Курс «Пршраммирование параллельных процессов (PARALLEL COMPUTING)

-

Задачи курса

Параллельное программирование – путь к созданию приложений, направленных на обработку больших объемов данных

Применение в направлениях:

data mining, recommender systems, scientific computation, financial modeling and multimedia processing,…

Цель курса :

- понимать проблемы, возникающие при распараллеливании алгоритмов, и решать их,
- научиться распараллеливать программы и алгоритмы для повышения эффективности обработки данных.

Структура курса

1 Схемы параллельных взаимодействующих процессов. Моделирование с и анализ взаимодействующих процессов

2 Эффективные вычислительные алгоритмы

3 Open MP, MPI и Open CL – широко используемые стандарты разработки параллельных программ

4 Многопоточность. Синхронизация параллельных процессов и классические задачи синхронизации. Объекты ядра ОС

5 Распределенные взаимодействующие процессы

6 Паттерны параллельного программирования

Зачем нам нужно параллельное программирование?

Моделирование образования белка: нужно 10**25 операций, что займет на одноядерном ПК с тактовой частотой 3.2 Ghz 10**6 веков

Прогноз погоды в масштабах всей планеты (ячейки 1 миля до высоты 10 миль, шаг моделирования 1 минута для получения прогноза на 10 дней потребуется 10**16 операций СПТ, что составит 10 дней

Зачем нам нужно параллельное программирование?

Моделирование образования белка: нужно 10**25 операций, что займет на одноядерном ПК с тактовой частотой 3.2 Ghz 10**6 веков

Прогноз погоды в масштабах всей планеты (ячейки 1 миля до высоты 10 миль, шаг моделирования 1 минута для получения прогноза на 10 дней потребуется 10**16 операций СПТ, что составит 10 дней

Зачем нам нужно параллельное программирование?

Моделирование образования белка: нужно 10**25 операций, что займет на одноядерном ПК с тактовой частотой 3.2 Ghz 10**6 веков

Прогноз погоды в масштабах всей планеты (ячейки 1 миля до высоты 10 миль, шаг моделирования 1 минута для получения прогноза на 10 дней потребуется 10**16 операций СПТ, что составит 10 дней

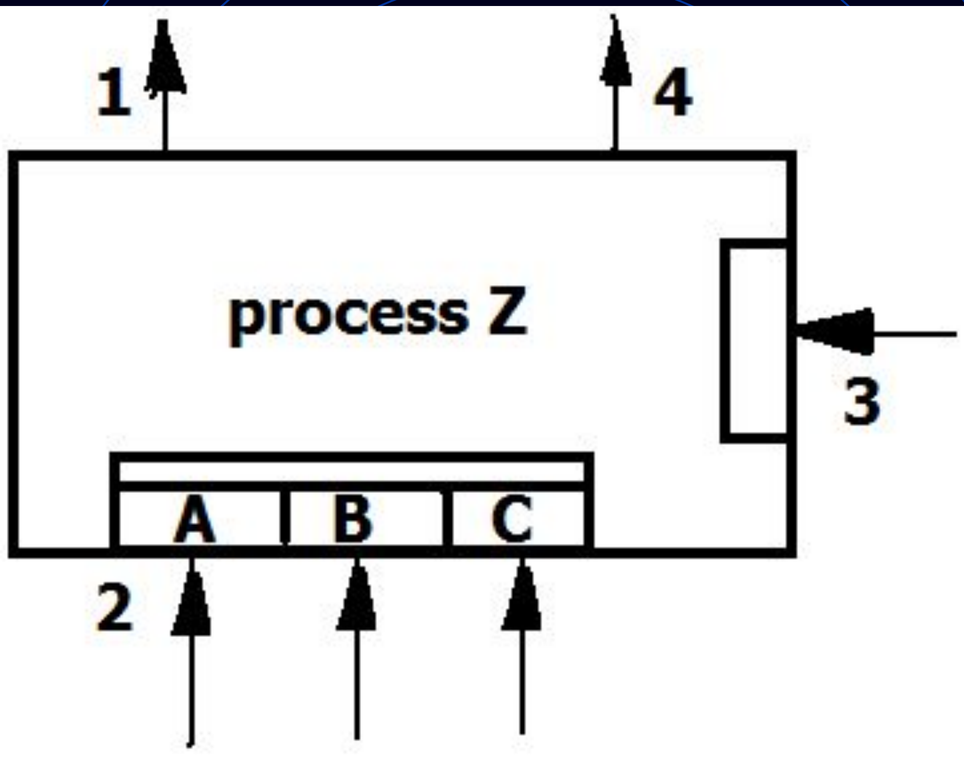
Зачем нам нужно параллельное программирование?

Дейкстра :

„To put it quite bluntly: as long as there were no machines, programming
was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.“

Схемы
взаимодействующих
процессов
-

Процесс

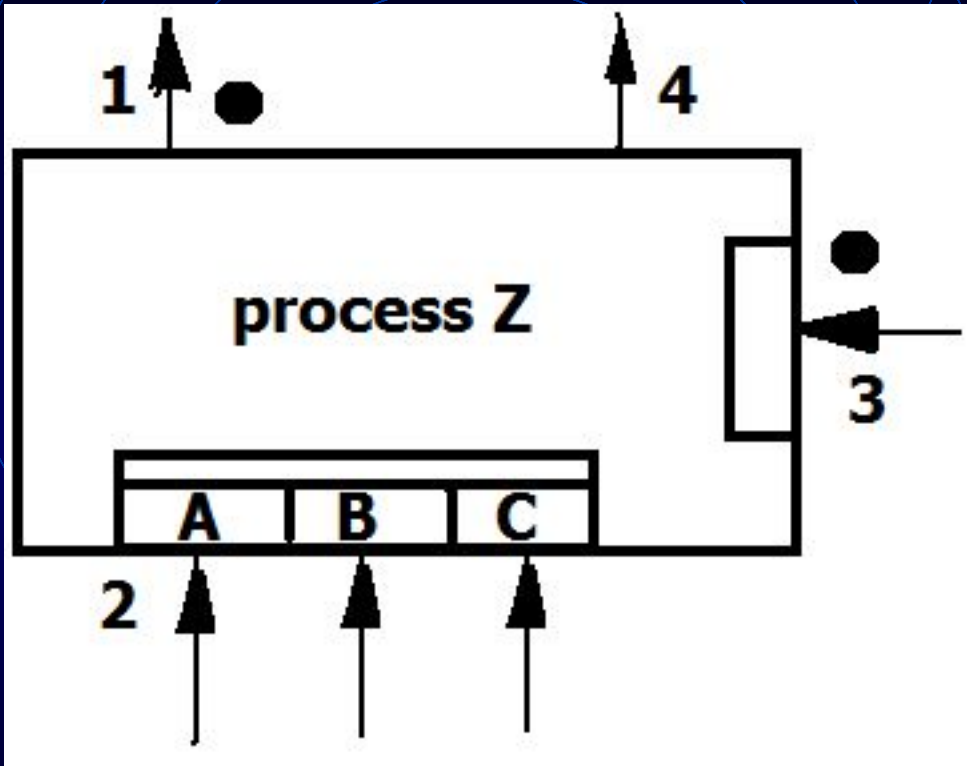


1, 4 - выполняет взаимодействие

2, 3 – входы (принимает взаимодействие)

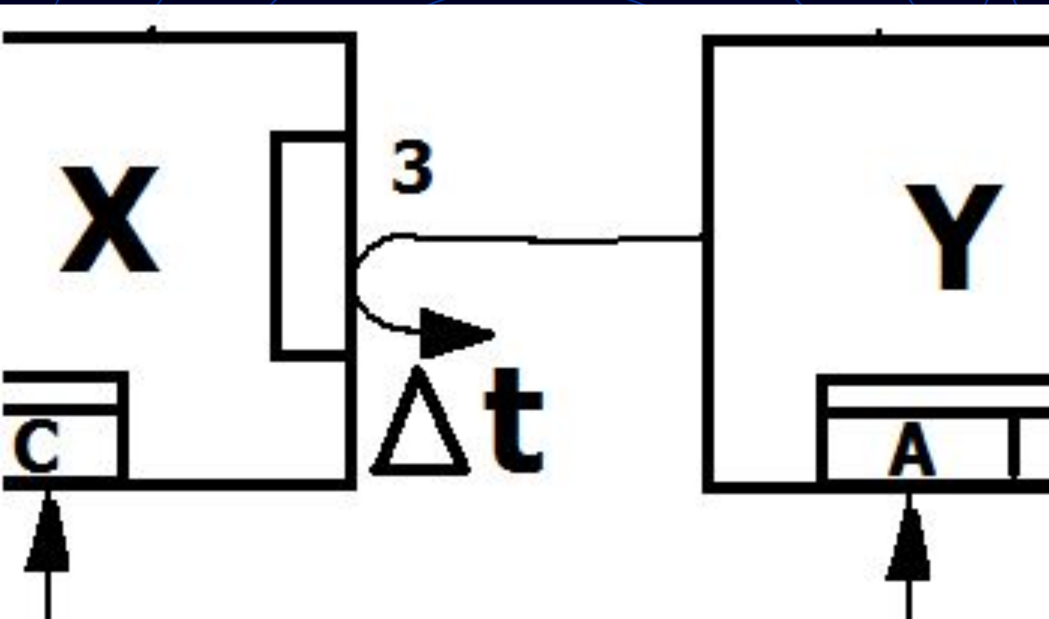
3 – простой вход
2 – альтернативный вход

Процесс



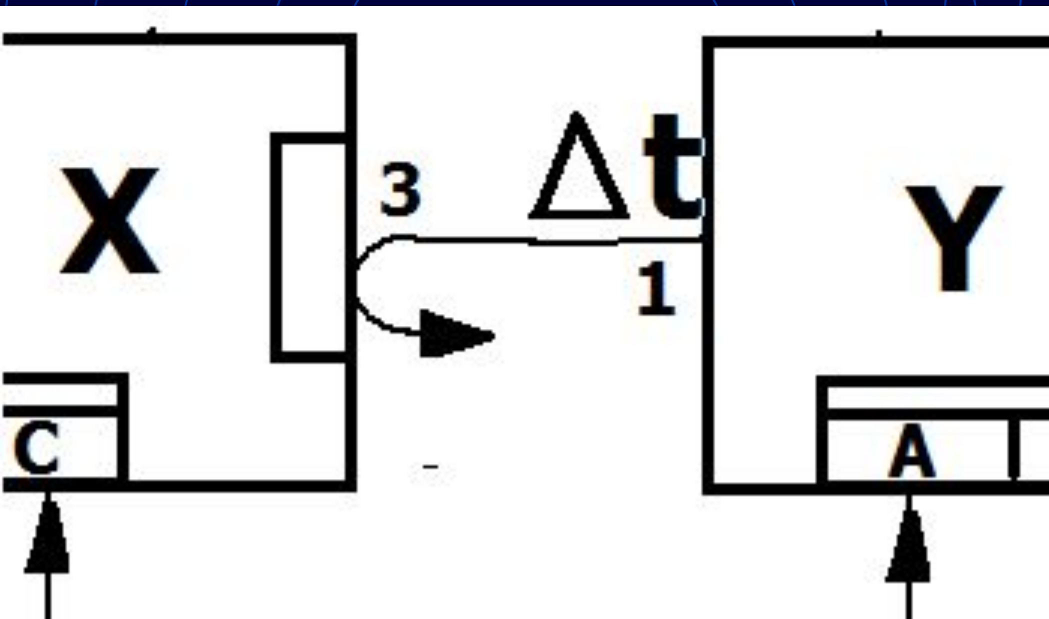
1, 3 - по некоторым причинам процесс может не выполнять взаимодействие

