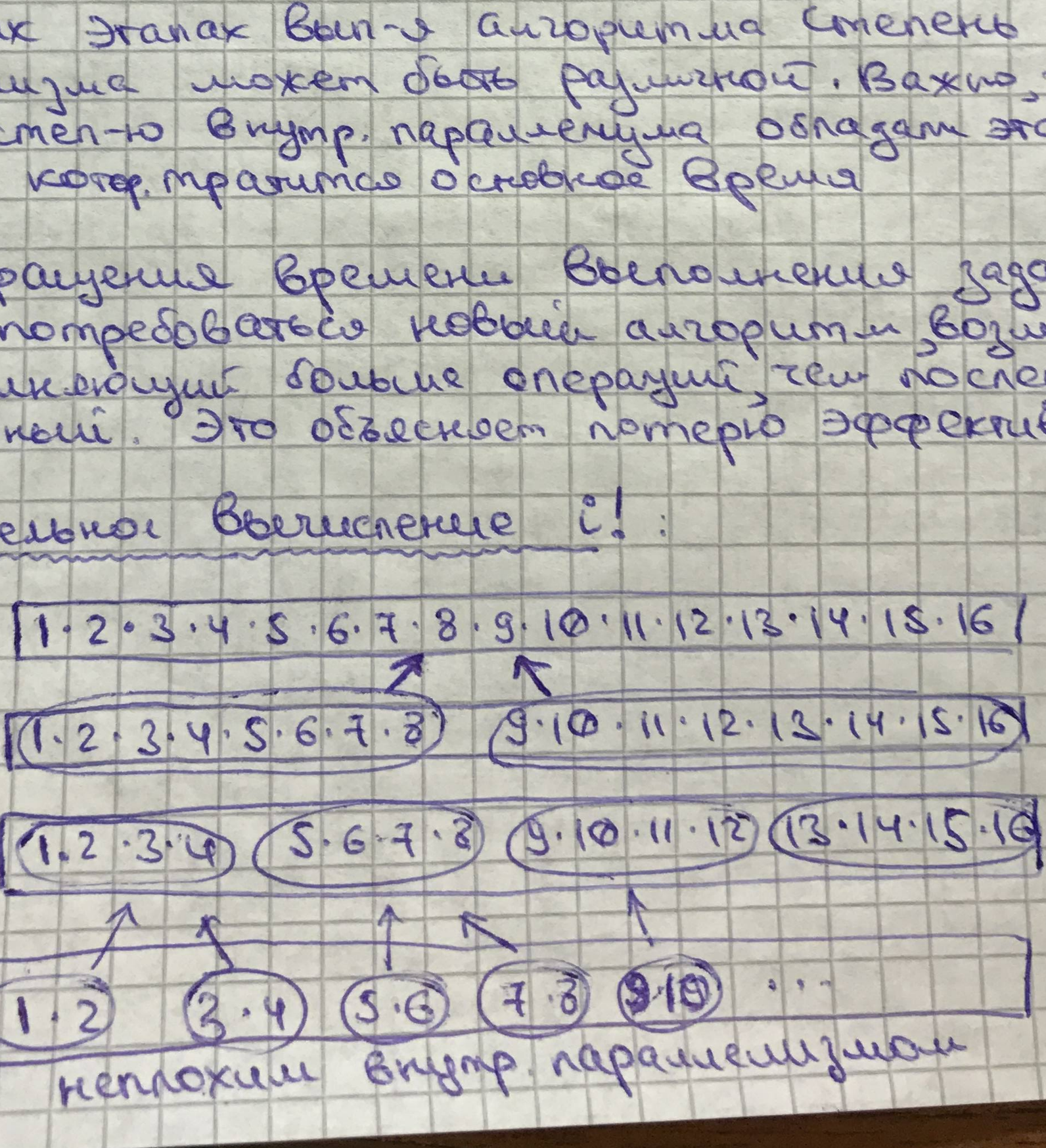
Алгоритм обладает внутренним параллелизмом, если в нем присутствуют действия, для которых допустимо одновременное выполнение. Это св-во именно алгоритма, оно не имеет отношения к вычислительной системе.

Степень параллелизма алгоритма – это число операций, выполняемых которых можно в любом порядке.

На разных этапах вып-я алгоритма степень параллелизма может быть различной. Важно, чтобы высокой степ-ю внутр. параллелизма обладали этапы, на вып-е которых тратится основное время

Для сокращения времени выполнения задачи может потребоваться новый алгоритм, возможно, выполняющий больше операций, чем последовательный. Это объясняет потерю эффективности

параллельное вычисление i!:



обладает неплохим внутр. параллелизмом

Сортировка пузырьком плохо параллелится;
но если передавать процесс объединение
результатов за O(n) (merge), то
получаем значительный эффект.

Билет №8

Ускорение, эффективность, масштабируемость. Закон Амдала.

Ускорение и эффектив. – две основные хар-ки, необх. для количеств. оценки св-в паралл. алгоритмов

Ускорение паралл. алг-ма – отношение времени выполнения последоват. алгоритма T_1 ко времени вып-я парал. алг. T_p на заданных p процессорах:

$$S_p = T_1 / T_p$$

Что такое „время вып-я алгоритма"? На практике:

1) Число выполняемых алгоритмом операций
   - –: операции, как правило, более высокоур., чем команды процессора
   - –: невозможно оценить время работы последовательностей операций (кэш, ...)
2) Замеряется время вып-я прогр-ы, реализующей алг. на некоторой вычисл. сис-ме
   - –: необходимость запрогр-ть ⇒ время работы программы, а не алгоритма ⇒ существенно зависит от ВС, СП и т.д.

Эффективность паралл. алг-ма – отношение ускорения к числу процессоров, на кот. она достигнута

$$E_p = S_p / p$$

При одинак. числе проц. большая эфф. обеспеч. рем-е задачи за меньшее время. Однако, если низкая эф-ть сохраняется постоянной при достаточно большом числе проц., то тоже интересно. Пример – сеть Бэтчера

Масштабируемость парал. алг-ма – наименьшее число процессоров, на которых достигается максимальное ускорение

$$M = min p^* : \forall p \; S_{p^*} \geq S_p$$

Ускорение – формальный показатель. Его знач-е существ. завис. от того, с каким именно послед. алг. вып-ся сравнение.

Будем считать „наилучшим" последовательный алгоритм, затрачивающий на реш-е задачи наименьшее время среди всех имеющихся в нашем распоряж. алгоритмов

⇒ аналогичные определения, но по отнош. к наилучш.

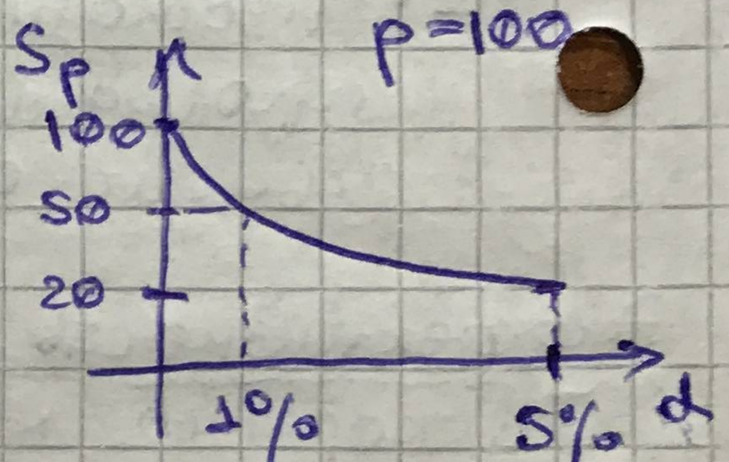
Пусть T_1 – время вып. алг. на одном процессоре
α – доля операций алг., вып-е кот. невозм. одноврем. с друг.
⇒ $(1-\alpha)$ – доля операций со степ. паралл. сколь угодно большой
на p проц. вып-ся за $\frac{(1-\alpha)T_1}{p}$
t_d – дополнит. время, потраченное по сравн. с послед. алг. (синхронизация, обмены)

⇓

ускорение, достигаемое на p процессорах:

$$S_p = \frac{T_1}{\left(\alpha + \frac{1-\alpha}{p}\right)T_1 + t_d}$$

При $\alpha \neq 0$, $t_d = 0$, $S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$



– Закон Амдала в предполож., что все действия вып. либо с мин., либо с макс степ-ю паралл., а потери с доп. операций отсутствуют

Билет №9

- Сверхлинейное ускорение. Возможные причины. Формальное преобр-е параллельного алг. в последовательный

Сверхлинейное ускорение - это ускорение, превышающее число используемых для расчета процессоров: $S_p > p$. Существуют как минимум две причины его возникновения.

1) Неравномерность условий выполнения

Вычислительная система может обладать нетривиальными свойствами.

Пример: чтение из большой БД. Когда процессоров мало - читаем всегда с диска. Потом с увеличением их числа растет ОП, следов. → скорость, когда БД помещается в ОП. Второй скачок - помещается в кэше.

2) Алгоритмическая причина

Выбор неудачного последовательного алг-ма.

⇓

Стремимся выбирать наилучш. послед. алг-м

Но как быть, если не знаем, какой последов-ый алгоритм эфф. наилучший?

⇓

Формальное преобр. парал. алг. в "наилучш." посл.

Пусть парал. алг. предполаг. вып-е на двух проц.
⇒ Выполнение двух слабосвяз. послед. процессов преобразует исходный алг. в послед.:

Шаг 1: Выберем для вып-я первый процесс. Выполним действия, предшеств. оп-ам взаимод. между процессорами

Шаг 2: Выберем для вып. втор. пр. Вып-им все дейст. предш. оп-ам взаимод. между проц.

Шаг 3: Вып. оп-ы, соответств. требуемому взаимодействию.

Шаг 4: Перейти к Шагу 1.

Если сверхлин. ускорение было вызвано неудачны наследов. алг., то данный процесс позволит решить проблему.

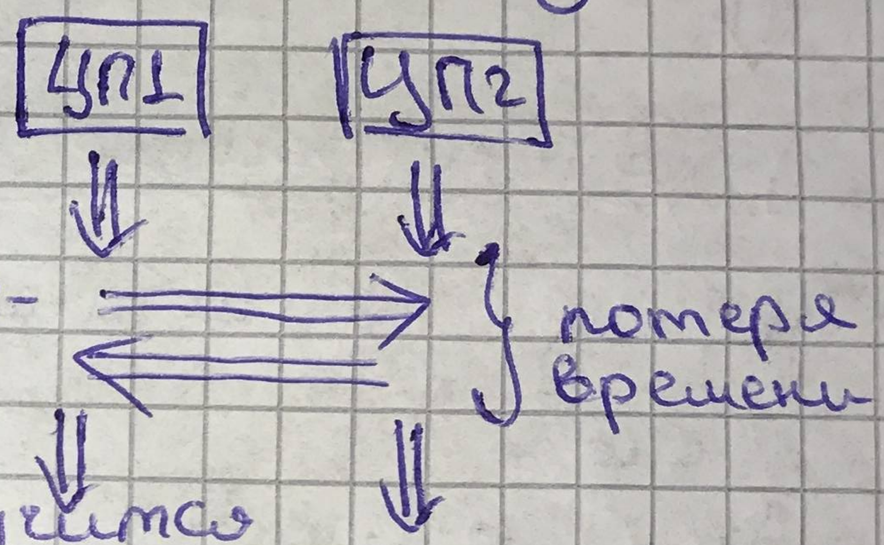
Билет №10

Накладные расходы в параллельных алгоритмах

- Это все те операции, которые отсутствуют в наилучшем последовательном алгоритме

1) Обмен данными

- Есть два процессора, они хотят передать друг другу данные



Можно попробовать сделать асинхронно вместе с вычислениями

Но полностью скрыть не получится

Часть задержки (латентность) связана с вызовом подпрограммы передачи данных

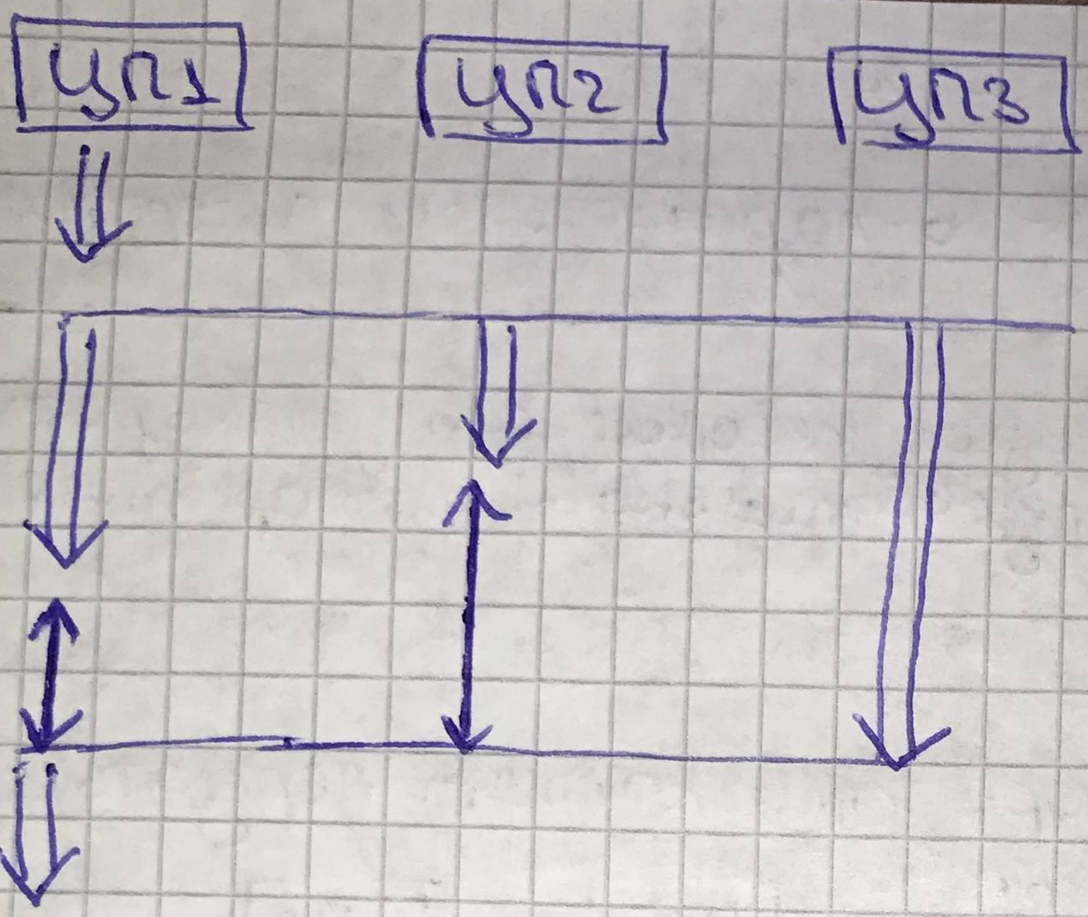
2) Синхронизация

- Есть три процессора, они делают разные операции; не бывает, чтобы все делали одинаково

Один процессор закончил раньше, остальные еще нет, но нужно ждать; причина - неудачно распределили ресурсы

- Но некоторые процессоры могут поникать частоту и/-да перегрева, т.е. на уровне алгоритма ничего сделать не можем;

но обычно есть возможность перебалансировать в рантайме ⇒ динамическая балансировка



3) Дублирование операций

Пример: сумма массива на двух процессорах

```
S=0;
for (i=0;i<n1;i++)
  S+=a[i];
send(S);
```

```
S=0;
for (i=n1;i<n;i++)
  S+=a[i];
send(S);
```

```
Recv(S1);
Recv(S2);
S=S1+S2;
```

На двух процессорах выполняется одна и та же операция: формально-параллельно, но время выполнения данной операции не уменьшилось

4) Новые операции

Пример - подсчет факториала (см. Билет №7) много лишних умножений, но позволяют сделать быстро

Шаг	УП1	УП2	УП3	УП4
1	1·2	3·4	5·6	7·8
2	1·2·3	1·2·3·4	5·6·7	5·6·7·8
3	1·2·3·4·5	1·2·3·4·5·6	1·2·3·4·5·6·7	1·2·3·4·5·6·7·8

Все это составляет t_d в 3. Амдала (см. Билет №8)

Билет №11

Модель параллельной программы. Выч. сис-ме с распред. памятью

- Полагаем, что вып-е паралл. прогр на распр. памяти начинается с запуска заданного числа одинаковых процессов ⇒ в команде запуска указывается название прогр. и число процессов.
- Программе выделяется некоторое множество процессоров, число кот. в общем случае может не совпадать с числом процессов. Возможен даже запуск нескольких процессов на одно-одноядерном процессоре.