Процесс

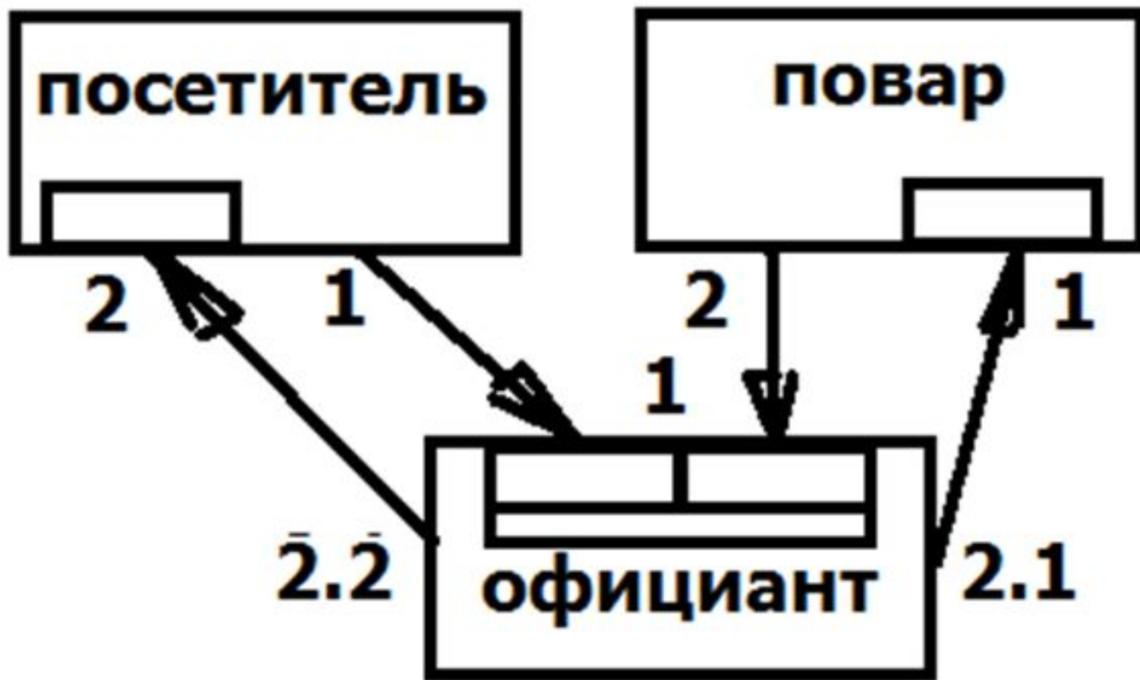


Процесс X, начав ожидание по входу 3 и не дождавшись его , через некоторое время прекращает ожидание и продолжает работу со следующей операции

Процесс



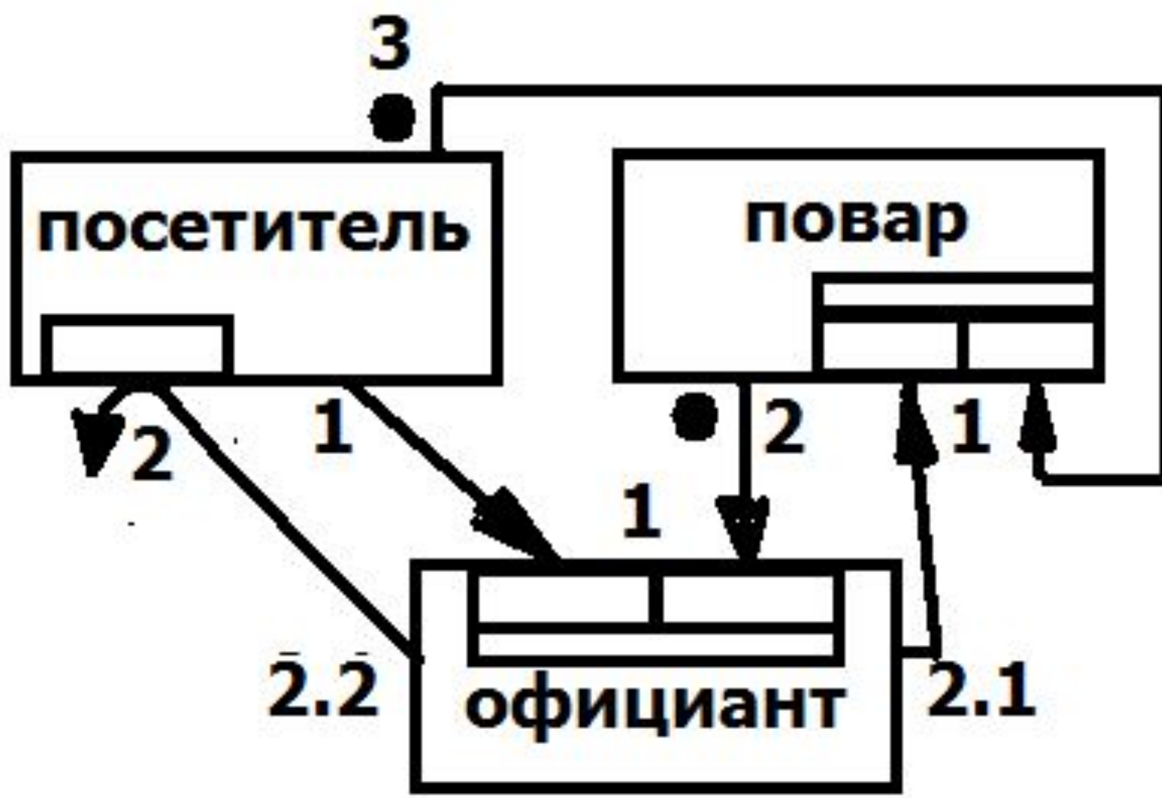
Процесс Y, начав вызов 1 и не дождавшись приема взаимодействия, через некоторое время прекращает ожидание и продолжает работу со следующей операции

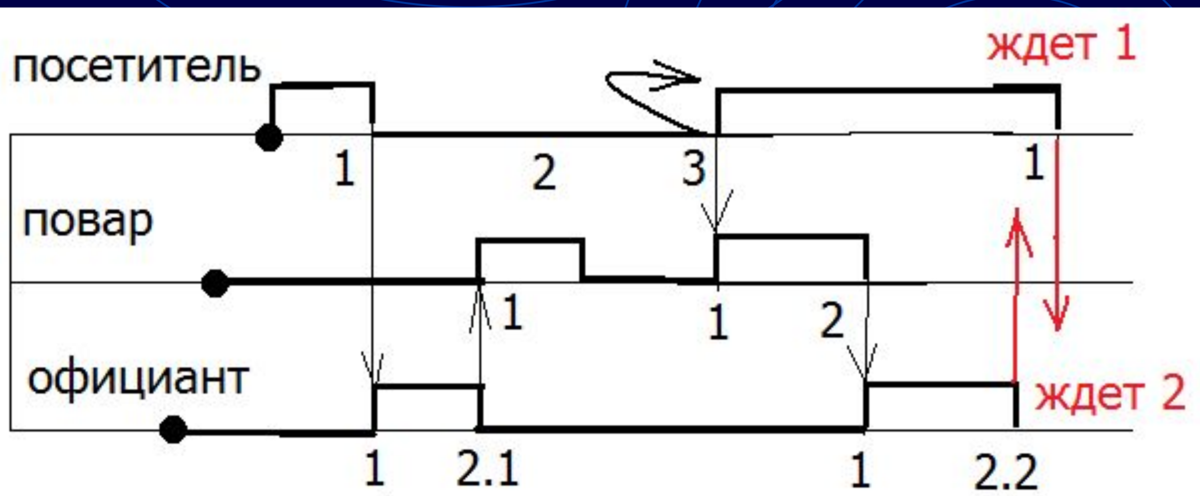


пример

Посетитель: 1 -дал официанту заказ
2 – ждет еду
Повар : 1 – ждет заказ
2 – отдает приготовленную еду
Официант: 1 –ждет, кто к нему обратится
2 – в зависимости от ситуации выполняет 2.1 или 2.2

Пример зависания системы





Посетитель:
2 – ждет еду, не дождавшись идет
к поvarу (3)

Повар : 1 – ждет заказ
2 – может не приготовить еду

Диаграмма последовательности – линии жизни

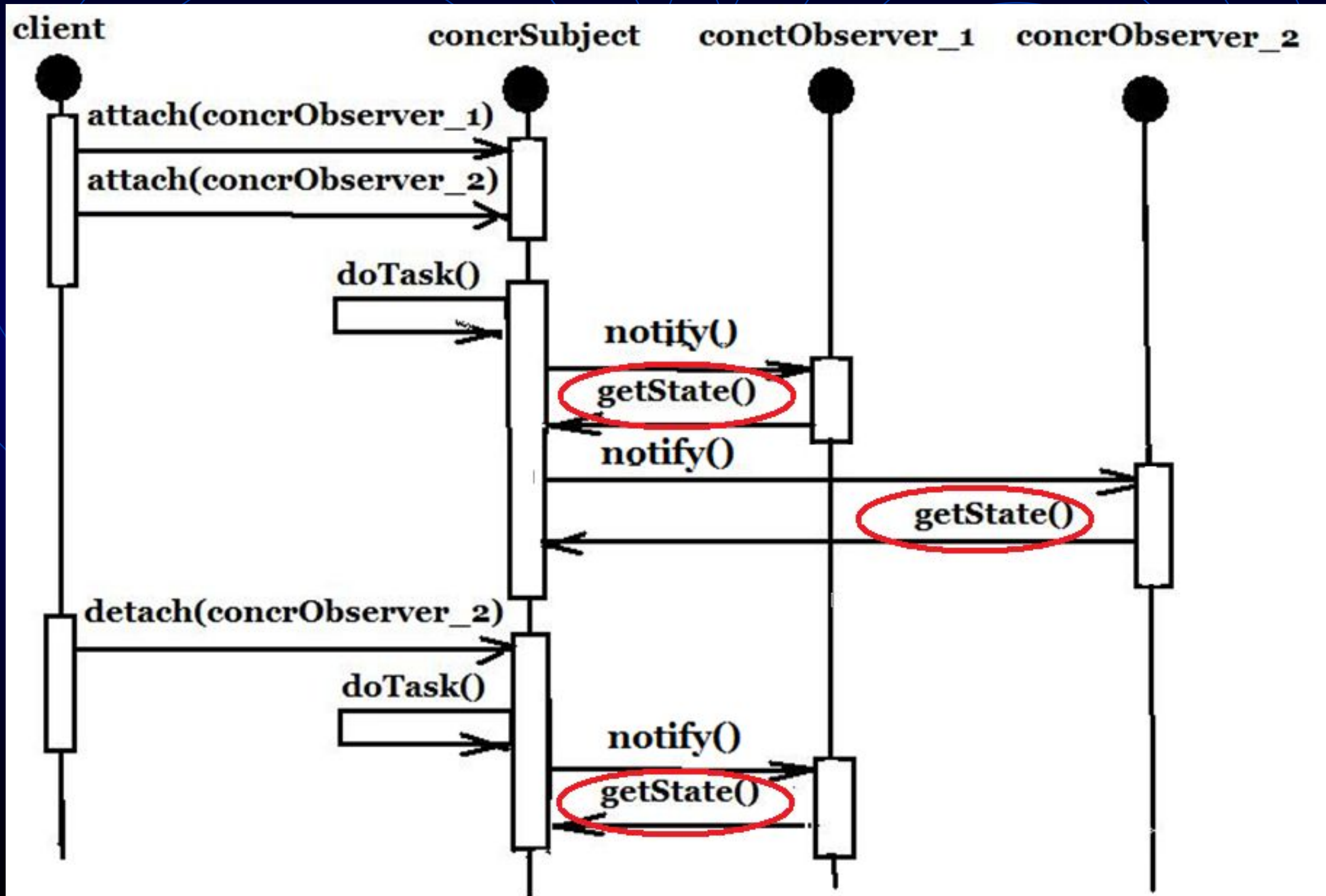
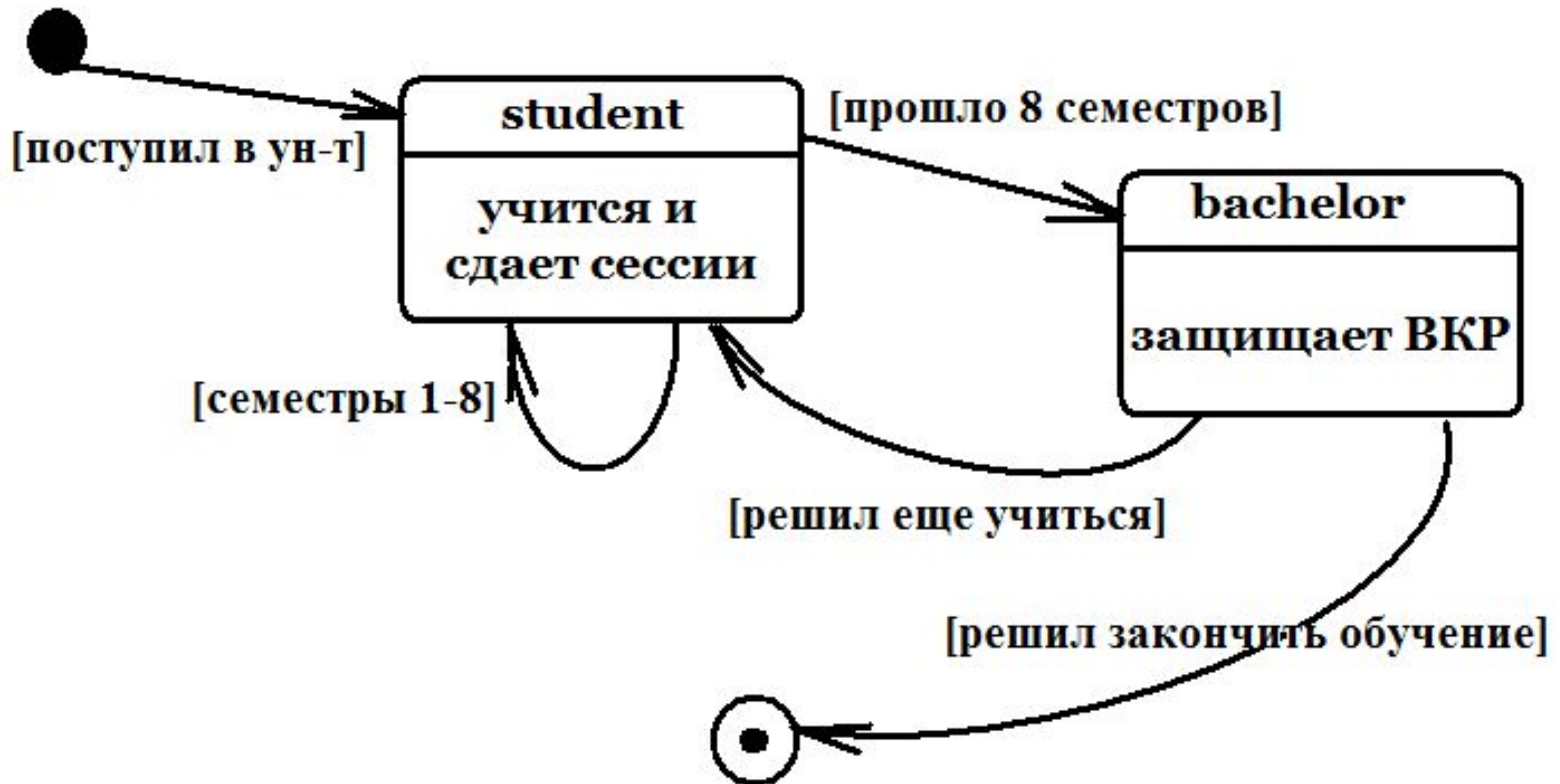
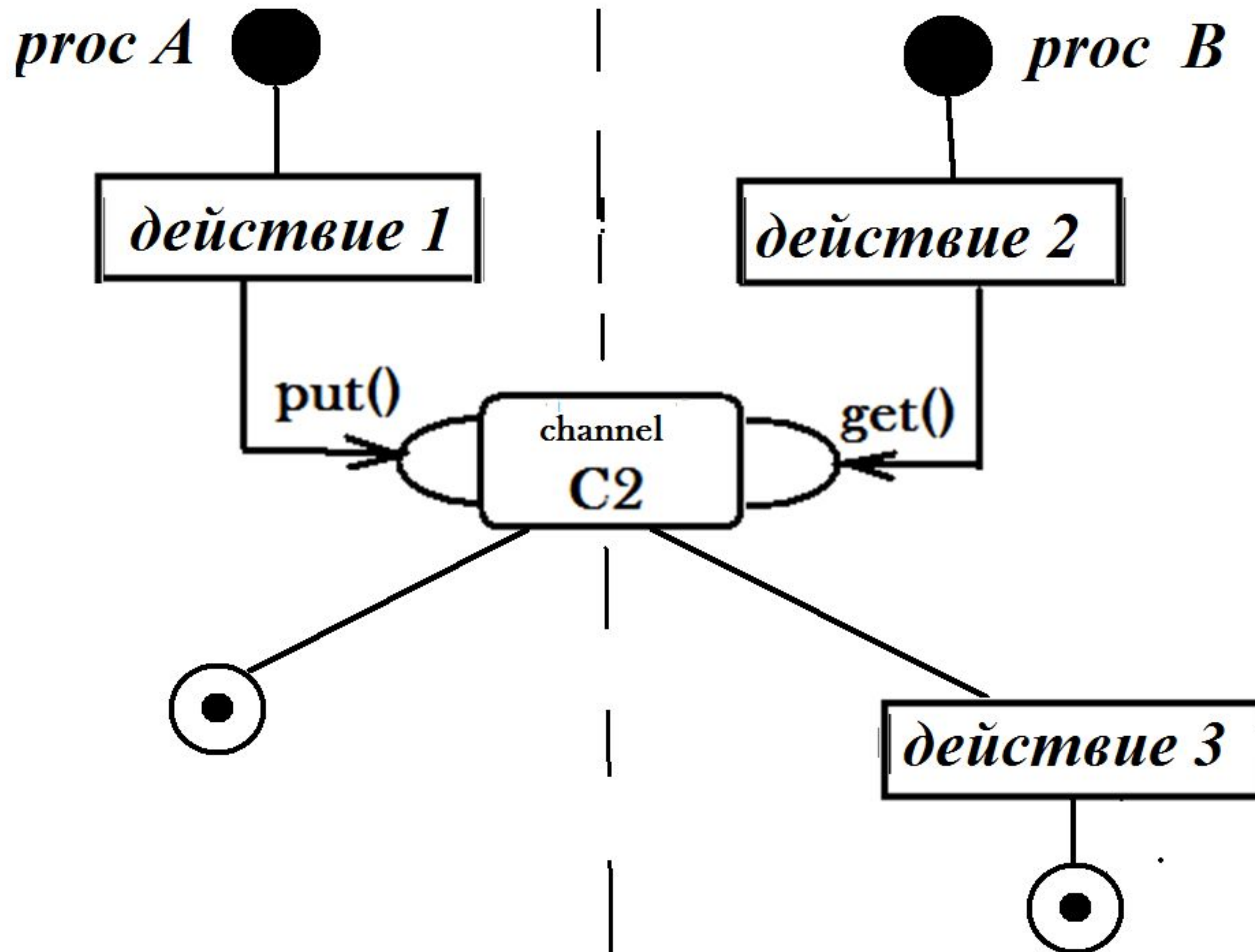


Диаграмма состояний

Синтаксис состояния = название + деятельность
Синтаксис дуги = событие [условие] / действие.



Плавательные дорожки



Эффективные алгоритмы параллельного программирования -

Закон Амдала

T - время работы программы при однопоточном выполнении

P - часть кода осталась последовательной

N – количество потоков

коэффициент ускорения равен

K=T/(T*P + T*(1-P)/N) = 1/(P+(1-P)/N)

(алгоритмов без последовательных команд практически не существует)

	P=0.2	**P=0.4**	**P=0.6**	**P=0.8**
N=2	1.67	1.48	1.25	1.10
N=5	2.78	1.92	1.47	1.19
N=100	4.98	2.49	1.67	1.25

Проблемы параллельного программирования

Хотя основная трудоёмкость жизненного цикла программ связана с тестированием и отладкой программ, это направление за полвека профессионального программирования практически не получило должного прогресса.