В любом случае, с точки зрения программиста, у каждого запущенного процесса будет своя ОП, к которой не имеют доступа другие процессы.

Каждая копия программы запускается на отдельном Виртуальном процессоре, не имеющем с точки зрения стиля отличий от физического одноядерного процессора.

- Каждая из запущенных копий программы получает от опер. системы 2 переменные.
- Первая - число запущенных процессов. Вторая - номер запущенного процесса.

С помощью этой нумерации процессы могут обмениваться данными между собой.

:

Билет №12

Методы передачи данных (синхр., асинхр., ...)

I. Синхронный метод

Предполагает, что при передаче данных указывается адрес передаваемых данных, их размер (число байт или эл-тов) и номер процессора, с которым осуществить взаимодействие

Синхронные функции:

- Send (номер проц., адрес данных, размер данных) - синхронная передача данных
- Recv (-//-) - синхронный прием

Эти ф-и срабатывают только после того, как они обе вызваны. Иначе - состояние ожидания ⇒
⇒ блокирующие вызовы.

Эти функции затрачивают на передачу данных наименьшее время, но только при условии, что были вызваны одновременно; иначе - ожидание.

II. Асинхронные методы

Для сокращения подобных непроизводительных потерь исп-ся асинхр. методы. Отличие - практически сразу возвращают управление вызвавшей их программе.

Асинхр. ф-я не выполняет передачу данных, а размещает заказ на эту операцию. Фактически в этот момент порождается дополнит. нить, в рамках которой и осуществляется передача с помощью синхронных ф-й.

Асинхр. методы делятся на буферизованные (ABSend) и небуферизованные (ASend, ARecv, ASync)

С одной стороны асинхр. методы позволяют оптимизировать работу системы, но, с другой стороны, накладывают ограничение на послед-ть выполняемых оп-и.

Ответственность за коррект. исп-е этих методов лежит на прогр-те. Следует использовать дополнительную фун-ю синхронизации — ASync

ASend и ARecv блокирующими не являются, а ASync позволяет дожидаться вып-я передачи

Использование буфериз. передачи в принципе позволяет не использовать ASync. Но лучше это делать, т.к. могут возникать ошибки в буферах.

Асин

Билет №13

Разделяемые ресурсы. Семафоры Дейкстры, критическая секция, монитор

Совместное использование ресурсов двумя или более процессами, когда каждый из них некоторое время владеет этими ресурсами, назыв. ~~раздел. ресурсов~~

Если бы все действия над ними были неделимыми - ~~или - атомарными~~ - проблем бы не возникало. Но в реальности это не так.

Семафор - средство синхр-и проц-в. Различают бинарные и общего вида (рассм. общего вида, по терминологии Дейкстры)

Семафор - целочисленная неотрицательная пер-я, над которой можно выполнять только две атомарные операции P и V.

V(S) - атомарная неблок.; увеличивает S на 1.

P(S) - атомарная блокирующая:

1) если $S > 0$, то P(S) уменьшает S на 1;
2) иначе процесс, выполняющий P(S), переходит в режим ожидания. Управление будет возвращено, когда кто-то сделает V(S).

Семафор - глобальная переменная; могут применяться при общей памяти

Но программы с семафорами трудно отлаживать: ошибки сложны в локализации

Существование сокращает число ошибок позволяет чисто выделить в прогр-е критические интервалы и их реализацию с пом-ю мониторов.

Критический интервал - это последовательность действий, выполняемых которые должны как одна неперывная операция

В принципе, семафоров должно быть достаточно. Но для уменьшения числа ошибок рекомендуется закрывать ресурсы с помощью построенных на семафорах мониторов.

Монитор - набор подпрограмм, инкапсулирующих все обращения к защищаемому ресурсу. Доступ к ресурсу, защищенному с помощью монитора, допустим только с помощью подпрограмм монитора

Билет № 14

- Барьер. Схемы реализации и оценки времени вып-я. Реализация барьера на основе синхронных обменов. -//- на семафорах

Барьер - это функция, вызываемая всеми процессами, участвующими в акте данной синхронизации, ни один из вызвавших эту ф-ю процессов не завершит ее выполнение до тех пор, пока все процессы не вызовут эту ф-ю.

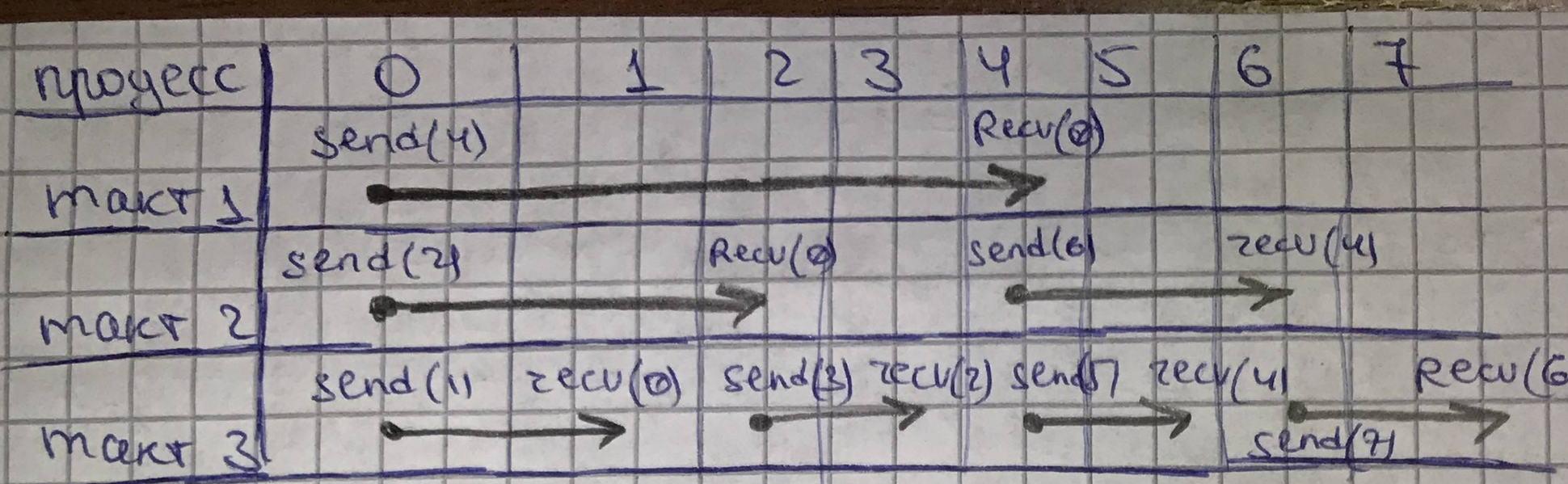
Операция предполагает взаимодействие множества процессоров, поэтому требует существенного времени

При возможности, исп-е барьеров стоит избегать

Реализация на основе синхр. обменов

Алгоритм рассылки сообщений, использующий метод сдваивания и работающий за $T_p = \tau_s \log_2 p$

```
SendOneToAll () {
    for (p1=1; p1<p; p1*=2) { }
    if (rank>0) { // определяем номер младшего
                  // в rank разряде, отличного от 0
        for (delta==1; (delta & rank)==0; delta*=2);
        from = rank-delta; // определяем номер
                           // передающего процессора
        Recv(from);
    } else {
        delta=p1;
    }
    for (delta/=2; delta>0; delta/=2) {
        next=rank+delta; if (next<p) Send(next);
    }
}
```



Изменим порядок выполнения тактов этого алгоритма на обратный и изменим направления каждой передачи инф-и на обратное

Получим алгоритм RecvOneFromAll().

Тогда барьерная синхронизация может быть реализована:

```
Barrier() {
    RecvOneFromAll();
    SendOneToAll();
}
```

Вызов ф-и Recv...() гарантирует, что первый процесс не закончит ее выполнение до тех пор, пока все остальные процессы не выполнят ее только одно.

Через вызов Send...() на всем процессорам будет распространена инф-я о том, что барьерная синхр. состоялась.

Время вып-я алгоритма: $T = 2\tau_s \log_2 p$

Есть альтернатива - „бабочка"

Билет №14 лист №2 из 2

Реализация с помощью семафоров

```
n = number of processes;
count=0;
mutex=Semaphore(1);
Barrier = Semaphore(0);

P(mutex);
count+=1;
V(mutex);

if (count==n) V(Barrier);

P(Barrier);
V(Barrier);
```

Билет №15

Модель парал. прогр., выполняющейся на вычислительной сис-ме с общей памятью. Нити (легковесн. процессы, threads)

Определим *процес. послед-й процесс* как послед-ть действий, выполняемых автономно (независ. от окруж. обстановки)

Будем считать, что кроме явных редких моментов связи процессы вып-ся совершенно независимо — *слабо связанные последовательные процессы*

Под *параллельной программой* будем понимать всю совокупность слабо связанных процессов, необходимых для вып-я требуемых вычислений.

Выполнение начинается с запуска единственного последовательного процесса. В тот момент, когда появляется возможность параллельного выполнения команд, вызывается функция порождения дополнительной *нити* кода. Число порождаемых нитей определяется требованиями алгоритма и числом доступных процессоров; может быть конкретное выполнение.

У каждой нити есть недоступные другим нитям локальные пер-е, хранимые в стеке нити. Кроме лок. пер-ных, каждая нить имеет доступ к общим глоб. пер-ным. Они как-то в общем адресном пр-ве. Каждая нить может их читать и изменять.

Билет №16

Методы статической и динамической балансировки загрузки процессоров

- Недетерминированность параллельных программ заставляет использовать определенные подходы к созданию надежного ПО для многопроц. систем.

- Несмотря на отсутствие формальных доказательств корректности рассматриваемых методов, их правильность подтверждена многолетней практикой. Их сила - в логической простоте. Именно простота обеспечивает их относительно легкую воспроизводимость и применимость для решения самых разных задач

* методы статической балансировки пр-ов
  - метод сдваивания (см. Билет № 17)
  - метод геометрия. параллелизма (см. Билет №18)
  - метод конвейерного параллелизма (см. Билет №20)

* Методы динамической балансировки загрузки процессоров
  - метод коллективного рег-я (см. Билет №19)
  - метод диффузной бал. заг-ки (см. Билет № 21)
  - метод серверного параллелизма (см. Билет № 22)

Билет №17

Метод сдваивания

- Рассм. задачу суммирования эл-тов массива
- из n эл-тов.

Оценим время работы последов. алгоритма:

```
S := a[0];
for (i=1; i<n; i++)
  S := S + a[i];
```

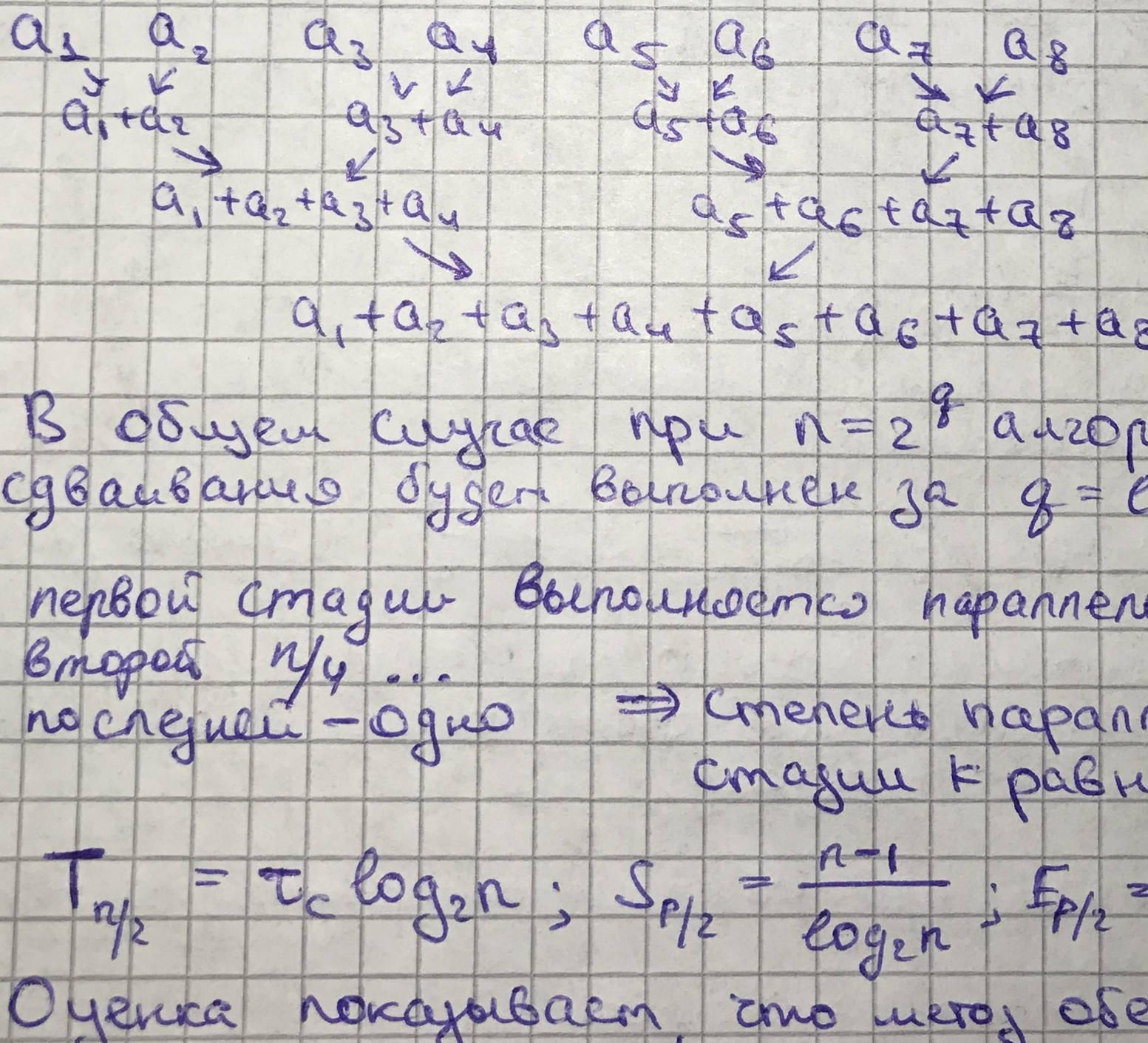
Пусть τ_c - время на одно сложение двух чисел

Тогда время работы послед. алгоритма:

$T_1 = (n-1)\tau_c$

- Сущуб последов. алгоритм; степень параллелизма равна 1.

Построим паралл. алг., используя метод сдваивания. Пусть $p=4$ и $n=8$.



- В общем случае при $n=2^q$ алгоритм сдваивания будет выполнен за $q = \log_2 n$ стадий.

На первой стадии выполняется параллельно $n/2$ действ
на второй $n/4$ ...
на последней - одно $\Rightarrow$ Степень параллелизма на стадии k равна $n/2^k$

- $T_{n/2} = \tau_c \log_2 n$; $S_{p/2} = \frac{n-1}{\log_2 n}$; $E_{p/2} = \frac{n-1}{\log_2 n} \cdot \frac{2}{n} \approx \frac{2}{\log_2 n}$
- Оценка показывает, что метод обеспечивает высокое ускорение, но достаточно низкую эф-ть.

Основная причина низкой эф-ти в том, что бо́льшая часть процессоров простаивает.

Такой подход может быть применяемым в узкоспециализированных системах, но на практике применим другой подход.

Массив длины n разбивается на p равных частей, обработка каждой из которых происходит на соотв. процессоре.

На первом этапе каждый из процессоров находит сумму n/p эл-тов соответствующей ему части.