Если первый барьер к успеху в параллельном программировании обусловлен последовательным стилем мышления, то второй зависит от до сих пор не преодолённой трудоёмкости отладки.

Проблемы при параллельном выполнении

- Выделение порций работ, которые могут быть выполнены параллельно
- Гонки данных
- Взаимная блокировка
- Синхронизация алгоритмов на разных этапах выполнения
- Эффективность распараллеливания и голодание потоков
- Масщтабируемость

Синхронизация и обмен данными

- Синхронизация — это процесс, при помощи которого два или более программных потока координируют свои действия. Например, один поток, чтобы продолжить выполнение, ждет, когда другой закончит свое задание.
- Взаимодействие — обмен данными между программными потоками, с которым связаны проблемы ширины полосы пропускания и задержек.

Мертвая блокировка (Deadlock)

Мертвая блокировка происходит тогда, когда поток заблокирован в ожидании ресурса другого потока, который никогда не освободится. В зависимости от обстоятельств могут иметь место различные варианты блокировки:

- самоблокировка,
- рекурсивная мертвая блокировка ,
- мертвая блокировка по порядку блокирования.

Балансировка и масштабируемость

- Балансировка нагрузки — распределение работы между несколькими программными потоками так, чтобы все они выполняли примерно одинаковый объем работы.
- Масштабируемость — проблема эффективного использования большего числа программных потоков при запуске программы на более мощной системе. Например, если программа написана для эффективного использования четырех ядер, будет ли она эффективно работать на системе с восемью процессорными ядрами?

Пример гонки данных

```
int i = 0; // переменная, видимая из двух потоков

//код в теле первого потока:
for(int k=0; k<50000; k++){
     i--;      i++;
// при последовательном выполнении переменная
 // остается равной нулю
}

//код в теле второго потока:
for(int k=0; k<50000; k++) {
     i++;    i--;
}
```

Пример гонки данных

```
G:\...Magistr_PSMP\01_Алгоритмы\thread>main.exe
Producer 2 stop
Consumer 2 stop
Producer 1 stop
Consumer 3 stop
Consumer 1 stop
549
G:\...Magistr_PSMP\01_Алгоритмы\thread>main.exe
Producer 1 stop
Producer 2 stop
Consumer 2 stop
Consumer 3 stop
Consumer 1 stop
590
G:\...Magistr_PSMP\01_Алгоритмы\thread>main.exe
Producer 2 stop
Consumer 2 stop
Producer 1 stop
Consumer 3 stop
Consumer 1 stop
345
G:\...Magistr_PSMP\01_Алгоритмы\thread>main.exe
Producer 1 stop
Producer 2 stop
Consumer 2 stop
Consumer 1 stop
Consumer 3 stop
753
G:\...Magistr_PSMP\01_Алгоритмы\thread>main.exe
Producer 2 stop
Consumer 2 stop
Consumer 3 stop
Producer 1 stop
Consumer 1 stop
475
G:\...Magistr_PSMP\01_Алгоритмы\thread>main.exe
Producer 1 stop
Producer 2 stop
Consumer 2 stop
Consumer 1 stop
Consumer 3 stop
596
G:\...Magistr_PSMP\01_Алгоритмы\thread>
1Help  2UserMn 3View  4Edit  5Copy  6RenMov 7MkFold 8Delete 9Conf
```

{G:\LECTURES\Mag... {C:\LECTURES\Mag... Microsoft PowerPoi... EN

Уровни распараллеливания

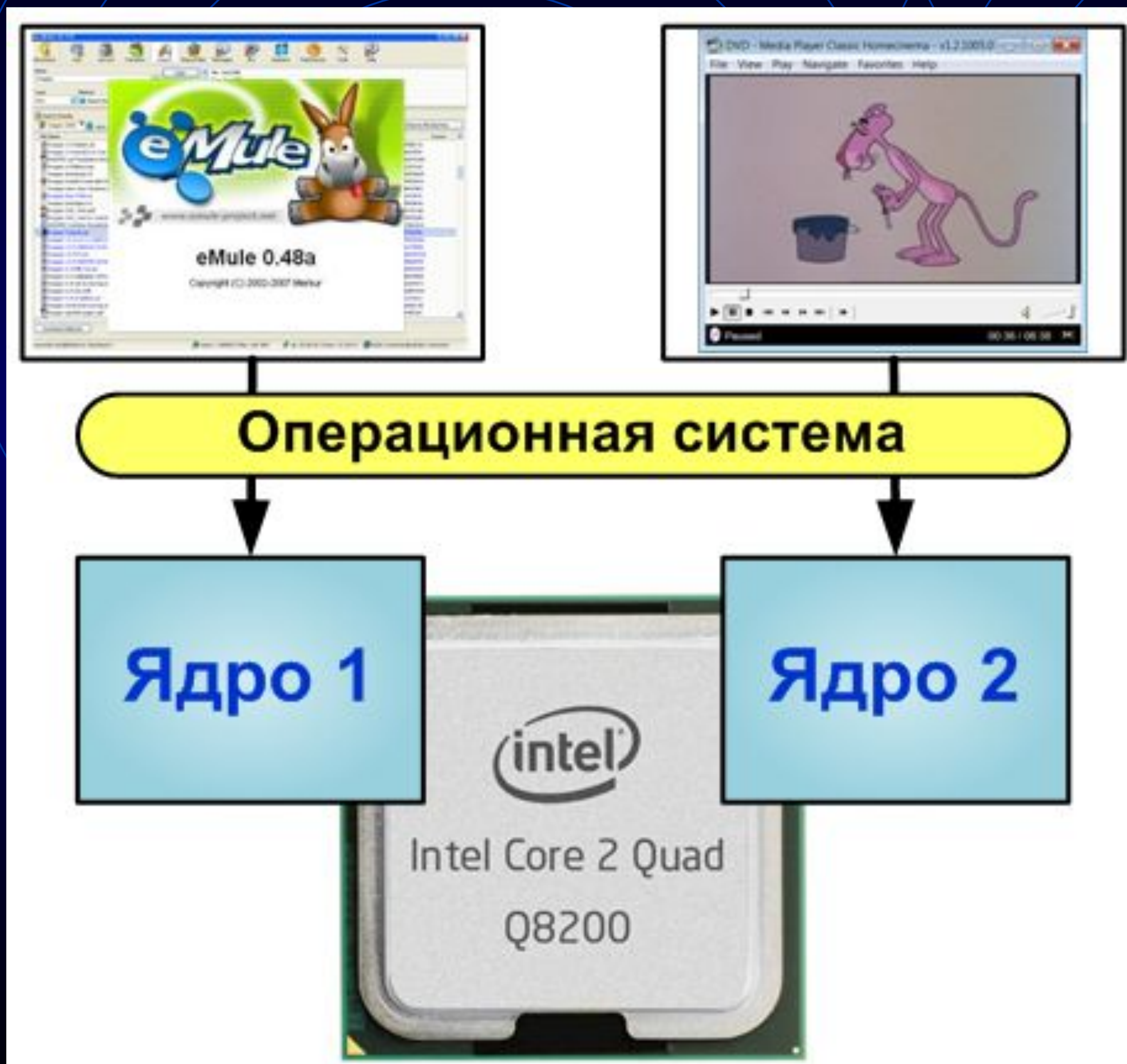
Распараллелить решение задачи можно на нескольких уровнях.

Классификация методов деления достаточна условна и служит для демонстрации разнообразия подходов к задаче распараллеливания.

Между типами распараллеливания нет четкой границы и конкретную технологию распараллеливания бывает сложно отнести к одному из них.

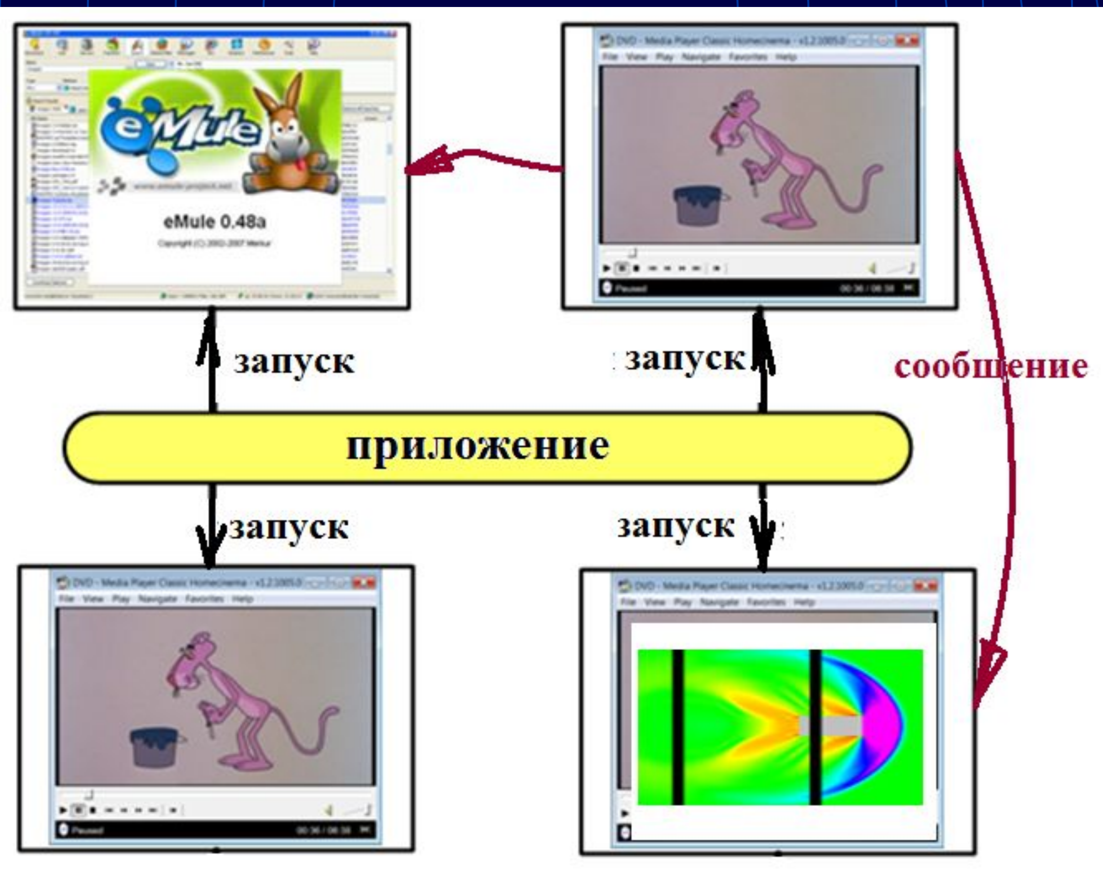
Классификация методов
распараллеливания алгоритмов