На втором этапе с пом-ю метода сдваивания определяется окончательная сумма.

$$T_p = \tau_A \cdot \frac{n}{p} + \tau_A \log_2 p$$

$$S_p = \frac{n\tau_A}{\tau_A \cdot \frac{n}{p} + \tau_A \log_2 p} = p \frac{1}{1 + \frac{p}{n} \log_2 p}$$

$$E_p = \frac{1}{1 + \frac{p}{n} \log_2 p}$$

При $p \ll n$ метод обеспечивает высокую эф-ть и ускорение.

В частности, при $p = \frac{n}{\log_2 n}$: $S_p = \frac{p}{2}$; $E_p = 50\%$

Метод сдваивания хорошо применять, когда данные уже распределены по процессорам. Иначе, все время уйдет на передачу.

Применяется при выполнении редукционных операций
- сумма эл-тов массива
- мин. эл-т массива
- широковещ. рассылка данных ...

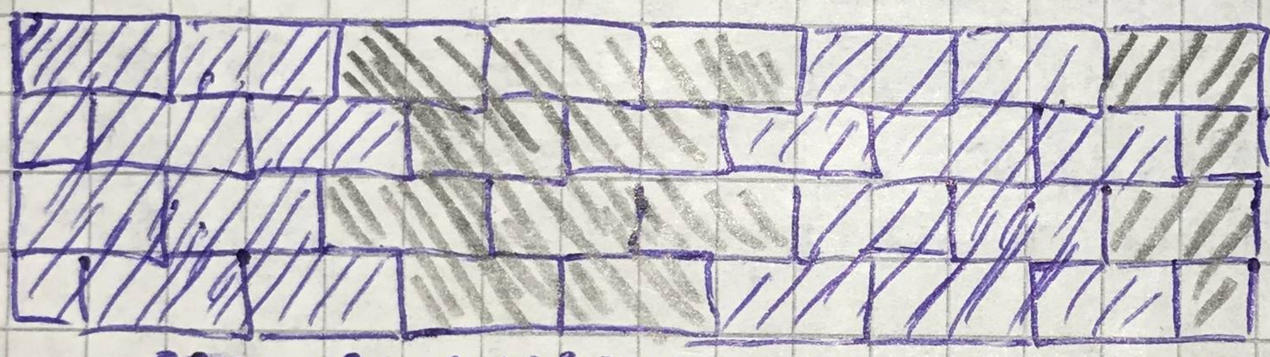
Билет №18 лист №1 из 2

Метод геометрического параллелизма

Предназначен для выполнения множества однотипных действий над однородными данными (взаимодействие)

Решая задачу построения „стены Фокса", можно разбить кладку на разные по длине участки и поручить постройку каждого участка отдельному каменщику. В этом случае все каменщики могут начать работу одновременно, укладывая нижний слой кирпичей.

Перед укладкой очередного слоя каждому каменщику следует убедиться, что кирпичи предыдущего слоя уложены не только на его участке, но и на прилегающих. Возникают вынужденные паузы из-за синхронизации.



геометрическое решение

n - число кирп. в ряду
k - высота стены
τ_c - врем. укл. 1 кирп.
T_p - время реш. на p пр.

В случае одного проц. - последов. укладка nk кирпичей

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

В случае p проц., каждому - участок длиной $\frac{n}{p}$
Объем полезной работы - $\tau_c \frac{n}{p}$. Однако для укладки нового кирпича на границах фрагмента, следует убедиться, что это можно сделать

Пусть процессы выполн. на ВС с распред. памятью, каждый - на отдельном процессоре. Перед укл. очередного кирп. на левой границе в процессе с номером rank надо получить сообщение от rank-1 ⇒ на rank-1 посылка в rank. и в обратную сторону ⇒ между двумя гранич. проц. - 2 сообщ

⇒ процесс, не 0 и не p-1 должен обмениваться с соседями четырьмя сообщениями ⇒
⇒ наименьшее время работы.

$$T_p = \tau_c \frac{kn}{p} + 4k\tau_s \quad (*)$$

Но! Время будет таким отнюдь не при любом алгоритме передачи данных

Пусть используем синхронные функции

Алгоритм 1:

```
for (шаг=0; шаг<k; шаг++)
{
    for (кирпич = rank*n/p ; кирпич<(rank+1)*n/p; кирпич++)
        уложить(кирпич);
    if (rank>0)   Send(rank-1, "кирпич уложен");
    if (rank<p-1) Send(rank+1, "кирпич уложен");
    if (rank>0)   Recv(rank-1, "место готово?");
    if (rank<p-1) Recv(rank+1, "место готово?");
}
```

Этот алгоритм неработоспособен, т.к. первые же идут фз передачи ⇒ взаимная блокировка

Можно исправить ситуацию, поменяв местами вызовы:

```
if (rank>0)   Send(rank-1, "кирпич уложен");
if (rank<p-1) Recv(rank+1, "место готово!");
if (rank<p-1) Send(rank+1, "кирпич уложен");
if (rank>0)   Recv(rank-1, "место готово");
```

Этот алгоритм будет работать за конечное время, но не за минимальное

$$T_p(кн) = \tau_c \frac{kn}{p} + 2pk\tau_s$$ (рисунок: >)

- Алгоритм обладает св-вом локальности ⇒ взаимные обмены между далеко стоящими
- друг от друга по цепочке процессорами не должны влиять друг на друга.

Соответствующая модификация программы основана на идее разбиения процессов на пары, осуществляющие обмены одновременно и независимо друг от друга

- Сначала взаимодействуют процессы с номерами
- 2k и 2k+1, а затем передают между собой данные 2k-1 и 2k.

Алгоритм 2:

—//—

```
if (rank % 2) {
    if (rank>0) Send(rank-1, "кирп. уложен");
    if (rank>0) Recv(rank-1, "место готово?");
    if (rank<p-1) Recv(rank+1, "место готово?");
    if (rank<p-1) Send(rank+1, "кирп. уложен");
} else {
    if (rank<p-1) Recv(rank+1, "место готово?");
    if (rank<p-1) Send(rank+1, "кирп. уложен");
    if(rank>0) Send(rank-1, "~~место~~ кирп. ул.");
    if (rank>0) Recv(rank-1, "место готово?");
}
```

(рисунок: )

Именно этот алгоритм достигает миним. времени (*), при этом:

$$S_p = p \frac{1}{1 + 4\frac{p}{n}\frac{\tau_s}{\tau_c}} \qquad E_p(кл) = \frac{1}{1 + 4\frac{p}{n}\frac{\tau_s}{\tau_c}}$$

Зкал-е ускорения и эфф. не зав. от высоты стека, а только от ширины каждого участка

Высокие показатели метода геом. паралл. обусловлены:

1) при увелич. числа процессоров время на их взаимод. не меняется и остается относ. малым

2) фактически нет новых операций, снижающих эффективность

Билет №19

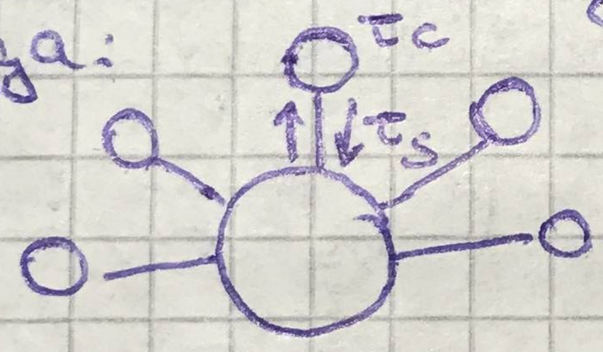
Метод коллективного решения

Как быть, когда заранее ничего нельзя сказать об относительной производительности и трудоемкости задач?

Метод коллективного реш-я предназначен для обработки множества не связанных между собой заданий с непредсказуемым временем обработки каждого из них на процессорах с неопределенной производительностью.

Будем говорить о задаче укладки паркета (любая плитка м.б. уложена независимо). Максимальная степень параллелизма равна общему числу плиток пк.

Все процессы разбиваются на два множества в первом - единственный процесс (управляющий), который обеспечивает динамическое распределение работ. Во втором - все остальные процессы (обрабатывающие), которые обесп. непоср. вып-е работ. Топология соединения процессоров - звезда:



Каждому обраб. мод-ю управляющий поручает укладку некоторой плитки. Обрабат. мод. выполняет задание и вновь обращается к управл. для получ-я следующего. В общ. случае, каждая получ. для обработки плитка может нах-ся сколь угодно далеко от ранее обработанных этим мод-ом плиток

⇒ динамическая балансировка

Причины, приводящие к простою процессоров:

1) Требуется синхр-я между управл. и обрабат процессами. На вып-е этих действий требуется время, поэтому в τ_{s_1} начале работы все процессоры, кроме первого, простаивают в ожидании задания

2) Выполнивший задание процесс тратит время на передачу рез-та управляющему. τ_{s_2} ← Если несколько пр-ров одновременно ⇒ простой.

3) Вероятен простой части пр-ров в конце работы, когда все задания уже распределены по пр-рам. Пр-ры, закончившие ранние всех, новой работы не получат.

Оценим метод при ограничениях:

- пр-ры имеют одинак. пр-ть;
- задачи имеют одинак. трудоемкость;
- $\tau_s = \tau_{s_1} + \tau_{s_2}$
- пренебрегаем управляющим пр-ром ⇒ p обрабатыв-х

Тогда каждому процессу на обработку $\frac{N}{p}$ заданий ($N \gg p$). Время на получение данных о задании — τ_s и его обработку - $\tau_s + \tau_c$. Считаем $p \ll N$.

$$T_p = \frac{N}{p}(\tau_c + \tau_s); \quad S_p = p\frac{\tau_c}{\tau_s + \tau_c}; \quad E_p = \frac{1}{1 + \frac{p}{\tau_c}}$$

Если $\tau_s < \tau_c$ ⇒ Высокая эф-ть; иначе - выдавать большие задачи

Пусть за одно обращ. передается инфа об r зад. Макс. ускорение достиг. при $r = \sqrt{N \cdot \frac{\tau_s}{\tau_c}}$

$$S_{max} = \sqrt{N \cdot \frac{\tau_c}{\tau_s}}$$

Конкретное з-ие r при распред. очередного задания следует выбирать, опираясь на доп. инфу о св-вах задачи.

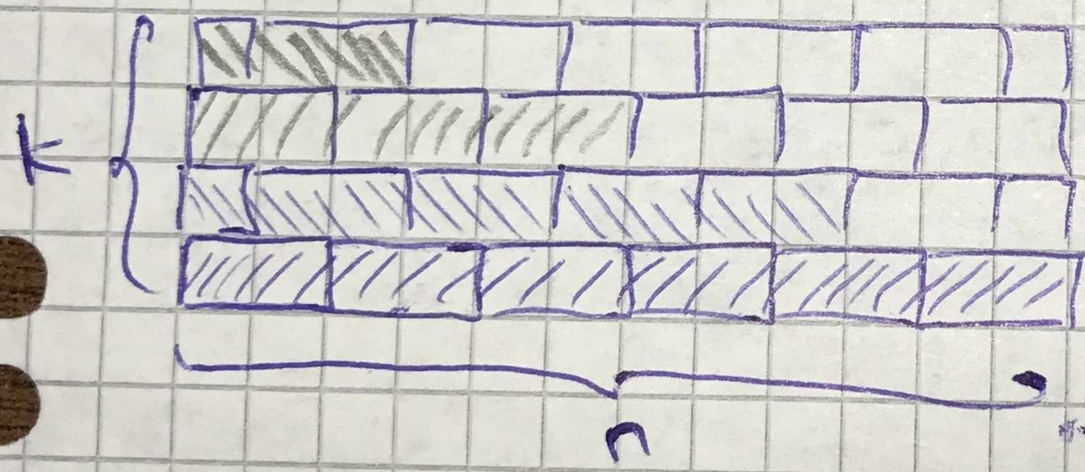
Метод конвейерного параллелизма

Продемонстрируем на примере стены Фокса. Конвейерная распр-е предполагает, что первый каменщик полностью укладывает первый слой кирпича, второй - второй и т.д. После того, как первый проф. закончит первый слой, он начинает укладку p+1 слоя.

При таком подходе стартовать может только первый каменщик, остальные ждут, пока будут уложены нижние слои. Аналогичная ситуация в конце, когда некоторые уже закончат.

При условии одинаковой пр-ти и если стена достат точно высок. и длинна, бо́льшую часть времени все каменщики будут равномерно загружены.

Условие одинаковости времени укладки ∀ кирпича существ.



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

$$T_p(kn) = \tau_c \frac{kn}{p} + \underbrace{k \frac{n}{p}}_{\text{потому что } \forall \text{ задания требуется обмен с нижними процессорами}} \tau_s$$

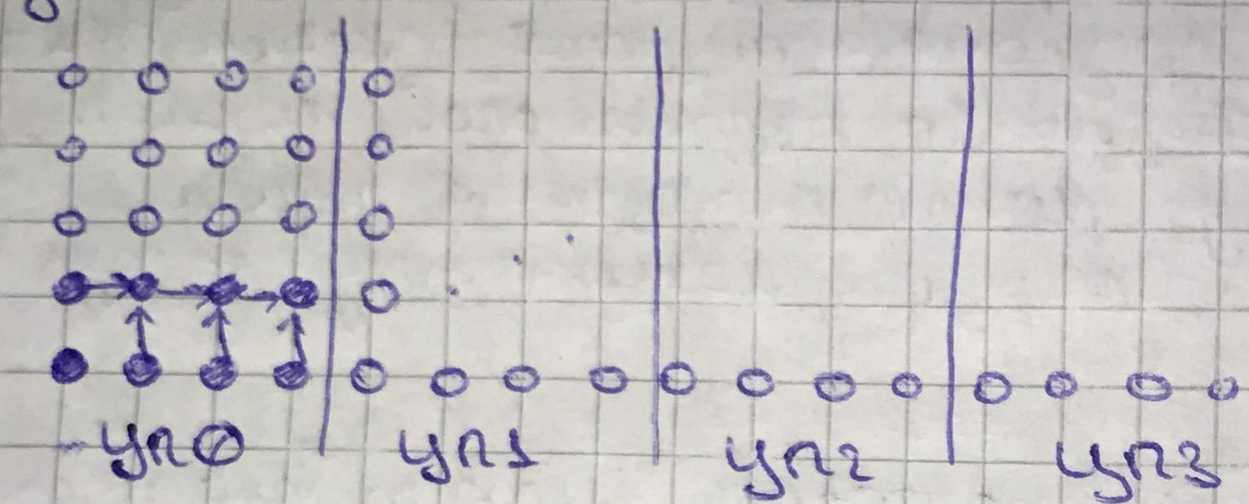
(Это даже без учета начальной/конечной задержки)

$$S_p(kn) = p \frac{1}{1 + \frac{\tau_s}{\tau_c}} \; ; \; E_p(kn) = \frac{1}{1 + \frac{\tau_s}{\tau_c}}$$

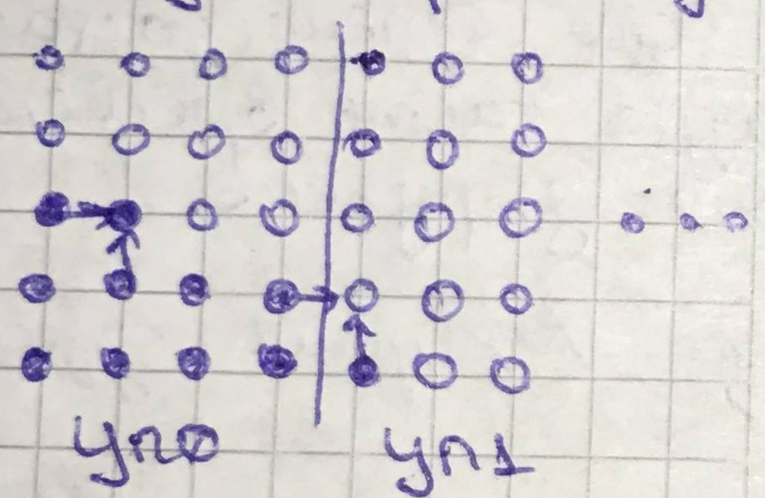
не зависят от размеров сетки

необходимо учитывать от чего передача данных при каждом задании

Другая стратегия распред-я данных: как в геометрич. паралл-ме. Вначале выполняет действия можно только на процессоре 0:



Остальные простаивают, пока на УП0 не будут обраб. Np точек. После этого инф-ия о точке с номером Np-1 следует передать УП1 и работу теперь могут продолжать две пр-ра



Поскольку на ∀ слое ∀ процессор только 1 ед принимает и 1 ед передает данные, оценки приобретают вид:

$$T_p = \tau_c \frac{kn}{p} + 2k\tau_s$$

$$S_p(kn) = p \frac{1}{1 + 2\frac{p}{n}\frac{\tau_s}{\tau_c}}$$

$$E_p(kn) = \frac{1}{1 + 2\frac{p}{n}\frac{\tau_s}{\tau_c}}$$

Данные оценки не учитывают начальные и конечные задержки.

Последний к работе приступит процессор p-1. Произойдет это после того, как будут обработаны $\frac{n}{p}(p-1)$ точек первого ряда:

$$T_{start} = (p-1)\left(\frac{n}{p}\tau_c + 2\tau_s\right)$$

До тех пор, пока первый процессор не закончит укладку верхнего ряда, в работе будут принимать участие все процессы. В таком режиме каждый из процессов будет обработано $(p-k)$ рядов.

$$T_{middle} = (k-p)\left(\frac{n}{p}\tau_c + 2\tau_s\right)$$

В заключение потребуется время на укладку $\frac{n}{p}(p-1)$ точек последнего ряда.

$$T_{end} = (p-1)\left(\frac{n}{p}\tau_c + 2\tau_s\right)$$

Общее время:

$$T_{all} = T_{start} + T_{middle} + T_{end} = (p+k-2)\left(\frac{n}{p}\tau_c + 2\tau_s\right)$$

Полагая $p+k >> 2$, получим:

$$S_p^{all} = p\,\frac{1}{\left(1+\frac{p}{k}\right)\left(1+2\frac{p}{n}\frac{\tau_s}{\tau_c}\right)}$$

$$E_p^{all} = \frac{1}{\left(1+\frac{p}{k}\right)\left(1+2\frac{p}{n}\frac{\tau_s}{\tau_c}\right)}$$

Особенности: а) эффективен при $p \ll k$

1) Сеть должна быть достаточно высокой, иначе - накладные расходы на старт и конец преобладают

2) Макс. степень паралл. - $\min(n, k)$.

3) макс. ускор. $\frac{pk}{p+k} \leq \frac{k}{2}$

Конв. паралл. часто используется при реш-е систем СЛАУ с помощью итерационных методов

Билет № 21

Метод диффузной балансировки загрузки

Предположим, что некоторое распределение узлов вычислительной сетки по процессорам уже известно, но, по каким-либо причинам, не является оптимальным.

Причины дисбаланса вычисл. нагрузки

1) неудачное начальное распределение
2) не всегда есть возможность априори знать трудоемкость обработки узлов сетки
3) используемые процедуры могут обладать разной не известной заранее трудоемкостью
4) трудоемкость обработки каждого из узлов расчетной сетки может отличаться на разных моментах модельного времени
5) пр-ть пр-ров может меняться

Рассмотрим задачу вычисления нового распр-я узлов сетки по процессорам:

Пусть расчетная сетка содержит n узлов, и на шаге j процессор i обработал n_i точек за время t_i. Предполагается, что t_i не включает затраты на ожидание других.

Тогда, в соответствии с основной идеей метода диффузной балансировки, можно определить новое распределение n_i' узлов сетки по процессорам, например, след. образом:

$$n_i' = n \frac{n_i / t_i}{\sum_{j=1}^{p} \frac{n_j}{t_j}}$$

← работает, если трудоемкость обр. каждой точки одинакова

То есть, данная формула применима, если ~~дисбаланс~~ дисбаланс вызван ~~разной~~ разной производительностью процессоров

В других случаях требуется применять иные подходы и формулы

Билет №22 лист №1 из 2

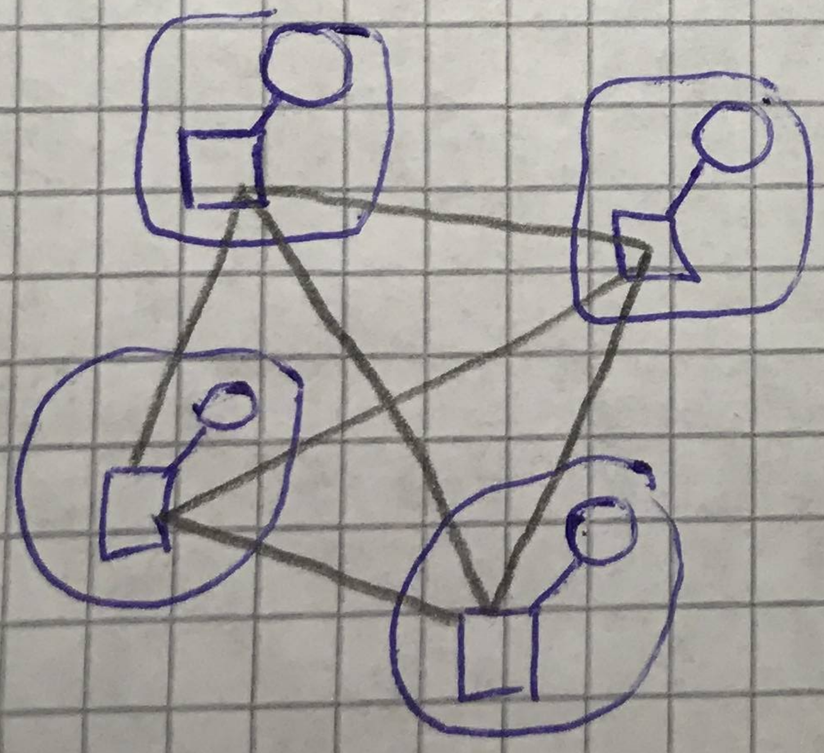
Метод серверного параллелизма

- Является развитием метода коллективного решения, но не требует наличия управляющего процессора, позволяет избежать:
  - потерь времени на пересылку всех данных
  - последовательного характера распределения заданий

Основные положения

1) отсутств. централ. сбор данных
2) отсутствует центр. сбор инфории. о "горячих" точках и текущей загрузки системы
3) преимущественная обработка лок. точек
4) стратегия поиска работы
5) коммуникации на фоне обработки
6) вторичное перераспр. точек
7) возврат точек направляющих, лишних промеж. этапы
8) сообщ-е обраб. сразу же после получения
9) на каждый запрос поступает ответное сообщ-е
10) на каждом процессоре выполняются управляющий и обрабатыв. процессы

- На каждом из процессоров одновременно выполняются два параллельных процесса –
  - управляющий и обрабатывающий.



○ - обрабат. процесс
□ - управляющий пр.

Существенно разные задачи:
- выполнение вых-й
- перераспр. "горячих" точек

Действия обрабатыв. процесса

1) ожидание сообщ-я от упр. пр-са М;
2) если сообщ. содержит инф-ю о необх-ти завершить работу, то завершает
3) если сообщ. содержит данные, описывающие очередное задание, то направляет упр. пр-су сообщение с результатами расчетов

Т.о., обр-пр-с взаимод. только с 1 упр. пр-ом

Цикл управляющего процесса

1) проверка условий возможности обработки точки — да → начать обработку точки
2) проверка условий возм-ти посылки запроса на получ. необработанных точек — да → послать запрос
   — нет ↓
3) проверка условий заверш-я работы — да → завершить работу
4) получить очередное сообщ-е
5) обработать сообщение
6) перейти к началу цикла

ТП.О. каждый из процессоров заканчивает выполнение алгоритма при выполнении ВСЕХ следующих условий:

1) нет локальных необраб. точек
2) нет внешних точек
3) нет обрабатываемых точек
4) всем пр-рам был послан запрос на получение необработанных т-к
5) всем пр-рам было послано сообщ-е о том, что необработанные т-ки предоставлены быть не могут
6) от всех пр-ров получено сообщ-е о том, что необработанные т-ки предост. быть не могут
7) все локальные точки обработаны и получены ру-ли обр-ки всех переданных т-к

Важно гарантировать отсутствие никем не принятых сообщ-й ⇒ ответное сообщ-е

Контроль возникновения тупиков осуществляется всей системой контролирующего процесса

Билет № 23

Отладка параллельных приложений, вып-е которых сопровожд. недетерм. потоком сообщений

Бывают топологии, когда мы не знаем заранее, какие процессоры у каких и сколько будут что-то спрашивать

- Алгоритм может быть трудным. Как отлаживать?
- Дважды в одну ситуацию попасть не можем. Возникла ошибка - тупиковая ситуация, но повторить ее не можем.

Программа может считаться неделями, а потом через полгода заказчик скажет, что она повисла.

Логирование - в принципе можно, но если пишем в ОП, то тоже повиснет. Сеть дает памяти с сотен пр-ров практически невозможно.

(речь идет про архитектуру серверного парал.)

- На узле вводится еще один процесс, который
- контролирует состояние обычных переменных.

Например, кол-во обработанных точек. Если оно меняется, то логами можно пользоваться, что что-то не так. То есть мы получаем управление, а дальше можем делать уже что угодно.

- Этот доп. процесс не ест ЦП, не требует
- синхронизации => он может оставаться при передаче заказчику

Если задача состоит из нескольких сильно отличающихся вычислительных фаз, то может быть так, что на каких-то прах ещё считается одна, а на других уже другая.

Тогда на практике помощь их разграничить — подобие глобального барьера.

Функцию барьерной синхронизации рационально делать на коммуникационных процессах с помощью управляющего процессора.

Билет №24 лист №1 из 3

Параллельные алгоритмы сортировки данных.
- Слияние методом сдваивания, двустороннее слияние. Сеть сортировки. Параллельные алгор. сорт. для гибридных выч. систем

Сравнивать надо с хорошими последовательными алгоритмом.

- Две разные задачи:
- A: объем ОП одного узла достаточен
  ⇒ расположить N эл-тов массива a таким образом, чтобы $\forall i \in [0; N-2]\ a_i \le a_{i+1}$
  B: -//- не достаточен.
  ⇒
  - пусть массив можно разместить на p процессорах
  - пусть на пр-ре rank размещено n^{rank} эл-тов массива a^{rank}

$$N = \sum_{rank=0}^{rankmax} n^{rank}$$

  - расположить N эл-тов массивов a^{rank} таким образом, чтобы:

$$\forall\ rank \text{ и } i = 0, \dots, (n^{rank} - 2)$$
$$a_i^{rank} \le a_{i+1}^{rank}$$
$$\forall\ rank \quad a_{n^{rank}-1}^{rank} \le a_0^{rank+1}$$

⇓

Этапы:
- локальная (послед) сортировка на пр-рах
- перераспр. эл-тов массива между пр-ми

Самое эффективное: меньше 10^5 эл-тов - параллельная
большие - слияние
Dsort

Параллельная сортировка сбалансированием

Нерекурсивный dsort позволяет паралл. обр.
Пусть $n = 16$, $p = 8$

Вначале на каждом из пр-ров одновременно упорядочиваются размещенные на них массивы из n/p элементов каждый, на что требуется $(n/p) \cdot \log_2(n/p)$ операций.

На последующих шагах с номерами $j \geq 1$:

* процессор с номером $2^j i$ передает процессору с номером $2^{j-1}(2i-1)$ отсортированный фрагмент массива длиной $2^{j-1}\frac{n}{p}$, что требует времени, необход. для передачи $p\frac{n}{p} \cdot 2^{j-1}$ чисел
* процессоры с номерами $2^j i$ выполняют слияние пар фрагментов длиной $\frac{n}{p} \cdot 2^{j-1}$, что требует $\frac{n}{p} 2^j$ операций.

5 2 2 1 9 5 7 3 3 4 4 0 3 6 8 2

52 21 95 73 34 40 36 82 — 8 пр-ров

25 12 59 37 34 04 26 28 — 2 такта

1225 3579 0344 2368 — 4 такта

1223579 0233446 8 — 8 тактов

01223334456 78 — 16 тактов

Итого 30 тактов

- Т.о., общее время выполнения сортировки при $p = 2^k$ и n кратном p составит:
- $$T(n,p) = \frac{n}{p}\left(\underbrace{k_1 \log_2\left(\frac{n}{p}\right)}_{\text{сортировка}} + \underbrace{k_2(p-1) + 2k_3(p-1)}_{\text{передача данных}}\right)$$

Положим $k_1 \approx k_3$, поскольку на соответствующих шагах выполняются сходные операции слияния массивов

- Т.о., ожидаемое ускорение равно:
- $$S(n,p) = \frac{p}{1 + \left(2(p-1) - \log_2 p + \frac{k_2}{k_1}(p-1)\right)\log_n 2}$$

Такая сортировка проста в реализации, но эффективна только при обработке массивов ограниченного размера на небольшом числе процессоров с общей памятью.

Алгоритм двустороннего слияния

- Эффективный комбинаторный алг. на двух процессорах с общей памятью
- Для слияния на одном процессоре двух упорядоченных массивов по 3 эл-тов потребуется 16 тактов. В двухпроц. системе с общ.-пам. второй процессор может одновременно с первым обрабатывать массивы, начиная с эл-тов со старшими номерами, выбирая на ~~4~~ такте элемент с максимальным значением. Такое слияние потребует 8 тактов.

Алгоритм эффективен с точки зрения исп. кэша.

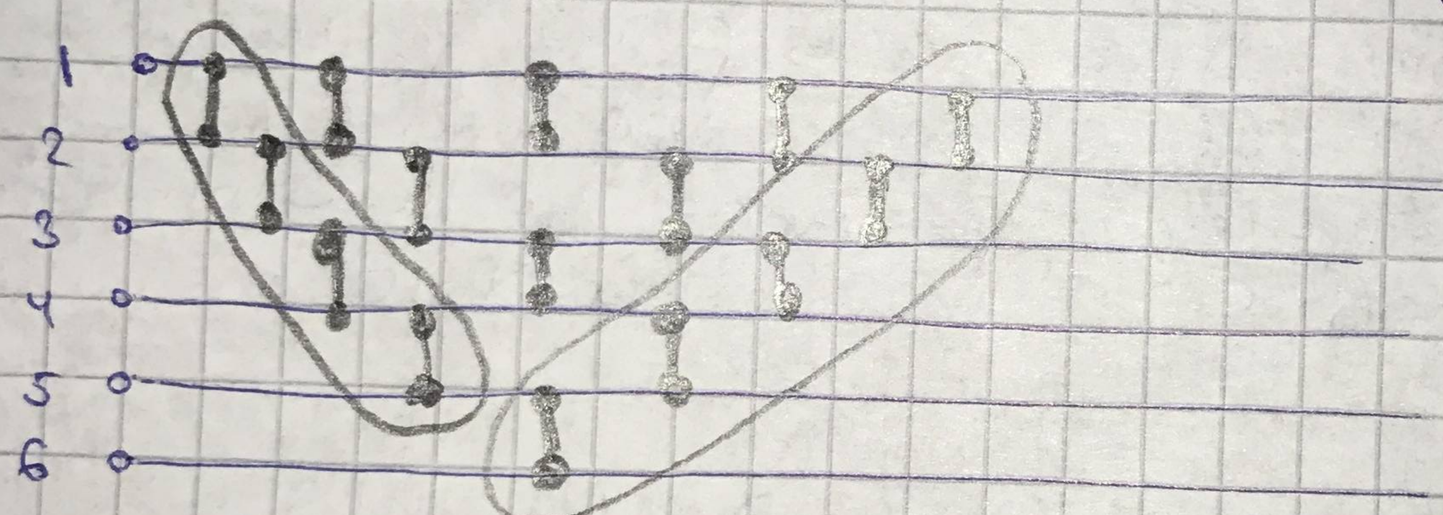
Оба предыд. подхода не позволяют эффективно использовать большое число процессоров и не обеспечивают возможность обработки данных, размер кот. превышает объем ОП одного узла