(1)Распараллеливание на уровне задач



Независимость решаемых подзадач (параллельно работают : обработка файлов, компиляция в VS, paint…)

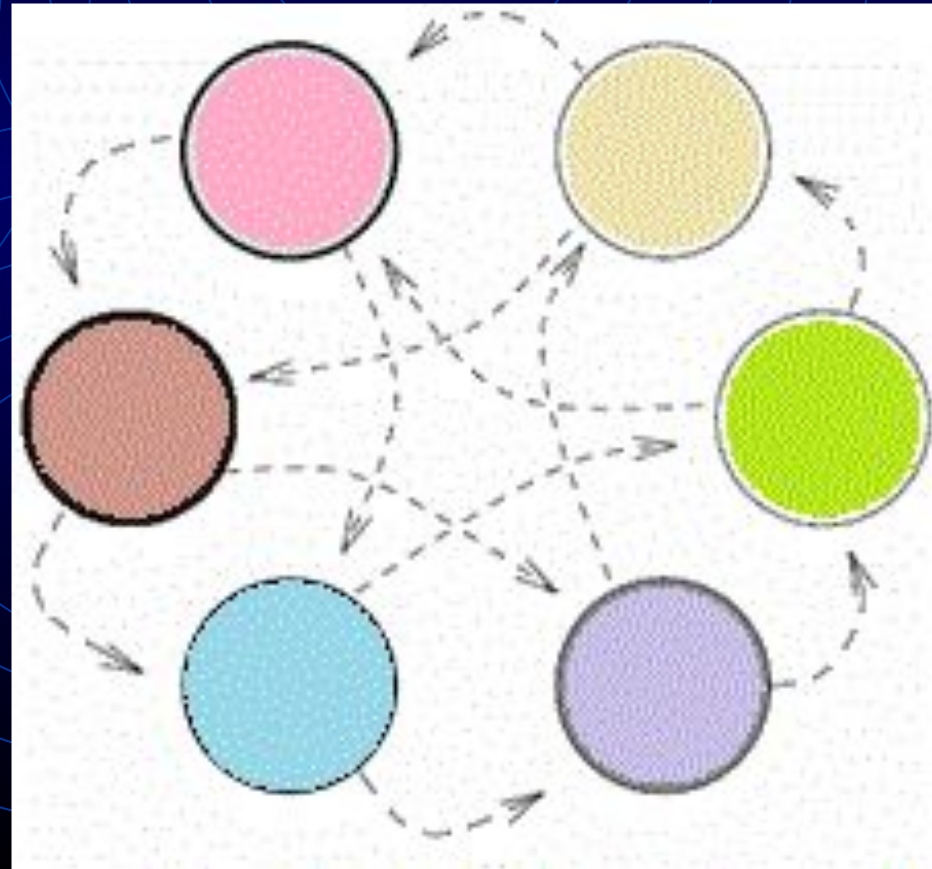
(2) Распараллеливание на уровне передачи сообщений



Приложение состоит из набора процессов с различными адресными пространствами, каждый из которых функционирует на своем исполнителе.

Пример: Островная модель генетического алгоритма

Исходная популяция делится на несколько частей, каждая из которых развивается обособленно от всех остальных (на своем «*острове*»). Каждому острову выделяется процесс. Через N поколений процессы обмениваются лучшими особями (*мигрантами*).



Островная модель генетического алгоритма

Островная модель не просто предоставляет способ распараллеливания вычислений, но и **имеет преимущество над тривиальным многократным запуском** одинаковых генетических алгоритмов: в момент миграции вновь прибывшая особь может вывести популяцию из области локального экстремума, что позволит продолжить эволюцию в сторону глобального максимума/минимума

Зернистость при реализации системы на уровне передачи сообщений

Зернистость – это мера отношения количества вычислений, сделанных в параллельной задаче, к количеству пересылок данных. Мелкозернистый параллелизм – очень мало вычислений на каждую пересылку данных. Крупнозернистый параллелизм – интенсивные вычисления на каждую пересылку данных (данные могут пересылаться большими порциями).

Чем мельче зернистость, тем больше точек синхронизации

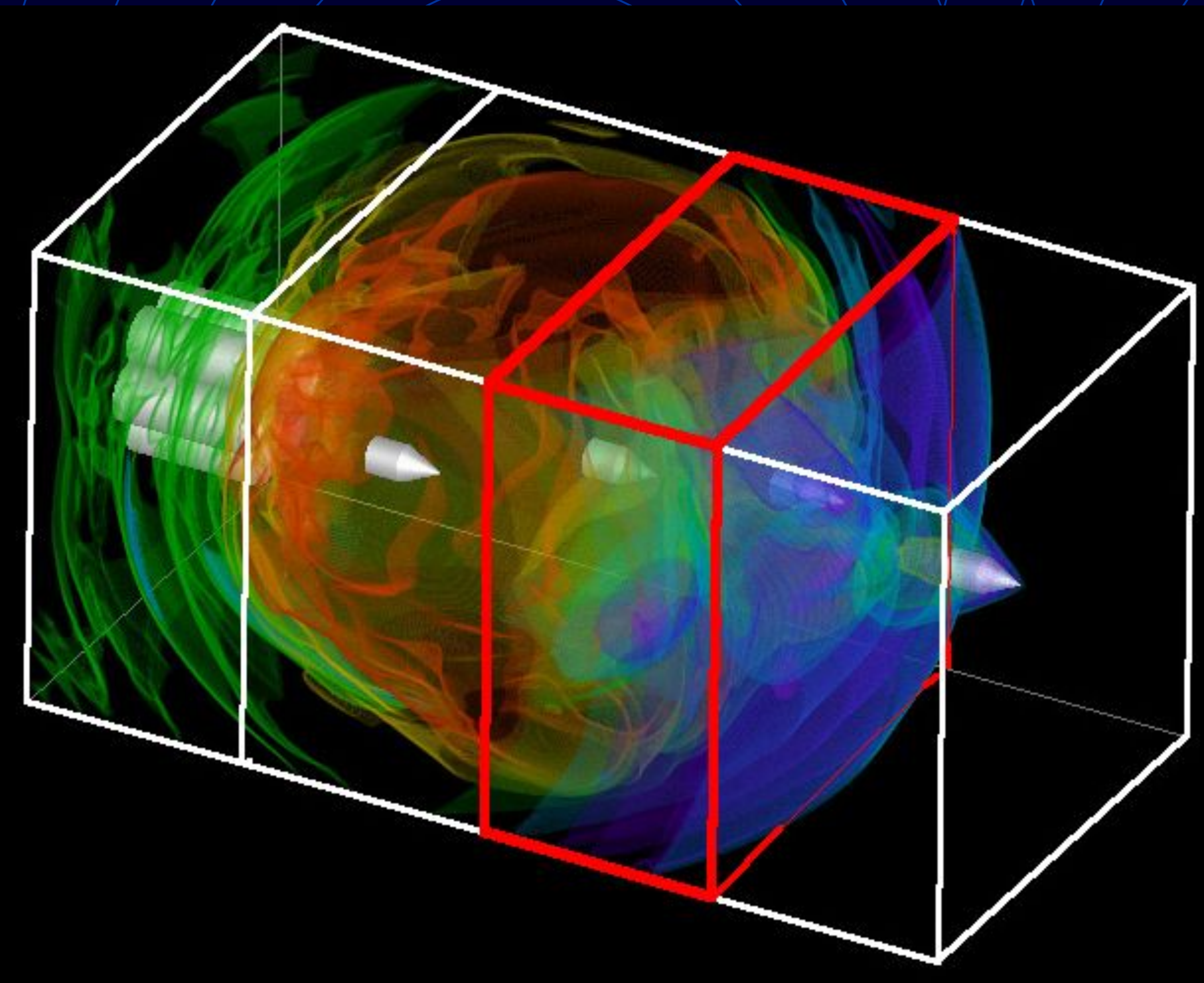
(3) Распараллеливание на уровне разделяемой памяти

Приложение состоит из набора нитей исполнения, использующих разделяемые переменные и примитивы синхронизации. Две подмодели:

— использование системных/библиотечных вызовов для организации работы потоков (полный контроль над выполнением, но трудоемко)

— программирование на языке высокого уровня с использованием соответствующих прагм (легко в программировании, но нет полного контроля над процессом выполнения).

(4) Распараллеливание на уровне данных (декомпозиция по данным)

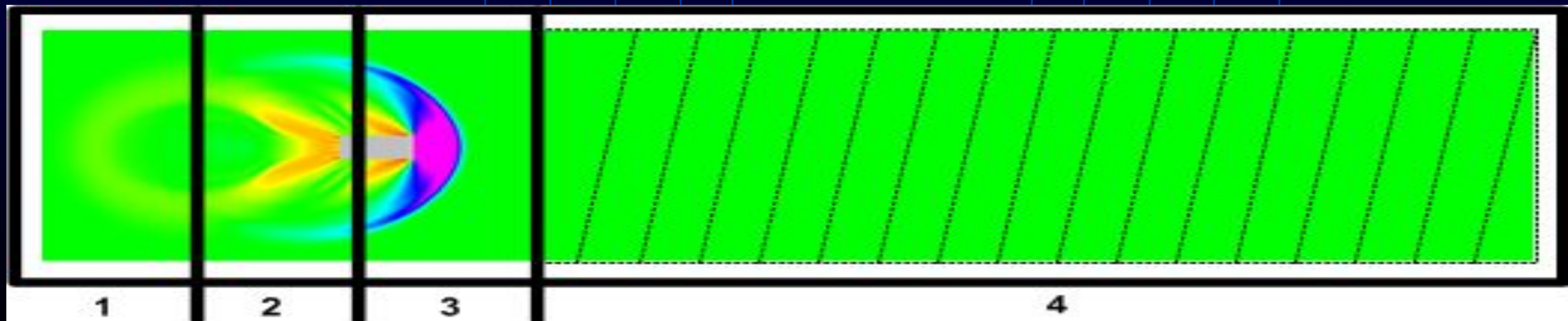


Применение одной и той же операции к элементу данных (компиляция в VS, прогноз погоды, расчет статистики социологических опросов,…)