Сетки сортировки –

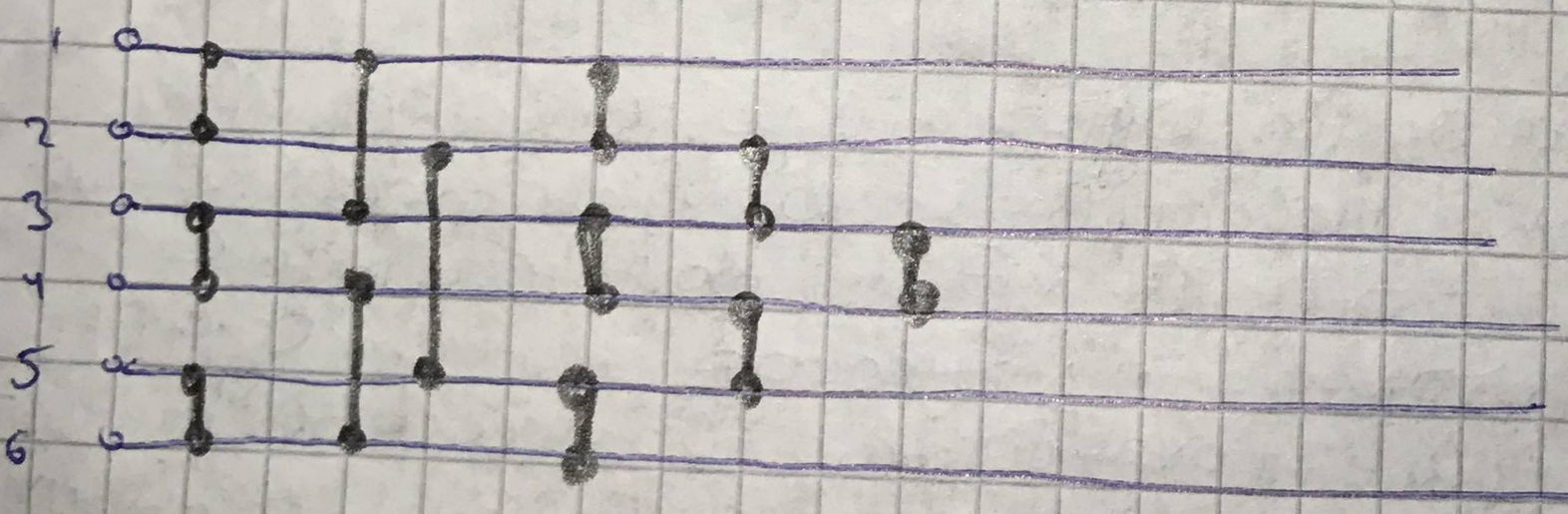
– Вид алгоритмов, в которых порядок вып-я операций сравнения и их кол-во не зависят от значения элементов сортируемого массива

Элементы – горизонтальные линии; обрабатываются компараторами: на верхнем выходе оказ. больший, на нижнем – меньший эл-т

Пример сети: пузырек n=6 S = 2n-3 = 9 шаги



минимальная: n=6, S=5



Билет №24 лист № 3 из 3

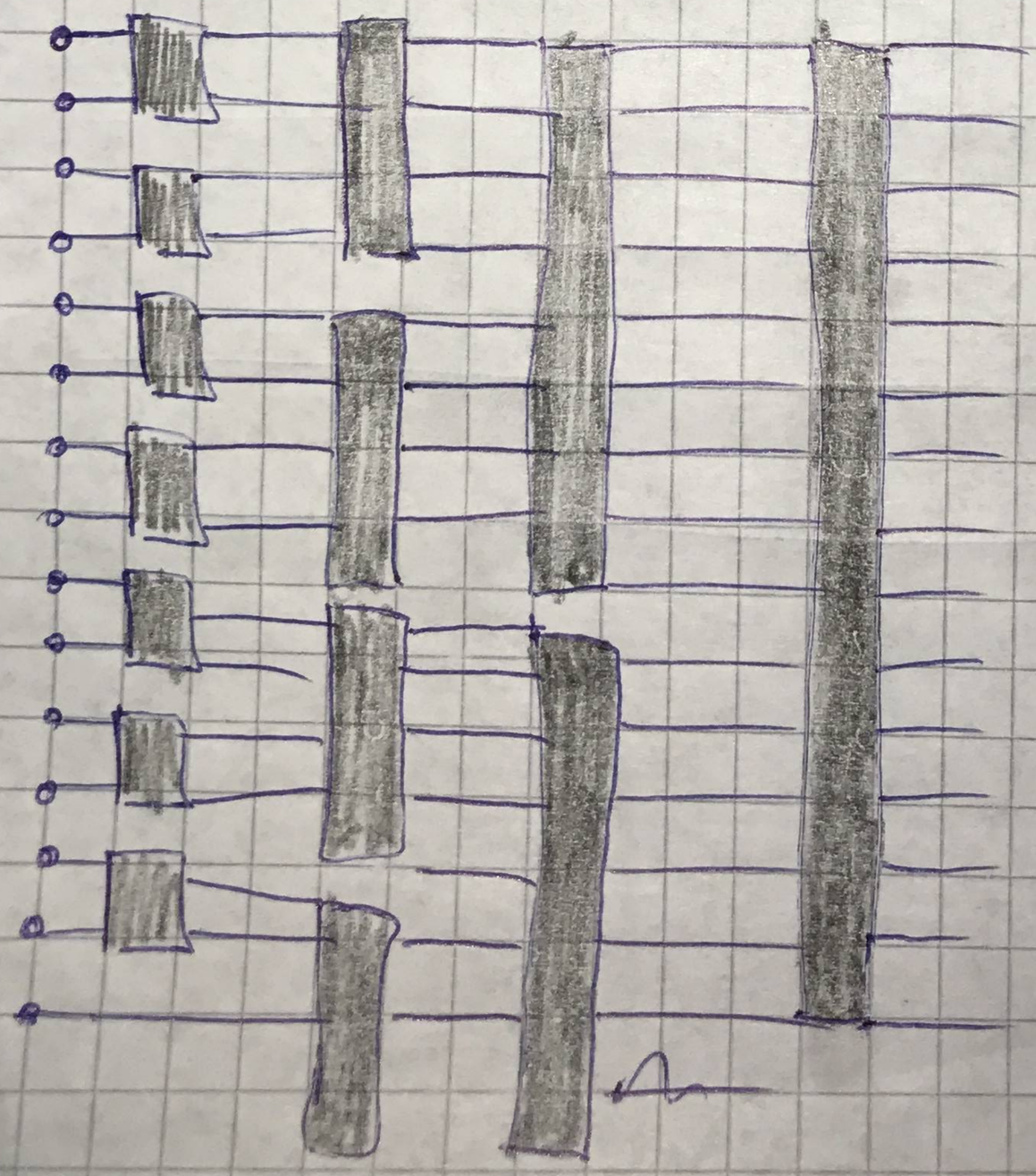
Можно строить минимальные сети для конкретных p, но они получены перебором ⇒ не масштабируемые

Сеть тетко-кекетной сортировки Бэтчера слияния

Пусть есть два отсортированных фрагмента

Выбираем из первого массива все четные строки (нумеруются с 0) и из второго все четные (тоже с 0). Выполняем обводку. Эти массивы между собой также покупаем с нечетными. Получим два отсортир. массиве - отдельно четных строк и отдельно нечетных. Потом ставим челочку компаратора начиная со второго сверху эл-та

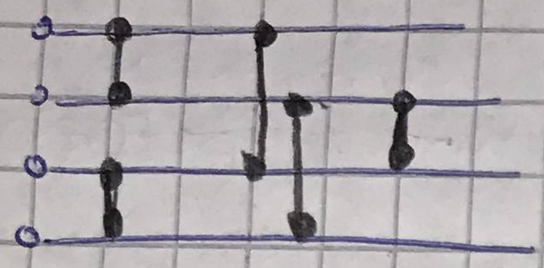
~ общая схема: сначала сортируем пары, потом четверки, потом восьмерки, потом объединяем



Оценка числа шагов:

$$S_p = \frac{\lceil \log_2 p \rceil (\lceil \log_2 p + 1 \rceil)}{2}$$

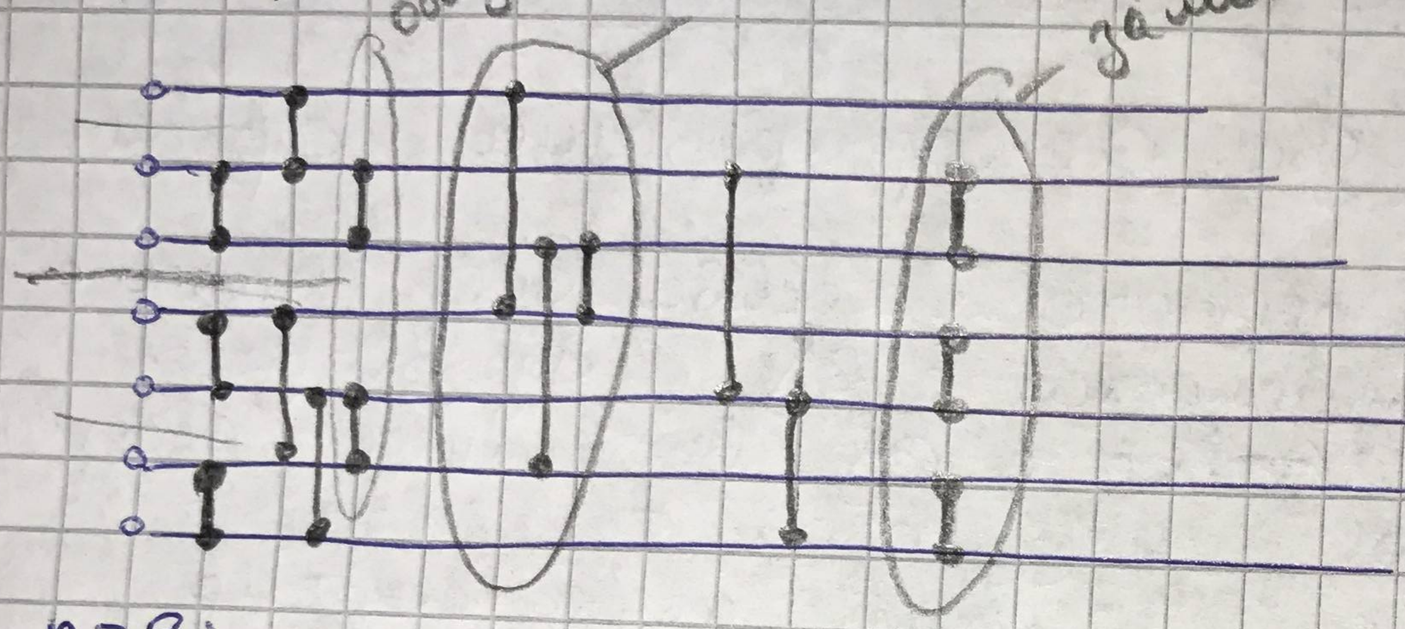
n=4:



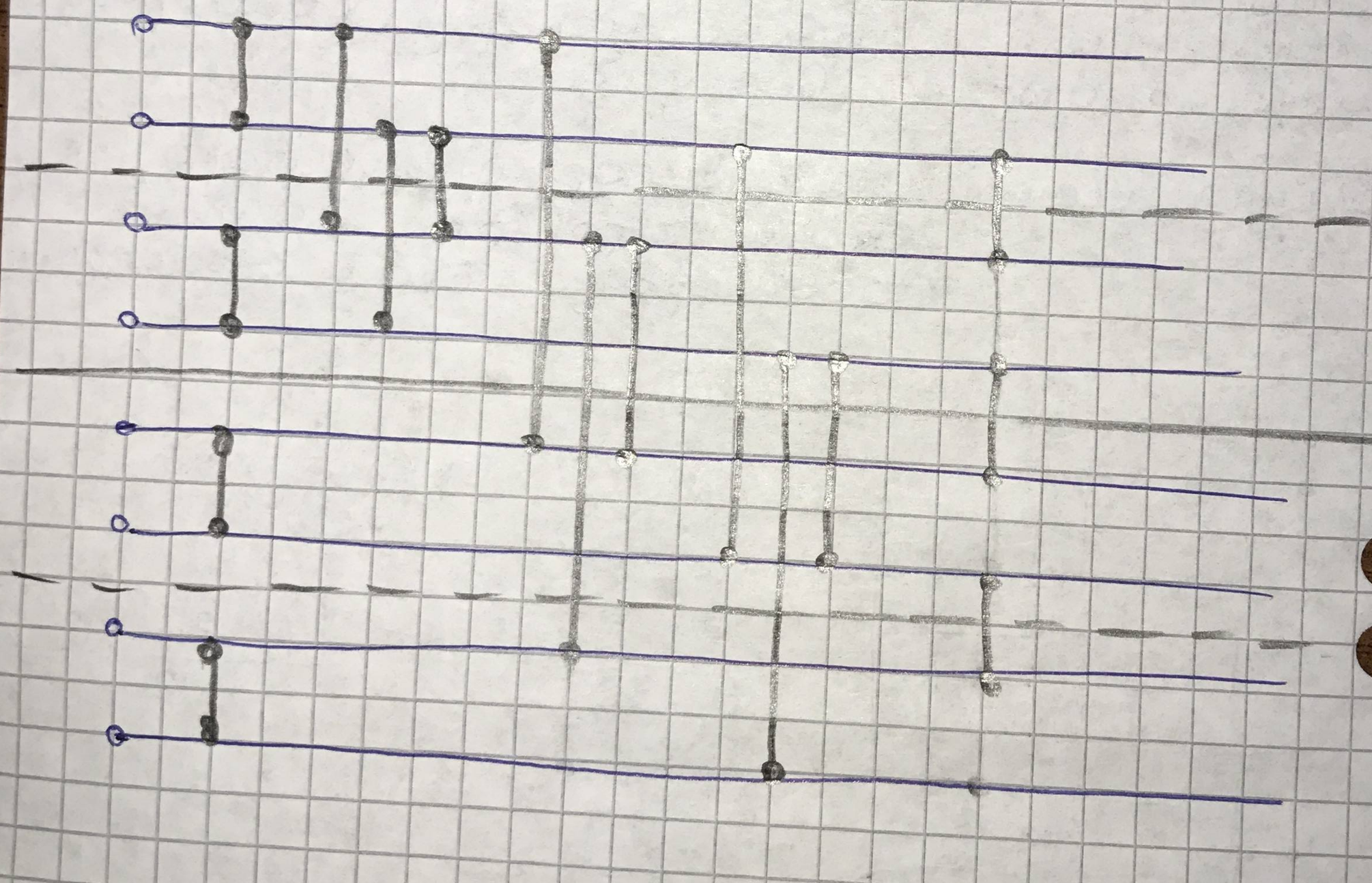
n=7:

обведенные четные и нечетные фрагменты

обведенные четные строки



n=8:



Билет № 25 лист № 1 из 3

Паралл. алгоритмы генер. псевдослуч. чисел. Требования к генераторам псевдослучайных чисел. Лин.-конгруэнтные генераторы. М-последовательности. Достоинства и недостатки

Для организации парал. обр-ки разных экспериментов требуется указать метод формирования на каждом из процессоров уникального набора СЧ.

Два вида генераторов:

1) генераторы случ. чисел, основанные на некот. физ. принципах

2) генераторы псевдослуч. чисел, основанные на детерминир. алгоритмах

Ген-ры случ. чисел:

- формирование прошлого длинных посл-тей СЧ
- обеспечивают формирование на различных пр-рах различных посл-тей
  - не обеспечивают воспроизводимость рез-тов
  - св-ва последовательностей могут меняться в зависимости от окруж. среды
  - не поддаются аппаратному тестированию
  - аппаратно-зависимые

Декер. псевдослуч. чисел:

- длина посл-тей ограничена величиной периода
- на различных пр-рах различные посл-ти
- возможность согласованного формир. на многих пр-рах фрагментов единой последовательности
- могут быть предварительно протестированы

III.0. предварительно надо ответить на вопросы:

должен ли ответ парал. пр-ры в точности совп. с ответом послед.?

НЕТ

ДА

где взять нужное кол-во равных „своих" эл-тов?

каким образом на каждом из пр-ров вычислить знач-я именно „своих" эл-тов, в ко вычисл. знач-я нужны?

физ. генератор? как обеспечить уверен. в сохранении св-в?

брать на пр-ре rank числа rank+P*I как вычислять каждое P-ое число?

на rank числа из диап. (2N/P)*rank... ...(2N/P)(rank-1) Как попасть в нужн диапазона?