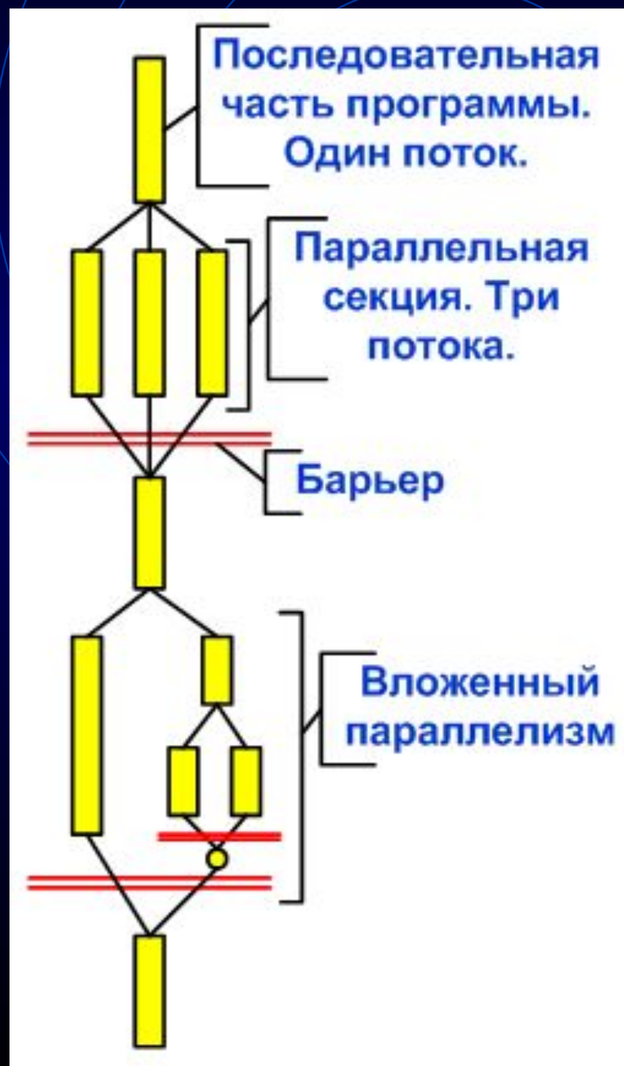
(4а) Частный случай распараллеливания по данным – геометрический параллелизм

1- необходимо передавать данные получаемые на границах геометрических областей другим ядрам.

2 - методы повышения скорости расчета за счет балансировки нагрузки между вычислительными узлами.



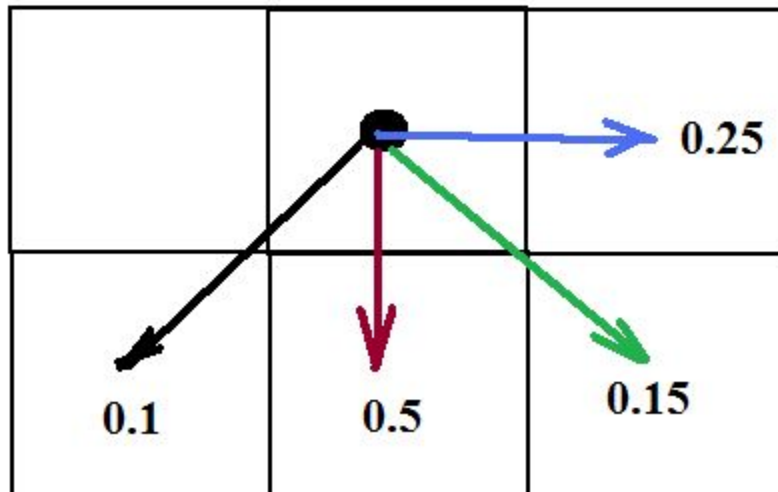
(5) Распараллеливание на уровне алгоритмов (функциональная декомпозиция)

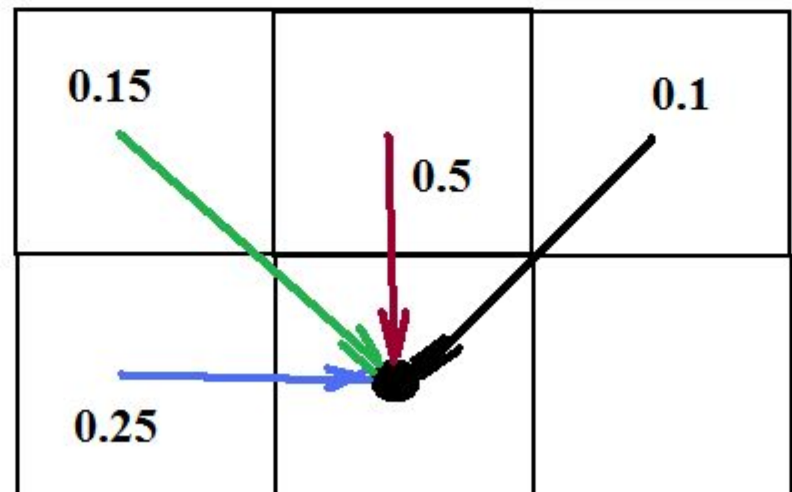


Распараллеливание отдельных процедур и алгоритмов (примеры: алгоритмы параллельной сортировки, умножение матриц, решение системы линейных уравнений,...) Удобно использовать OpenMP

(5) Рапараллеливание по операциям

1. Поиск минимального элемента в массиве – разбиение задачи по операциям, сложность O(1). Это пример задачи с одновременной записью без гонок КТО ПРЕДЛОЖИТ АЛГОРИТМ?
2. Преобразование цветного изображения в черно-белое





(7) Параллелизм на уровне инструкций

Осуществляется на уровне параллельной обработки процессором нескольких инструкций (конвейеры команд, предсказание команд, переименование регистров – работу по низкоуровневой оптимизации выполняет компилятор)

Параллельные алгоритмы и их анализ

Dependency Graphs – граф зависимостей

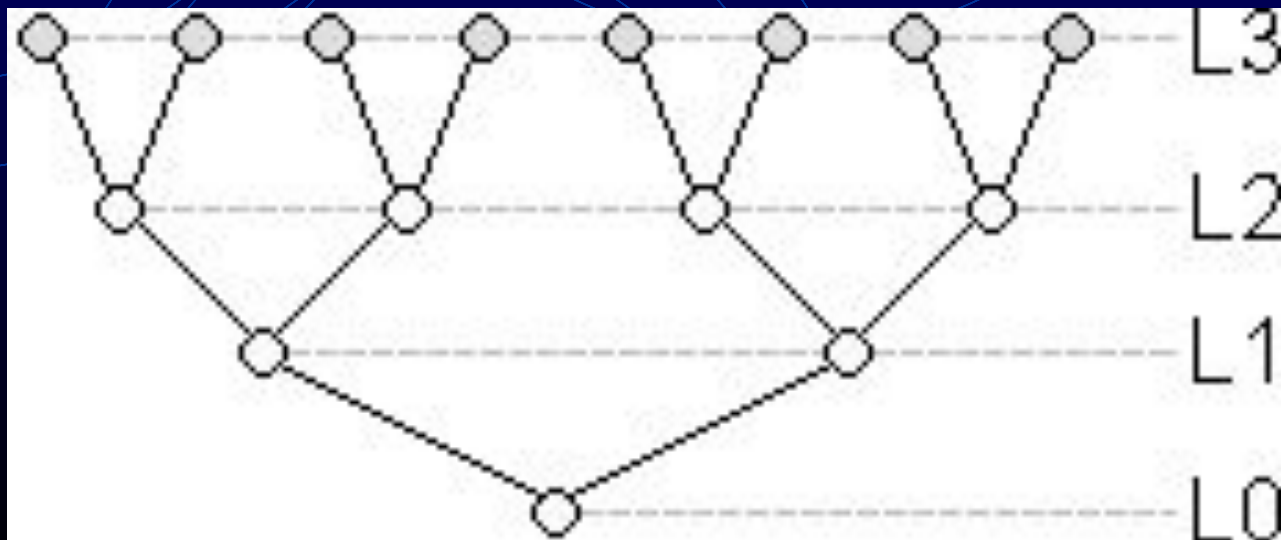
Mapping – отображение графа зависимостей на выполнение на конкретных процессорах

Анализ распараллеливания алгоритма на зависимости между данными по операциям

Вершины графа зависимости – фрагмент вычислений

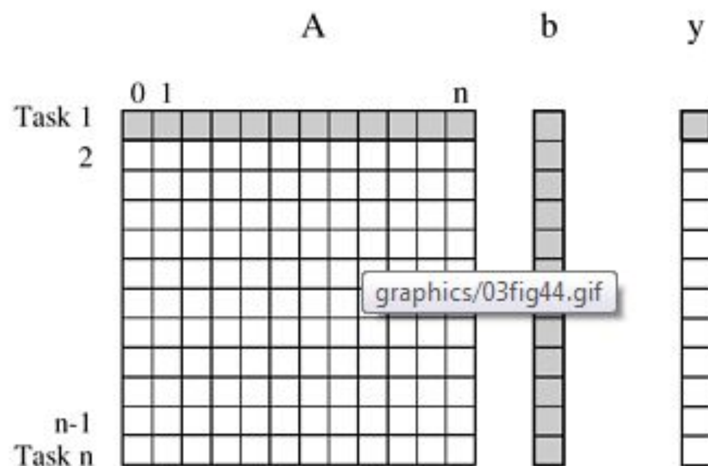
Ребра – зависимости между выходными данными вычислений и входными данными следующего этапа

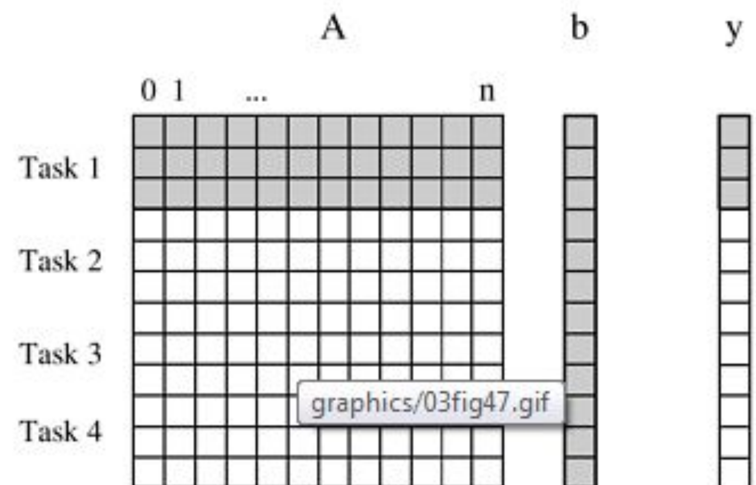
высота (число этапов) и ширина схемы (число процессов)



Декомпозиция по данным

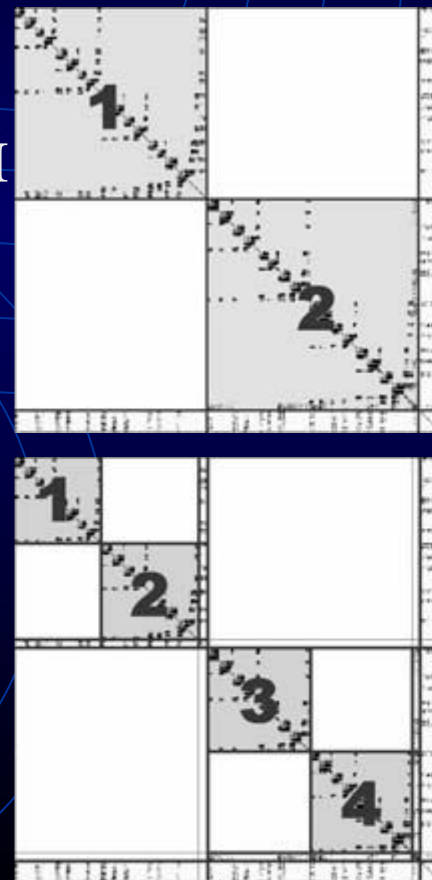
Умножение матрицы на вектор – декомпозиция на N задач, где N – число рядов матрицы. Все задачи независимы и могут выполняться в любом порядке. Возможна декомпозиция на K задач, например, на 3 задачи – по N/3 рядов на задачу.





Пример декомпозиции по данным: Блочно независимые вычисления

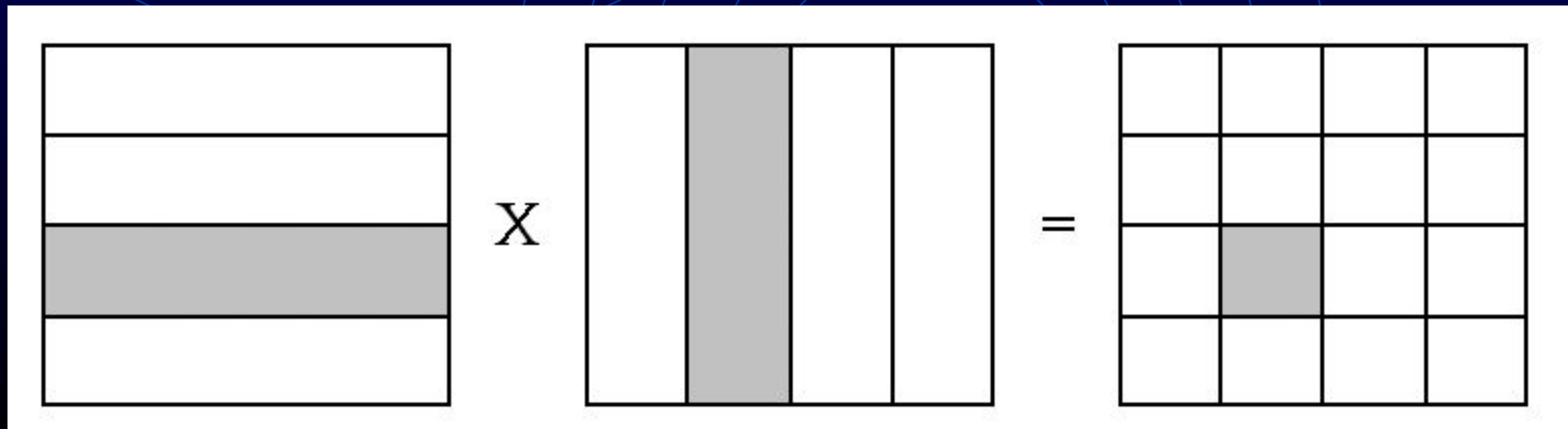
Пример – анализ зависимости операндов и построение бинарного дерева зависимых вычислений для блочной матрицы.



Блочно независимые вычисления связаны с независимой обработкой диагональных блоков

Ленточный алгоритм умножения матриц O(2N^3/p)

В процессе вычислений i-ый процессор умножает i- ый горизонтальный блок на текущий принадлежащий ему вертикальный блок. На следующем шаге происходит циклический сдвиг влево вертикальной полосы к следующему процессору



Метод сдваивания (задача свертки – reduce problem)

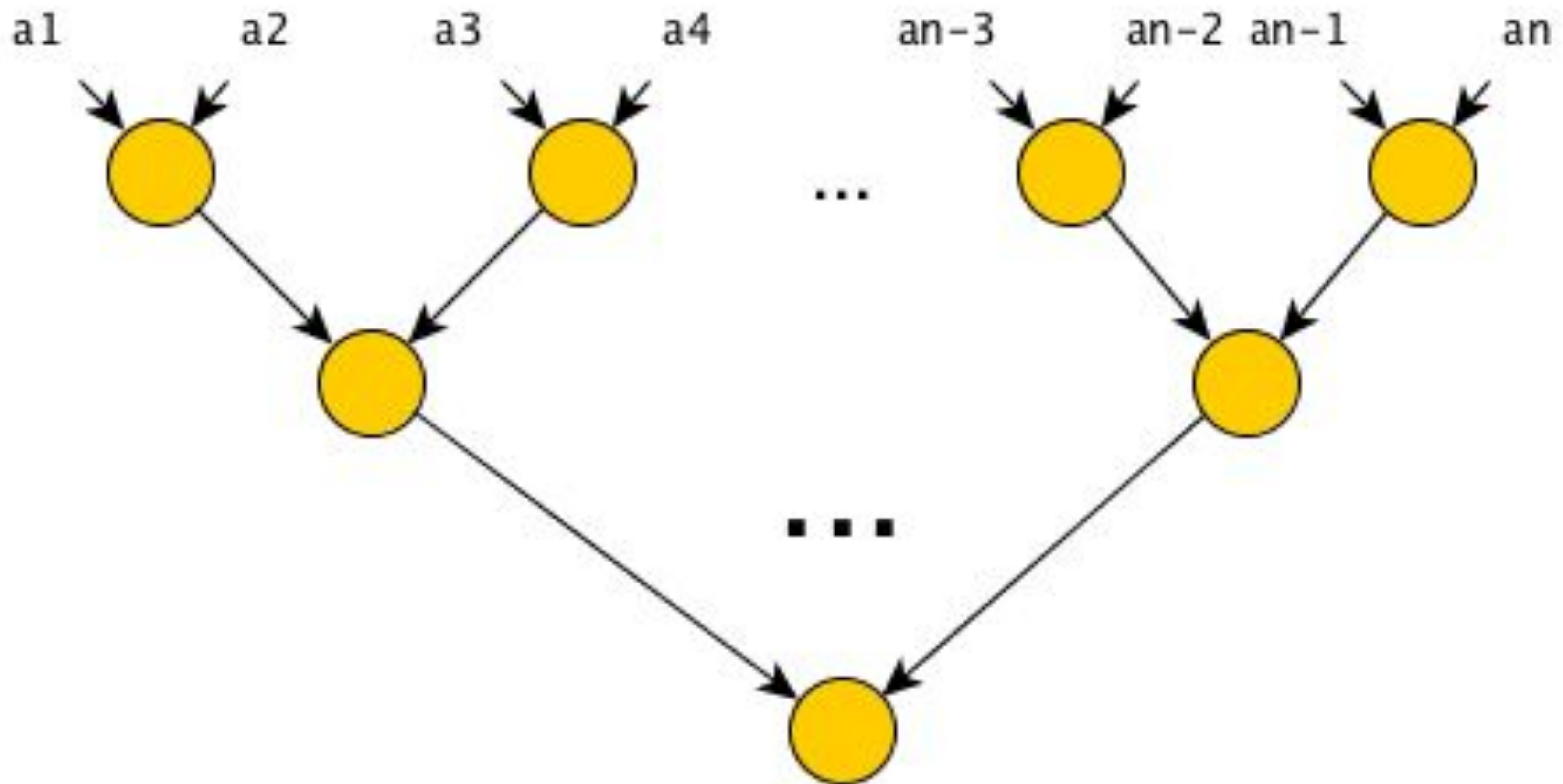
Найти произведение a1*a2*…*a8

Уровень								
0	a1	a2	a3	a4	a5	a6	a7	a8
1	a1a2		a3a4		a5a6		a7a8	
2	(a1a2)(a3a4)				(a5a6)(a7a8)			
3	(a1a2a3a4)(a5a6a7a8)							

Высота схемы 3, ширина 4

O(n) □ *O(log(n))*, количество операций — *O(n)*

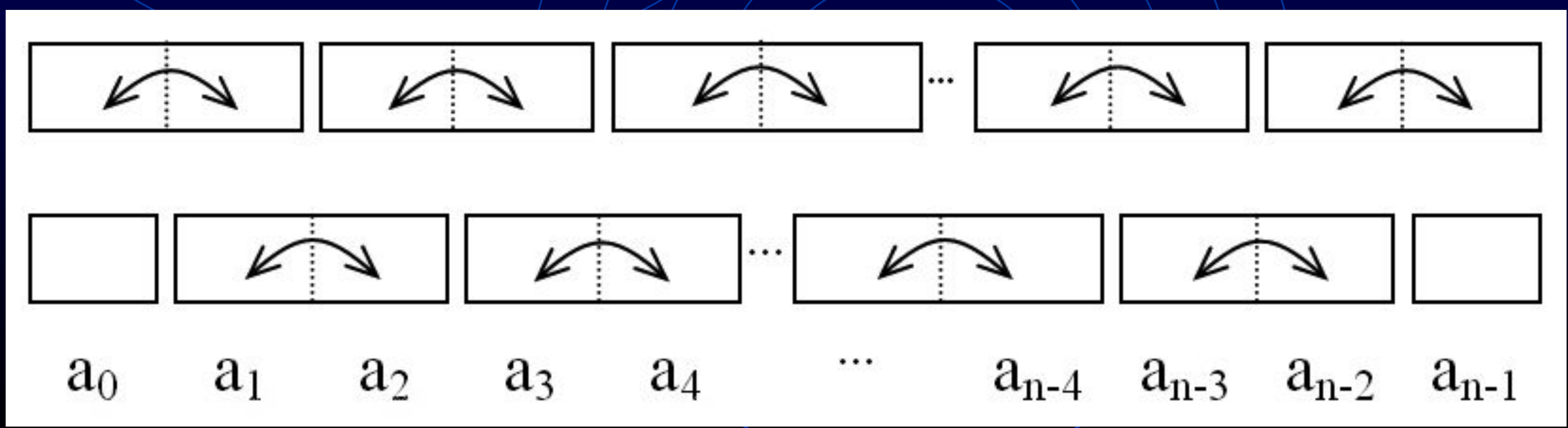
Метод сдваивания
(задача свертки – reduce problem)



Чет - нечетная сортировка

Шаг 1 –сортировка a[i] и a[i+1]
Шаг 2 –сортировка a[i+1] и a[i+2]
Повторяем «Шаг 1» и «Шаг 2» N раз
Сложность O(N)

Число процессоров может быть равно N/2



Чет - нечетная сортировка

шаг 1 8 7 6 5 4 3 2 1

шаг 2 7 8 5 6 3 4 1 2

шаг 3 7 5 8 3 6 1 4 2

шаг 4 5 7 3 8 1 6 2 4

шаг 5 5 3 7 1 8 2 6 4

шаг 6 3 5 1 7 2 8 4 6

шаг 7 3 1 5 2 7 4 8 6

шаг 8 1 3 2 5 4 7 6 8

1 2 3 4 5 6 7 8

Обработка списков за $\log_2 N$ (пример – найти свой номер от конца)