Подводя итог, основные требования, предъявл. к генераторам псевдослуч. чисел:

1) Возможность формир. последн достаточно большой длины

2) Возможность определения любого элемента последовательности за короткое время без вычисления всех ее предыдущих членов

Линейно-конгруэнтные генераторы

- $U_{n+1} = (aU_n + c) \bmod m$
- ЛКГ позволяют вычислить одновременно на векторном компьютере с k процессорами очередные k элементов последовательности, используя для каждого O(1) операций; достаточно предварительно один раз вычислить величины

$$a^k \bmod m \text{ и } \left(\frac{a^k - 1}{a-1} c\right) \bmod m$$

$$U_n = \left(a^n U_0 + \left(\frac{a^n - 1}{a-1} c\right)\right) \bmod m \quad (*)$$

$$U_{k(n-1)} = \left(a^k U_{kn} + \frac{a^k - 1}{a-1} c\right) \bmod m$$

Можно показать, что с пом-ю (*) можно вычислить эл-т с номером n за $O(\log n)$ операций.

a^n: $a^n \bmod m = a^{\lceil \frac{n}{2} \rceil + (n - \lceil \frac{n}{2} \rceil)} \bmod m =$

$$= \left(a^{\lceil \frac{n}{2} \rceil} \bmod m * a^{n - \lceil \frac{n}{2} \rceil} \bmod m\right) \bmod m$$

$a^n - 1$: $a^n - 1 = a^t(a^k - 1) + a^t - 1$

$n = k + t \qquad k = \lfloor \frac{n}{2} \rfloor$

$$\left(\frac{a^n - 1}{a-1}\right) c \bmod m = \left[\left(a^t \frac{a^k - 1}{a-1} + \frac{a^t - 1}{a-1}\right) c\right] \bmod m =$$

$$= \left\{\left(\left(a^t\right)_m \left(\frac{a^k - 1}{a-1}\right)_m + \left(\frac{a^t - 1}{a-1}\right)_m\right) c\right\} \bmod m$$

$k = \lfloor \frac{n}{2} \rfloor \qquad t = n - k$

Проблемы 1) малая длина периода ЛСЧ, т.к. для эфф. реализ. опер. умнож-я и слож. по mod m, разрядность модуля огран разрядностью операндов

2) d-мерные точки, сформир. исходной ф-лой, располагаются не более чем в $\sqrt[d]{d! m}$ гиперплоск. размерности d-1.

В частности при $a = 2^{16}+3$, $c=0$ и $m=2^{31}$ в трехмерном случае $d=3$ таких плоскостей не более 16.

Свойство: Если $c=1 \bmod 2$, $a=1 \bmod 4$, $m=2^k \to T=m$

M - последовательности

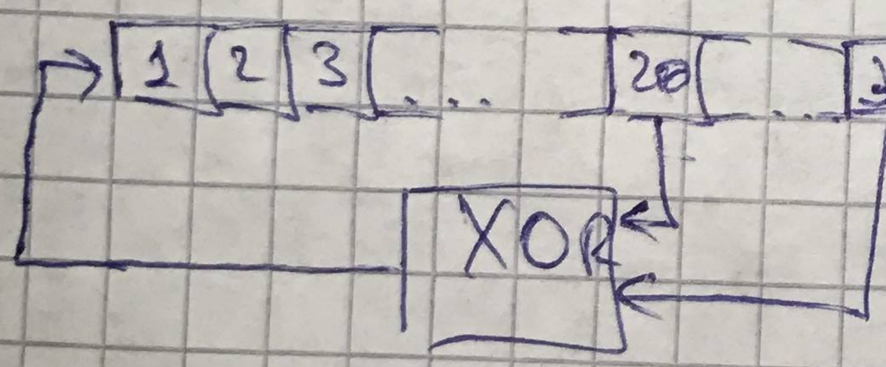
Рекуррентные соотношения порядка r:

$$U_n = a_{n-1}U_{n-1} + a_{n-2}U_{n-2} + \dots + a_{n-r}U_{n-r} \bmod 2$$

$$U_0 = 1, U_1, \dots, U_{r-1} = 0$$

Идея есть сдвиговые регистры; есть XOR; у регистра есть текущее состояние и мы можем посчитать XOR

далее все побитово сдвигаем направо, а в первую ячейку записываем XOR



$$\gamma_i \oplus \gamma_{i+20} + \gamma_{i+33} = 0$$

$$\gamma_{n+r} = \left(\sum_{k=0}^{r-1} e_k \gamma_{n+k} \right) \bmod 2$$

получим все возможные состояния, т.е. последовательность МАКСИМАЛЬНОЙ (M) длины

Билет №25 лист №3 из 3

рассмотрим послед-ть полиномов $t^n \bmod G(t)$

между полиномами и М-послед. есть связь:
оба метода двоичного формирования одни и
те же последовательности

Если $t^n = \sum_{j=0}^{z-1} a_{nj} t^j \bmod (2, G(t))$

то $x_n = \sum_{j=0}^{z-1} a_{nj} x_j \bmod 2$

Генерация эл-та с произвольным номером k:

$f_k = x^k \bmod G$; $k = \sum_{i=0}^{n-1} d_i 2^i$, $d_i \in \{0;1\}$

$$X^k = \prod_{i=0}^{n-1} \left(x^{2^i}\right)^{d_i} \bmod G$$

$$f_k = x^k \bmod (G)$$

Полином должен быть примитивен
Максимальная длина посл-ти только если
полином примитивен

Это определяется набором условий:

$$x^{2^z} = x \bmod (G(x))$$

$$x^{\left(\frac{2^z-1}{n_i}\right)} \neq 1 \bmod G(x)$$

n_i - простые делители

$2^z - 1 = \prod_i n_i$

Опр: $P(x)$ степени $z \geq 1$ примит. если он неприводим $\Rightarrow$

$$x^j \not\equiv 1 \pmod{P(x)} \quad \forall$$

$$j \in \{1, 2, \ldots 2^z - 2\}$$

Это можно быстро проверять, но на ПК искать простые делители очень долго

Но есть помимо с известным списком

Пусть есть регистровый 4-х разрядный счет. начинаем с сост.-я всех единиц. Когда пол-ся совпадет с остатком

$x^i \bmod (x^4+x^3+1) \bmod x$

↙ одна и та же посл-ть

Выводы: 1) Есть возм-ть вычисления за $O(\log k)$ числа с произв. номером k с помощью бинарн пр-я $f_k = x^k \bmod G$

2) При наличии числа с номером k есть возможность быстрого вычисл. числа с номером $k+1$ с пом-ю:

$f_{k+1} = x \cdot f_k \bmod G$.

Билет №26

Декомпозиция расчетных сеток. Критерии декомпозиции. Методы декомпозиции.

Требуется выбрать такую декомпозицию расчетной сетки, при которых вычисл. нагрузка распределена равномерно между процессорами, а накладные расходы, вызванные дублированием вых-ых и передачей данных между проц. мелки

Критерии декомпозиции

В завис-ти от специфики задачи, могут быть предъявлены разные требования

I. Классический критерий

* равномерн. распред. по доменам суммарного веса узлов/ребер

* минимизация макс. веса исходящих из домена ребер

$G^0 = (V, E)$, $V = \{v_i\}$, $|V| = n$. Вершины имеют веса $w(v_i)$, ребра - $w(e_{ij}) = w(v_i, v_j)$

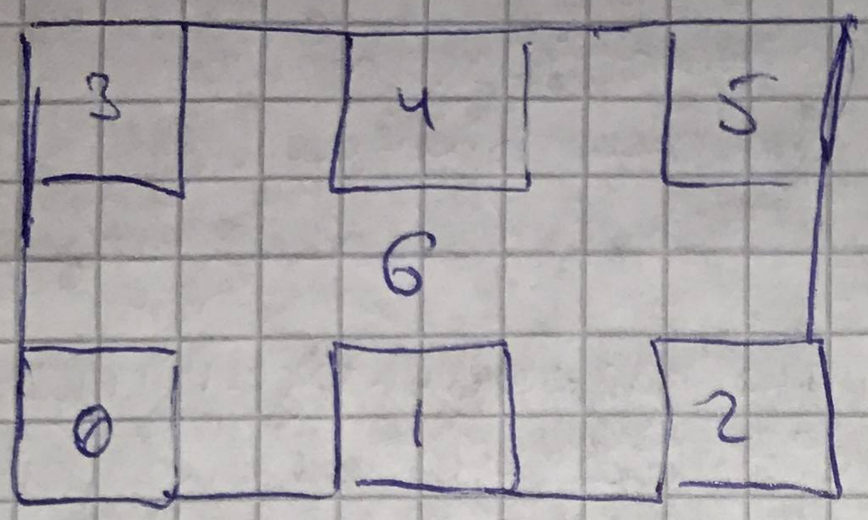
Требуется найти разбиение

$$R(V) = (V_1, \dots V_p)$$

$$V = \bigcup_{k}^{p} V_k \qquad V_i \cap V_j = \varnothing, \; i \neq j$$

$$\min_{R(V)} \left\{ J = \max_{k=1\dots p} \sum_{v_i \in V_k} \left(w(v_i) + \alpha \sum_{v_j \notin V_k} w(v_i, v_j) \right) \right\}$$

II. Выделение обособленных доменов



0-5: среди них нет ни одной пары доменов, соединенных ребром

все выходящие из доменов 0-5 ребра замыкаются на узлы из 6.

В системах с общ. памятью обработка 6 одним проц-ом увеличивает попадания в кэш.
На распред. — нетрадиционные стратегии работ.

III. Минимизация макс. степени домена графа

Обеспечивает снижение актов обмена данными между пр-ми на t шаге по времени

IV. Обеспечение связности графов каждого из доменов

Уменьшает фрагментированность доменов.

Методы декомпозиции

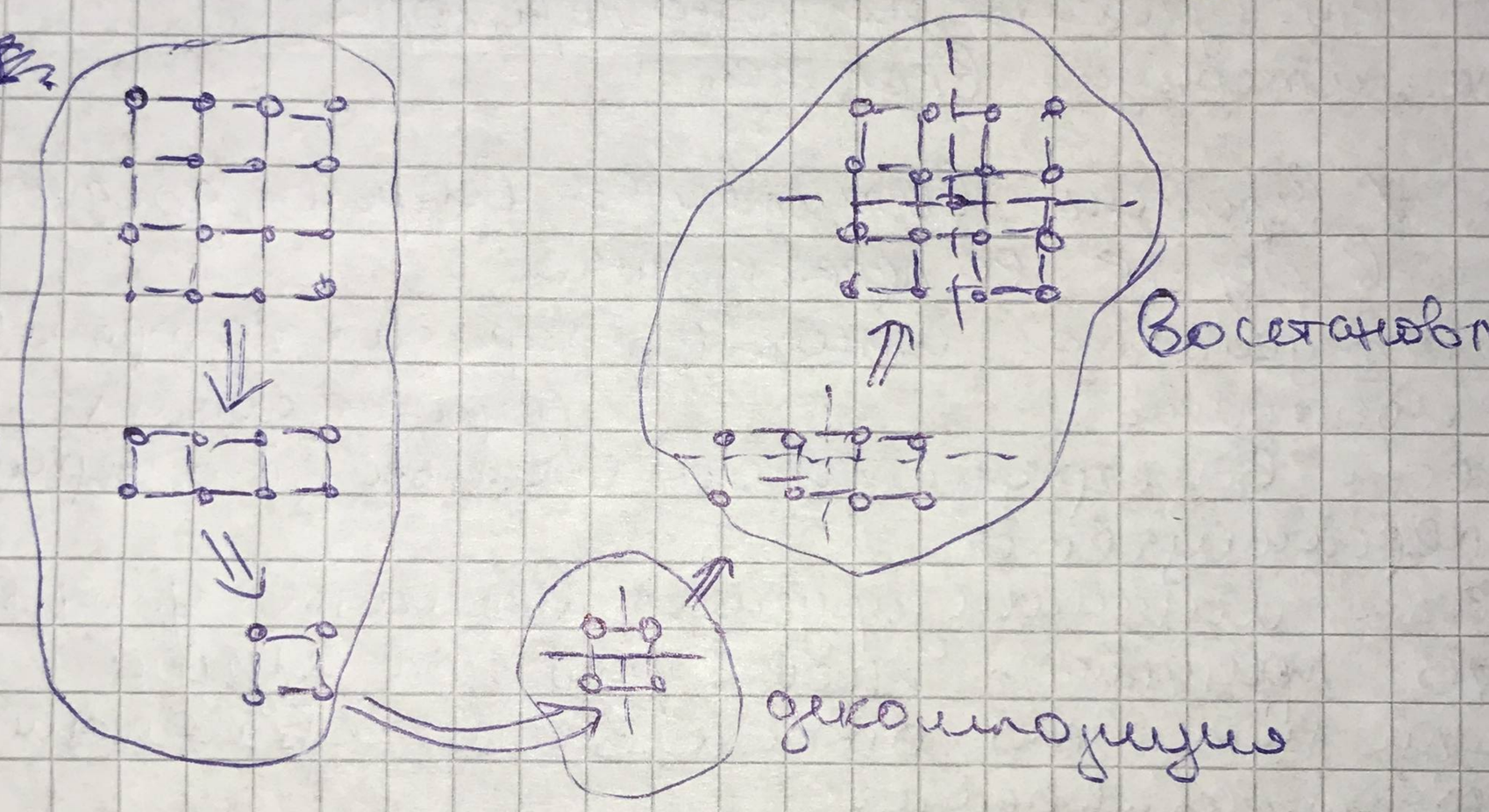
1) Иерархические (см. Билет № 27)
2) Рекурсивная бисекция (см. Билет № 28)
3) Инкрементный алгоритм (см. Билет № 29)

Билет №27

Иерархические алгоритмы разбиения графов. Локальное уточнение

Основным методом декомп. больших сетей является иерарх. подход, предполагающий огрубление сетки, декомпозицию огруб. образа и последующее уточнение решения на основе локальных крит-ев.

Этапы иерархического метода:



I. Огрубление графа

Итерационное стягивание ребер
В итерации выбирается множество ребер покрытия. С пом-ю этих ребер 2 вершины стягиваются одна

следует отдавать предпочтение ребрам, инцид. вершинам, имеющим наимень. веса

II. Декомпозиция огрубленного графа

Два способа: 1) сразу на нужное число доменов
2) рекурсивно

можно исп. практически любой алгоритм декомпозиц.

III. Восстановление графа

Порождающие вершины распред. в домены с теми же номерами, что и обрабатываемые ими вершины

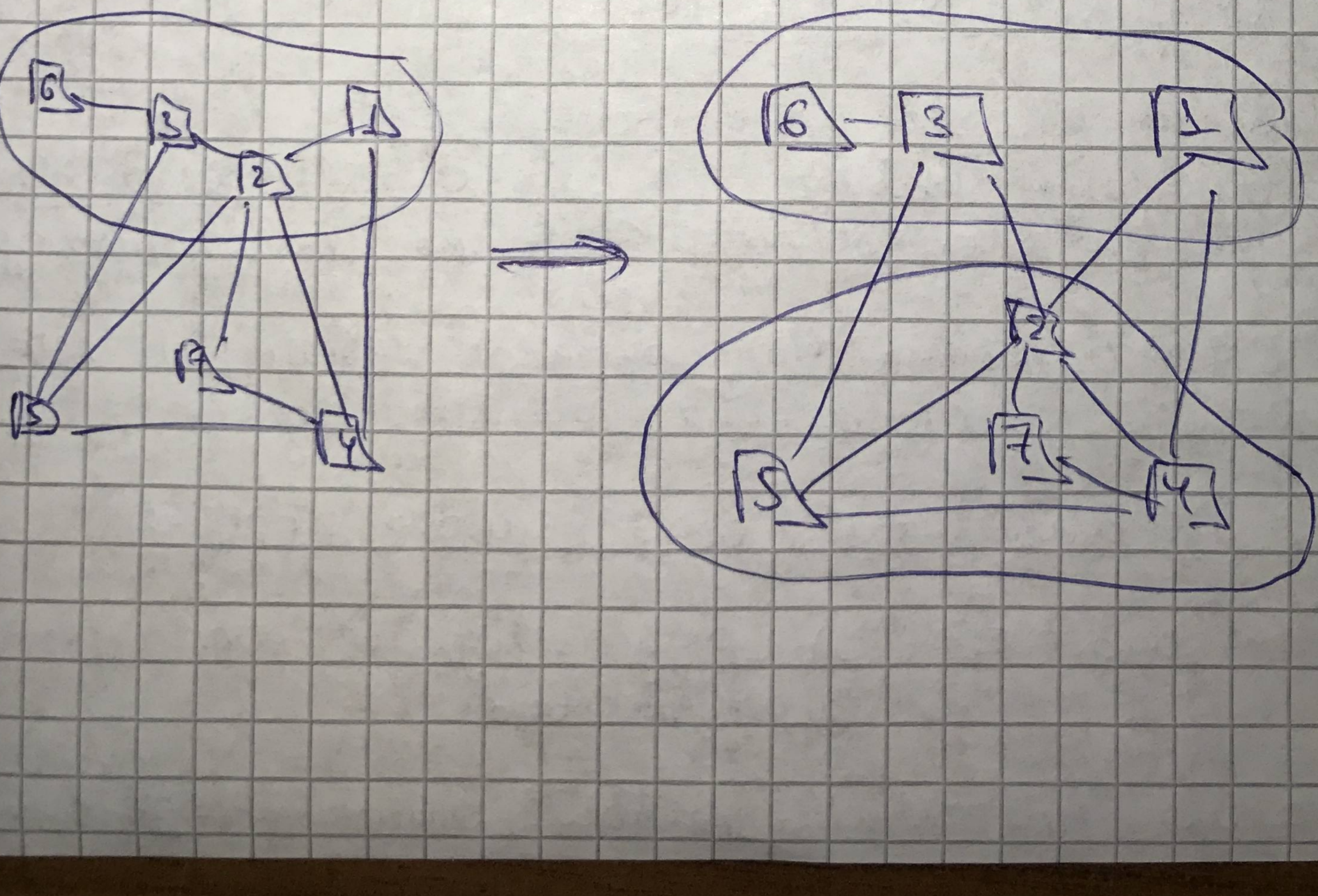
Локальное уточнение границ доменов

Позволяет выровнять между собой веса доменов и уменьшить число ребер, пересек. границы.

Позволяет избежать полного перебора

Каждая итерация включает:

* ∀ вершина вычисляется выгода, достигаемая в рез-те ее перемещения
* определяется множ-во пар (вершина, домен), для которых выгода на любой положительный
* если выгода положительная, то вершина перемещается
* если минимизируемая величина уменьшилась, то принимается новое разб. как лучшее
* если нет, то откатываемся назад и останавливаемся



Билет №28 лист №1 из 2

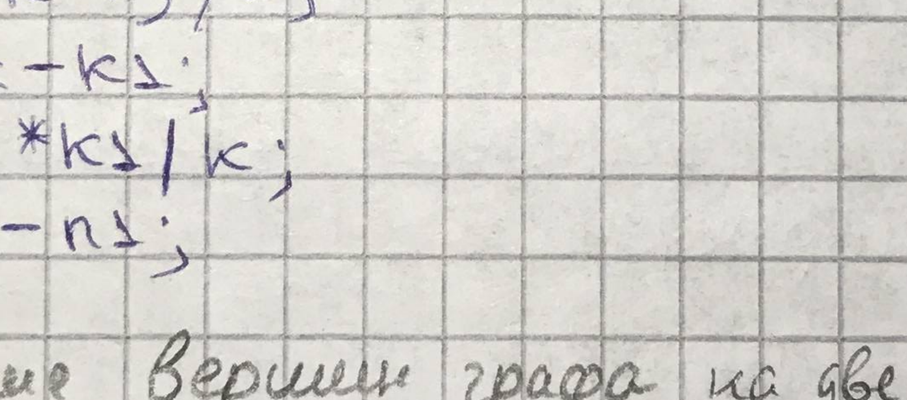
Метод спектральной бисекции графов

При решении задачи декомпозиции графов широко используется рекурсивная бисекция. В соответствии с ней, сетка последовательно, с помощью k-1 шагов, делится на k частей. На первом шаге сетка делится на 2 части. Далее, на каждом шаге одна из полученных ранее частей снова делится на две.

Независимо от того, какой именно алгоритм бисекции используется, следует определиться с процедурой вычисления желаемого числа вершин в каждой из формируемых частей графа.

Для разбиения графа, содержащего n вершин, на произвольное число доменов k может быть использован следующий алгоритм.

```
RecursiveBisect(graf, n, k){
    k1 = (k+1)/2;
    k2 = k-k1;
    n1 = n*k1/k;
    n2 = n-n1;
    // разбиение вершин графа на две части размером n1 и n2
    Bisect(graf, n, subgraf1, n1, subgraf2, n2);
    if (k1 > 1) RecursiveBisect(subgraph1, n1, k1);
    if (k2 > 1) RecursiveBisect(subgraf2, n2, k2);
}
```



Спектральная бисекция –

– это метод разбиения графа на два домена в заданном отношении весов, минимизирующий суммарный вес ребер, соединяющих вершины у разных доменов. Основан на использ. спектральной матрицы графа (матрицы Лапласа)

Определим на множестве вершин графа сеточную функцию q. Значение q_i, соотв. вершине i, может принимать -1 или 1. Будем считать, что вершины, в кот. $q = -1$, принадлежат одному домену, а остальные – другому.

Очевидно, что число разрезанных (соединяющих вершины у разных доменов) ребер:

$$|E_c| = \frac{1}{4} \sum_{i,j : e_{ij} \in E} (q_i - q_j)^2$$

e_{ij} – ребро, соед. вершины i и j. Именно величину $|E_c|$ надо минимизировать, но при доп. условии, обеспечивающем одинак. число вершин в обоих доменах:

$$\sum_{i=1}^{|V|} q_i = 0$$

Для замыкания системы добавим условие нормировки

$$\sum_{i=1}^{|V|} q_i^2 = q^T q = |V|$$

Спектральная бисекция состоит из этапов A–D.

Этап А: Построение спектральной матрицы графа

- Недиагональный эл-т матрицы l_{ij}, $i \neq j$, равен -1, если между i и j есть ребро. Диагональный эл-т – степень вершины i. Остальные – ∞.

$$l_{ij} = \begin{cases} -1, & e_{ij} \in E, \ i \neq j \\ \sum_{k \neq i} l_{ik}, & i = j \\ \infty, & \text{иначе} \end{cases}$$

- Этап В: определение первого собств. в-ра матр. Лапласа
- Решение системы $Lq = \lambda q$ дает собственные знач-я спектральной матрицы.

Преобр. минимизируемую квадратичную форму.

$$|E_c| = \frac{1}{4} q^T L q = \frac{1}{4} q^T \lambda q = \frac{1}{4} \lambda |V|$$

$$\Downarrow$$

надо найти с.в. ψ, соотв. миним. ненулевому соб. числу спектр. матрицы $L\psi = \lambda_1 \psi$; $e\psi = 0$

$\Rightarrow$ задача целочисленного программирования

Этап С Упорядочивание вершин графа

- Упорядочим номера вершин по возрастанию значений соответствующих им элементов вектора ψ. В рез-те получим новый порядок вершин. (вектор s)

Этап D Формирование двух доменов

Используя вектор s можно распределить вершины на две части в требуемой пропорции

- $Q = \frac{n_1}{n_1 + n_2}$, n_1, n_2 – числа вершин во фрагментах
- Первая часть содержит вершины, имеющие меньшие значения компонент наименьшего с.в., вторая – большие.

Билет №29

Инкрементный алгоритм декомпозиции графов

Включает следующие этапы:

1) инициализация доменов – определение p случайно выбранных вершин

2) распределение вершин по доменам методом инкрементного роста

3) локальное уточнение границ сформированных доменов

4) анализ свойств ядер сформированных доменов и окончание работы, если в ∀ домене достигнута свойство ядер заданного уровня.

5) перенос части закрепленных за доменами вершин в группу свободных вершин. переход к этапу 2.

Пусть D_i^k – множества вершин, принадлежащих доменам i ($i=1,\dots,p$) на шаге k. Вначале все вершины свободны: $D_0^0 = V$.

На этапе 1 случайно выбранные p вершин изымаются из группы свободных вершин, и каждая из них включается в отдельный домен D_i. $|D_i^1| = 1$.

На этапе 2: 1) присоединение к каждому из доменов прилегающих своб. вершин

2) дифузное перекрытие. вершин между границами доменов с целью выравнивания числа вершин

На этапе 3 - традиционный алгоритм локального уточнения

На этапе 5 выполняется частичное сокращение рудиментов доменов за счет перекоса вершин нескольких внешних слоев в множество свободных вершин

Инкрементный алгоритм, сохраняя высокое качество разбиения, позволяет уменьшить компактность доменов.

Билет №30 лист №1 из 3

Адаптивное интегрирование. Метод локального стека. Метод глобального стека

Существует множество задач, при решении которых элементарные задания порождаются динамически

Постановка задачи

Вычислить с точностью ε значение опр. инт

$$J(A,B) = \int_A^B f(x)dx$$

Пусть на $[A,B]$ задана равномерная сетка содержащая $n+1$ узел $x_i = A + \frac{B-A}{n} i,\ i=\overline{0,n}$

Тогда согласно методу трапеций

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

Будем полагать, что J найдено с точностью ε, если выполнено:

$$|J_{n_1} - J_{n_2}| \le \varepsilon |J_{n_2}| \qquad n_2 > n_1$$

Последовательный алгоритм

```
Integr Trap1 (A, B) {
  n=1;
  J2n = (f(A)+f(B))(B-A)/2;
  do {
    Jn = J2n; n=2n; S = f(A)+f(B);
    for (i=1; i<n; i++) S+= 2f(A+(B-A)i/n);
    J2n = S(B-A)/n;
  } while (|J2n - Jn| ≥ ε J2n)
} return J2n
```

Недостатки – в некот. т-ках знач-е подынт. ф-и вычисл. более одного раза
– на всем интервале интегр-я используется равн. сетка, тогда как число узлов сетки на единицу длины зависит от функции

Адаптивный алгоритм

```
main() { J = IntTrap3 (A, B, f(A), f(B)) }

IntTrap3 (A, B, fA, fB) {
    J=0; C=(A+B)/2; fC=f(C);
    SAB = (fA+fB) * (B-A)/2;
    SAC = (fA+fC) * (C-A)/2;
    SCB = (fC+fB) * (B-C)/2;
    SACB = SAC + SCB;

    if (|SAB - SACB| ≥ ε |SACB|)
        J = IntTrap3 (A, C, fA, fC) + IntTrap3 (C, B, fC, fB)
    else
        J = SACB
    return J;
}
```

Преимущества: + нет поворотных выч-ий функции
+ малое число вычислений на гладких участках

Недостаток: – координаты концов отрезков хранятся в программном стеке процесса и фактически недоступны программисту

```
Integral04(A,B){
    J=0; fA=f(A); fB=f(B);
    SAB=(fA+fB)*(B-A)/2;
    while(1){
        #обработка отрезка
    }
    return J;
}

    C=(A+B)/2;
    fC=f(C);
    SAC=(fA+fC)*(C-A)/2;
    SCB=(fC+fB)*(B-C)/2;
    SACB=SAC+SCB;
    if(|SAB-SACB|>=E|SACB|){
        PUT-INTO-STACK(A,C,fA,fC,SAC);
        A=C; fA=fC; SAB=SCB;
    }else{
        J+=SACB;
        if(STACK-IS-FREE) break;
        GET-FROM-STACK(A,B,fA,fB,SAB)
    }

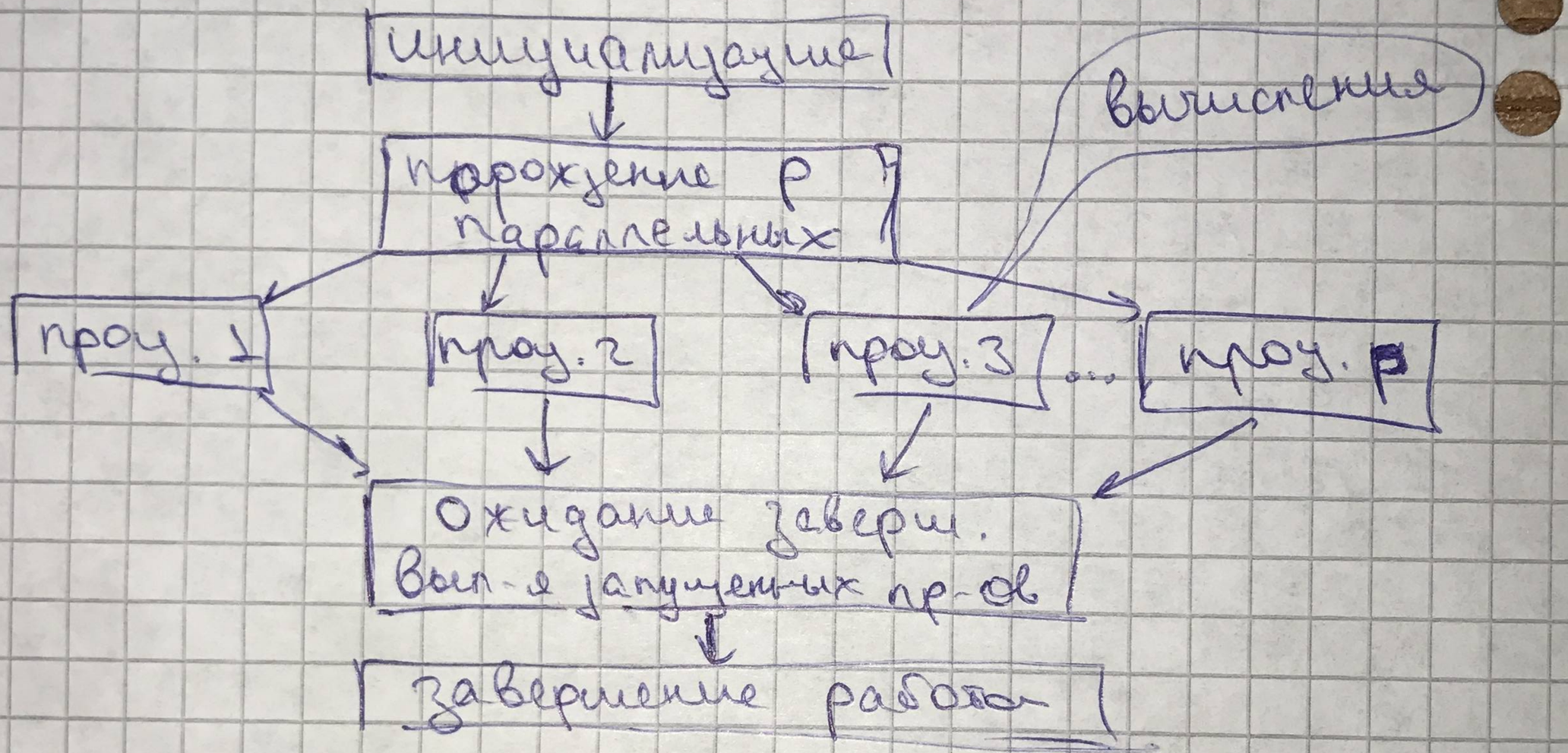
sp=0;
struct{A,B,fA,fB,S}stk[1000];
#define STACK-IS-FREE (sp==0)
#define PUT-INTO-STACK(A,B,fA,fB,S){stk[sp].A=A
                                    ... sp++;}
#define GET-FROM-STACK(A){sp--; A=stk[sp].A ... J;
```

Теперь хотим получить параллельный алгоритм.

Методы геометр. паралл. и коллективного реш-я практически не применимы к данной задаче (дисбаланс, $n = ?$)

Метод глобального стека

Общая память + треды + семафоры
Не требует управляющего процесса



Пусть есть доступный всем паралл. процессам список отрезков интегрирования, организованный в виде стека. Назовем его глобальным стеком. Сначала туда кладутся концы.

Каждый процесс:
пока в глоб. стеке есть отрезки:
1) взять один отрезок из глоб. стека
2) выполнить алгоритм локального стека, дополнительно выполняя: если в лок. стеке есть несколько отр-ков, а в глоб-стеке отрезки отсутствуют, то переместить часть в глоб.
3) по исчерпании лок. стека добавить полученную част. сумму к общ. и мах 1.

Билет №30 лист №3 из 3

Вопросы:

1) должен ли закончить работу процесс, если глоб. стек пуст?
2) сколько переменных из лок. в глоб.?
3) как обесп. запись откуков в стек?
4) в какой момент заверш. несколько проц. работу?

Ответы: 1) нет: сначала там только 1 откуок, который заберет 1 проц
3) семафоры

Окончательный алгоритм sdat - глобальные

```
int slave_thr() {
    sp=0; s=0;
    while (1) {
        P(sdat.sem_task_present);
        // чтение одного эл-та из списка
        P(sdat.sem_list); // здесь семафор, под. сверх
        sdat.ntask--;
        GET_OF_GLOBAL_STACK[sdat.ntask](a,b,fa,fb,sab);
        if (sdat.ntask) V(&sdat.sem_task_present);
        if (a<=b) sdat.nactive++;
        V(sdat.sem_list)
        if (a>b) break;
        while (1) { c=(a+b)/2; fc=func(c);
            sac=(fa+fc)*(c-a)/2;
            sab=(fc+fb)*(b-c)/2;
            if (!BreakCond(sacb,sab)) {
                s+=sacb; if (!sp) break;
                sp--; GET_FROM_LOC_STACK[sp]
                    (a,b,fa,fc,sab)
            } else {
```

```
        PUT_TO_LOCAL_STACK[sp](a,c,fa,fc,sac)
        sp++; a=c; fa=fc; sac=scc;
    }

if (sp>SPK) && (!sdat.ntask)) {
    P(sdat.sem_list)
    if (!sdat.ntask) {
        V(sdat.sem_task_present);
    }
    while ((sp>1) &&(sdat.ntask<sdat.maxtask)
    { sp--; GET_FROM_LOCAL[sp](a,b,...)
        PUT_TO_GLOBAL[sdat.ntask](a,b,...)
        sdat.ntask++;
    }
}
P(&sdat.sem_list); sdat.nactive--;
if ((!sdat.nactive) && (!sdat.ntask)) {
    for (i=0; i<nproc; i++) {
        PUT_TO_GLOBAL_STACK[sdat.ntask](2,1,...)
        sdat.ntask++;
    }
    V(sdat.sem_task_present)
}
V(sdat.sem_list)
}
P(&(sdat.sem_sum))
sdat.S_all += S;
V(&(sdat.sem_sum));
}
```

Билет №31

Визуализация сеток и сеточных данных

- Причины, по кот. необходимо привлекать удаленную параллельную ВС для визуал. эксп.
  1) недостаточная вычт. мощ. ПК
  2) пропускная способность сети не дает передавать все данные от СК к ПК
  3) ограничен. объем ОП ПК
  4) ограниченность ФС ПК

⇓

существ. часть операций для визуализации - на СК

Клиент-серверная технология

- серверная часть системы, выполн. на СК, обеспечивает обработку больш. данных
- клиентская часть - на ПК, дает интерфейс и непосредств. отображение

Online vs Offline визуализация

- Online - визуализация данных непосредственно при проведении расчета, используя ОП
  - плюсы: + : экономия дискового пр-ва и времени на чтение/запись
    + : возможность активного влияния на процесс
  - проблемы: а) большие требования к ОП
    б) затруднены возможности повторного изучения данных
    в) ограничены возможности визуализации динамики процесса
    г) огр. возм-ть виз-и заданных моментов
    д) неоптимируемость ухудшается (время балансировка процессоров

е) трудности с коллективным доступом к СК

Offline – выделение численного моделирования и визуализации в отдельные программы, взаимодействующие через ФС.

Существенный минус: надо периодически писать на диск, что долго.

Пусть исходные данные – сеточные. Скалярные сеточные ф-и представляются массивами, в каждом эл-те кот. хранятся знач-я иссл. величин. Векторные могут быть предст. наборов скалярных.

Наиболее затратные с точки зрения времени этапы

I. Моделирование: а) декомпоз. сетки
б) запись структуры сетки
в) запись фрагментов ~~сетки~~ сетки
г) запись фрагментов сеточной ф-и

II. Сервер визуализ.: д) чтение структуры сетки
е) декомпоз. сетки, пересыл. фрагментов на процессоры
ж) чтение фрагментов сетки
з) -//- сеточной ф-и
и) формир-е данных, описыв. вирт. сцену

III. Клиент визуализации: к) прием данных от сервера
л) отображение
м) накапливание эффектов

Для поверхностей: 1) аппроксимация
2) триангуляция
3) загрубление

Билет №32

Обработка и ввод-вывод больш. объемов сеточных данных, предн. для визуализации. Двухуровневое представление сеточных данных

Большой объем данных предполагает использование множества пр-ров для фильтрации нужных данных ⇒ необходим метод их рационального распределения. Умеется использовать двухуровневую стратегию хранения и обр. сеток.

- Расчетная сетка предварительно однократно разбивается на множество блоков небольшого размера - микроэлементов. В дальнейшем сетка хранится в виде наборов микроэлементов и графа, определяющего их взаимосвязи - макрографа.

+: В макрогр. меньше вершин ⇒ быстрее разбиение
+: можно хранить распределенно

Распределенный ввод-вывод данных

- Вершины, объединенные микроэлементом, сохраняются на носителе как единое целое. На этапе визуализации извлекаются на проц-р и читаются сразу вся группа.

+: быстрее чтение данных
+: независимое параллельное чтение
+-: при визуализации возможно лишнего данных

Огрубление и сжатие скал. сеточных ф-й

Стандартные методы сжатия без потерь неэффект. применительно к вещ. числам

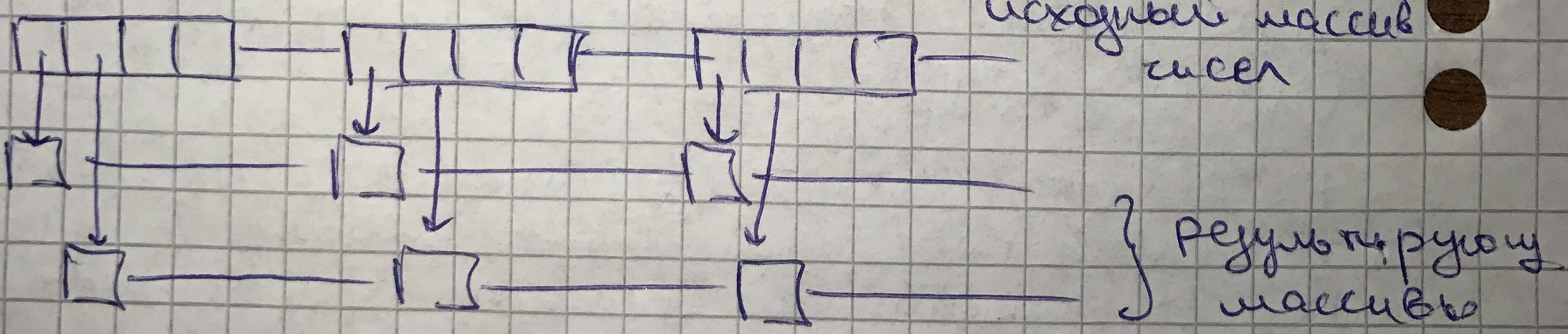
Можно игнорировать часть младших разрядов мантиссы ⇒ увеличение к-та компрессии

Двухур. сис-ма позволяет задать степени огрубления отдельно для каждого блока

Дополнительно увеличить к-т компрессии можно перегруппировкой разрядов вещ. чисел:

Идея: соседние в массиве числа с большой вер-ю имеют одинак. порядок и знак. ⇒ Выделим биты, отвечающие за сохр. порядка и знака в один массив, а остальные - в другой

Двоичн. предст. вещ. числа занимает 4 байта, что позволяет рассматривать одномерный массив чисел как двумерный массив байтов. Переколлекции массивы по старшинству байтов. Младшие байты - в первый, следующие - во второй и т.д. Такое представление будет лучше сжато.



Билет №33

Параллельные алгоритмы многоэкстрем. оптимизации, использующие кривые Пеано

Постановка задачи:

$$\varphi = \varphi(y^*) = \min\{\varphi(y) : y \in D\}$$

$$D = \{y \in R^N : a_i \le y_i \le b_i,\ 1 \le i \le N\}$$

где целевая ф-я $\varphi(y)$ удовл. условию Липшица с к. L:

$$|\varphi(y_1) - \varphi(y_2)| \le L|y_1 - y_2|,\ y_1, y_2 \in D$$

Используя кривые типа развертки Пеано $y(x)$, однозначно отображающие отрезок $[0;1]$ на N-мерный гиперкуб D

$$D = \{y \in R^N : -2^{-1} \le y_i \le 2^{-1},\ 1 \le i \le N\} = \{y(x) : 0 \le x \le 1\}$$

исходную задачу можно редуцировать к след. одномерной задаче:

$$\varphi(y(x^*)) = \min\{\varphi(y(x)) : x \in [0,1],\ g_j(y(x)) \le 0;\ 1 \le j \le m\}$$

Рассматриваемая схема редукции размерности сопоставляет многомерной задаче с липшицевой мин. ф-ей одномерную задачу, в которой целев. ф-я удовл. равном. усл-ю Гельдера:

$$|\varphi(y(x')) - \varphi(y(x''))| \le K|x' - x''|^{1/N},\ x', x'' \in [0,1]$$

N - размерность исх. задачи; $K \le 4L\sqrt{N}$

Индексный метод глобальной оптимизации

Редукция размерности

При исп-и отобр-й типа крив. Пеано близким в N-мерном пр-ве образам y' и y'' могут соотв достаточно далекие x', x'' на отрезке $[0;1]$.

Сохранить часть инф-и о близости т-к двумерн исх-е множество отображений

$$Y_L(x) = \{y^1(x) \dots y^L(x)\}$$ вместо одной кривой Пеано

$\forall y^i(x)$ м.б. получена в рез-те повороте развертки вокруг начала к-т. Макс. число различных в N-мерном кубе – 2^N. Использование всех возможных используется $N(N-1)$ (поворот $\pm \pi/2$)

Параллельный индексный метод и локально-глобальная оптимизация

Множ-во отобр. $Y_L(x) \Rightarrow$ множ-во одномерн. многомерных задач

$$\min \{\varphi(y^l(x)) : x \in [0;1], 1 \le l \le L\}$$

Каждая может решаться независимо на отд. процессоре

Для организации взаимодействия на N процессоре создается L очередей, в которых процессоры помещают информацию о вычислениях итерациях.

Билет №34 лист №1 из 2

Отказоустойчивые алгоритмы для многопроц. вычисл. систем. Локальные контрольные точки. Локальное парирование ошибок для гиперболических систем ур-й.

Не можем просто взять и передублировать всю задачу.

- Как создать/придумать алгоритмы?

Суперкомп. должен:
- отказоустойчивость (позволяет) не ~~должна~~ тревожить пользователя больше раза в неделю
- стек ПО, отвечающий потребностям широкого спектра приложений и рабочих нагрузок

С точки зрения алгоритмов:

- явные схемы позволяют создать логически простые алгоритмы, но они имеют ограничен. на дискретизацию по времени из условий устойчивости. Хорошо распараллеливаются. Уменьшили шаг по расстоянию → надо сильно уменьшить шаг по времени
- неявные схемы – такой тип, который решать систему ур-й, обращать матрицу – тяжело
- промежуточная: использует преимущества явной и не требует сокращения шагов по времени

Прогноз: на экзафлопсных системах откажет каждые 30 минут

- Уменьшение размера транзистора делает его менее устойч.
- меньший заряд легче украсть

ПО становится сложнее и содержит больше ошибок

Надо записывать контрольные точки, например, на общую ФС каждые 10 минут состояние расчета.

Если ГРО, можем начать считать с контр.т-ки

НО! создать контр. точку - 20~30 минут!

Есть 2 уровня для записи КТ:

1) системный уровень: можем отдать проблему системе и не беспокоиться

2) уровень пользователя: сами берем в свои руки
   - радикальное сокращ. объемов КТ
   - вместо рестарта всей системы - замена вычислительного узла
   - хранение данных не только на лок. НДП, но и в оперативной памяти

Существуют автоматические и полуавтоматические реализации КТ.

ULFM - User Level Failure Mitigation - дополнительные возможности для MPI, позволяет работать на системе, в которой узлы могут не отвечать (некоторые)

Минимальный функционал, сам не решает никаких проблем

Вопрос: как сделать так, чтобы время между откказами было больше чем время, требуемое на восстановление

Билет №34 лист №2 из 2

Можно перераспределять нагрузку с отказавшего процессора, но это очень сложно и не факт что получится эффективно

Более реальный способ: иметь запасные процессоры

Процессор вышел из строя ⇒ заменяем. Но где-то должны быть данные

Хотим тратить максимально мало времени на успись данных ⇒ разумно писать в ОП на соседние вычислительные узлы тору сеть ⇒ уже быстрее, чем за 30 минут

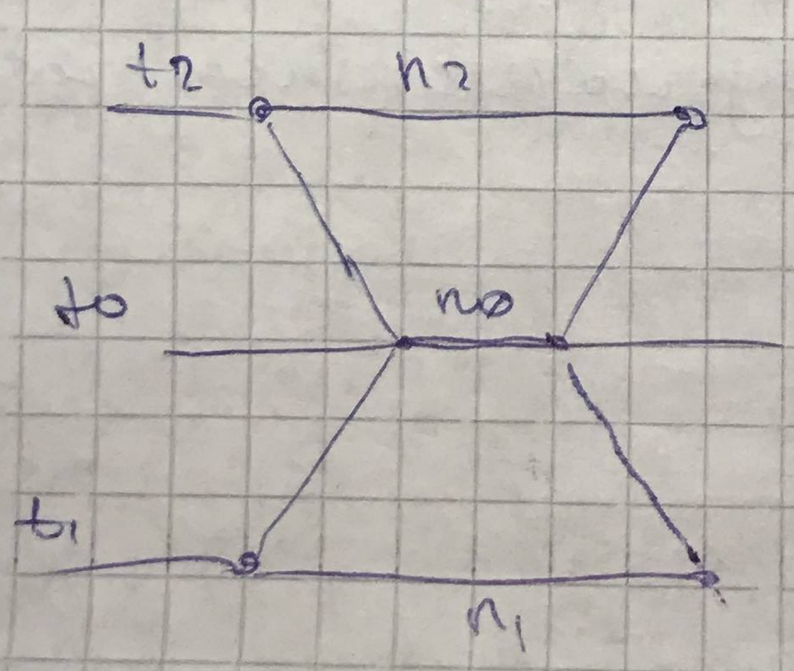
Это работает для любых задач

Будем рус задачи: Одномерное гиперболическое уравнение

$$\frac{\partial^2 \Phi}{\partial x^2} - \frac{1}{c^2}\frac{\partial^2 \Phi}{\partial t^2} = F(x,t)$$

Появляются две характеристики $x-ct$ и $x+ct$, определяющие обл-ть влияния на реш-е $\Phi(x,t)$ в (x,t)

Есть несколько пр-ров, каждый считает какую-т зону. Если процессор вышел из строя:



t_0 – время выхода из строя

t_1 – когда сохранили КТ

на каких-то процах содержится инф. с указанием

Надо умно считать в то по чувствительному

Влияние оценено ⇒ знаем, какие области влияют

Идея: взять инф-у, заново посчитать. Но пока будем пересчитывать, остальные пр-ры убегут вперед. Запасных процессоров должно быть больше.

Вопрос - что лучше? пересчитывать или хранить?

Сколько нужно доп. процессоров?

α - параметр, связывает отн. кол-ва узлов сетки на
k_1 - кол-во шагов после контр. точки проц.

$p_{1D} \geq 2\alpha$ (одномерн. случ.)

$p_{2D} \geq \frac{2}{3}(1+\alpha+\alpha^2)$

$p_{3D} \geq \frac{1}{2}(1+\alpha+\alpha^2+\alpha^3)$

$\gamma = \frac{c\Delta t}{h}$ - число Куранта характеризует

$\alpha = 1 + 2\gamma\frac{k_1}{k_0}$

Итого: 1) необх. поддержка реурс. алгоритмов и прикладного ПО, адаптир. к арх-ре с параллелизмом

2) Их создание обеспеч. рост во всех областях

3) Инструмент - фундаментальное исследование