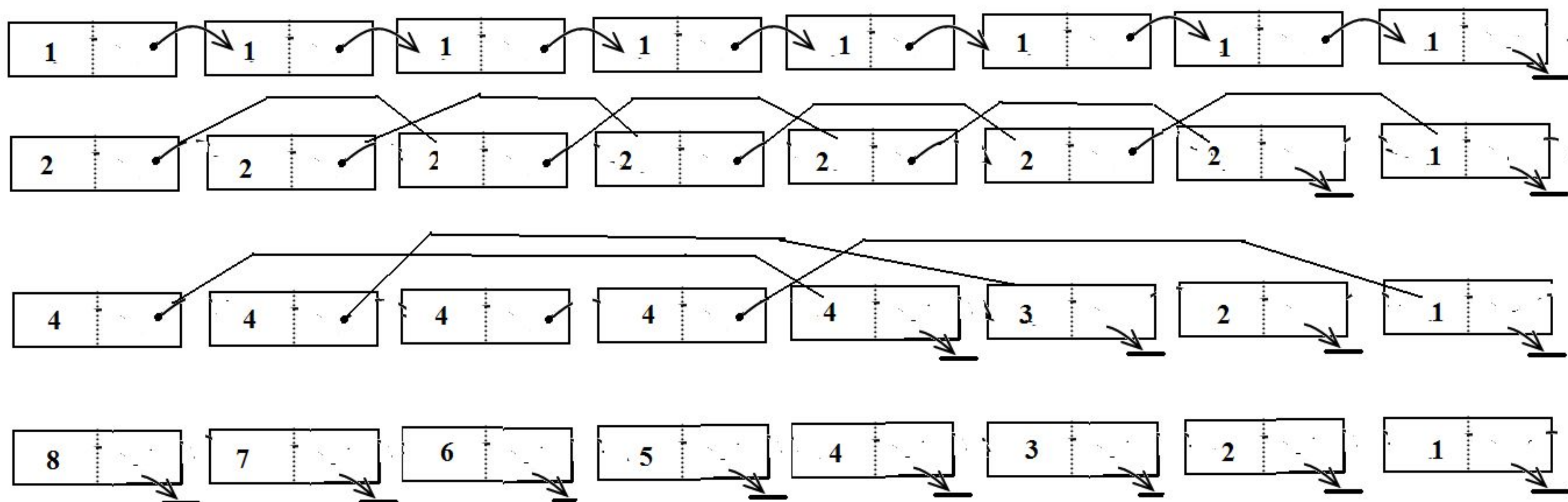
Подготовка: myNumber = 1;

Шаг 1: myNumber += next-> myNumber;

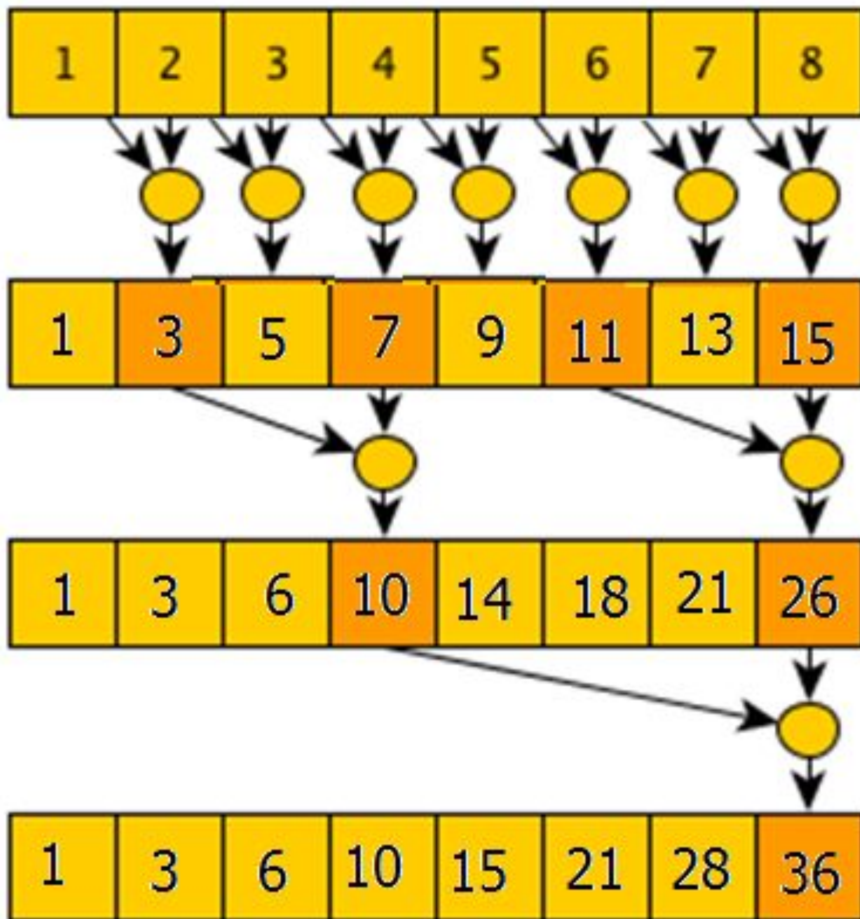
next = next->next;

Повторяем «Шаг 1» пока есть next != NULL

Сложность O($\log_2 N$)



Обработку списков за $\log_2 N$ модифицируем для массивов



(задача сканирования - Scan)

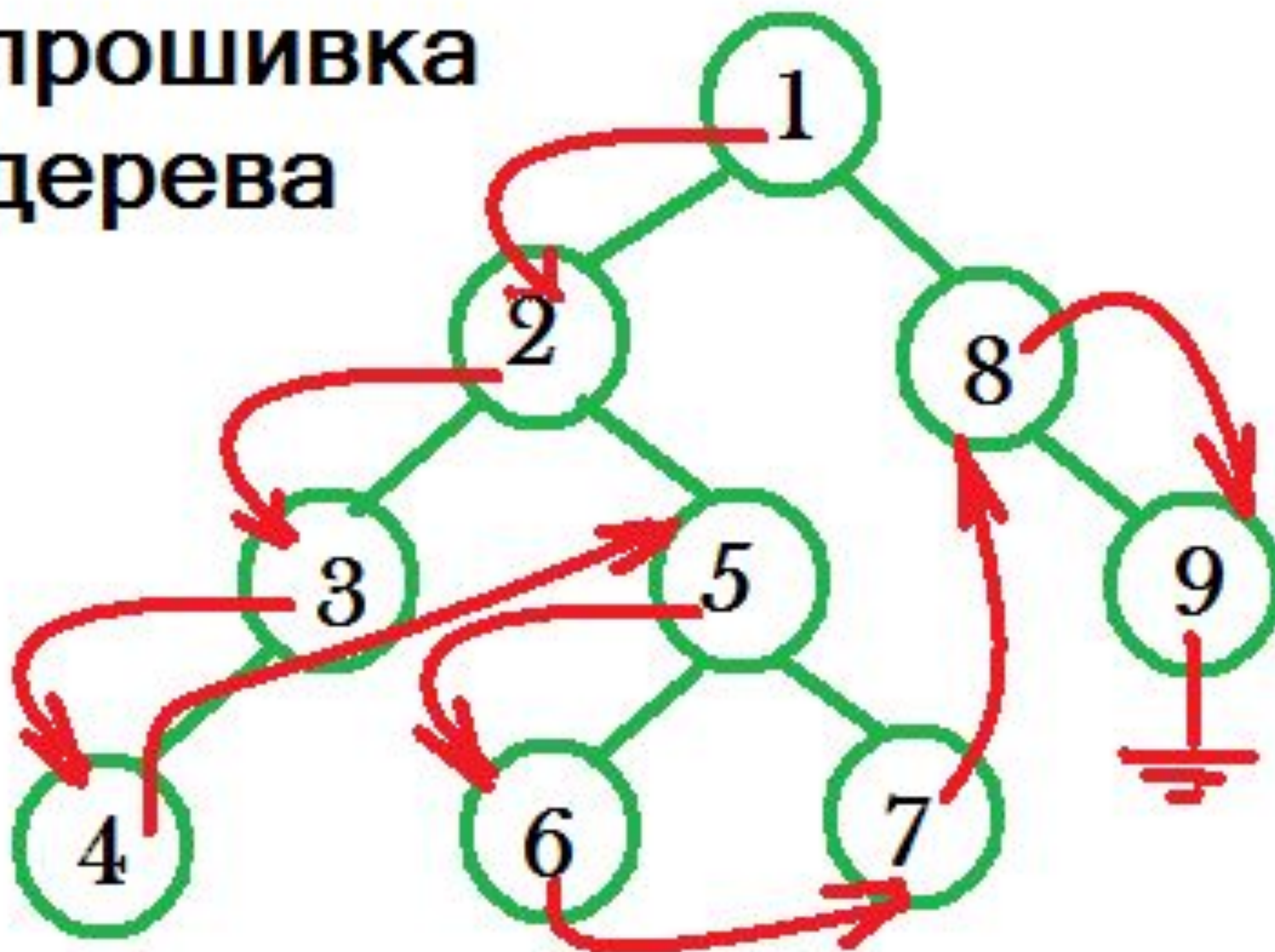
Считаем A [i] – сумма предыдущих от начала массива:

поток с номером i вычисляет

A[j] = A[i]+A[j]

и пересчитывает j

Обработка деревьев как списков

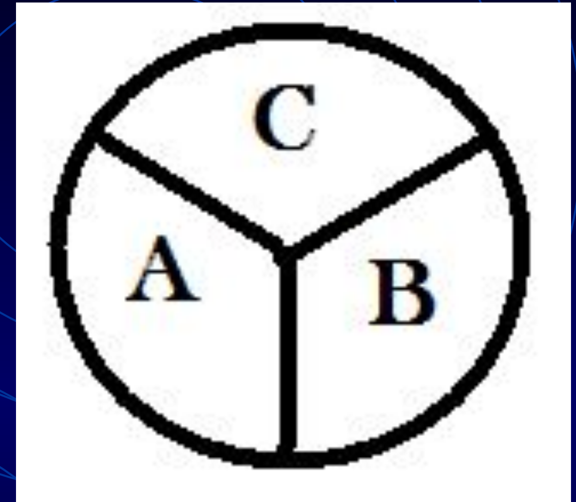


Обработка деревьев как списков за $\log_2 N$

Преобразуем дерево в список длины 3*N

Обрабатываем список за

$O(\log_2(N*(3*N))$

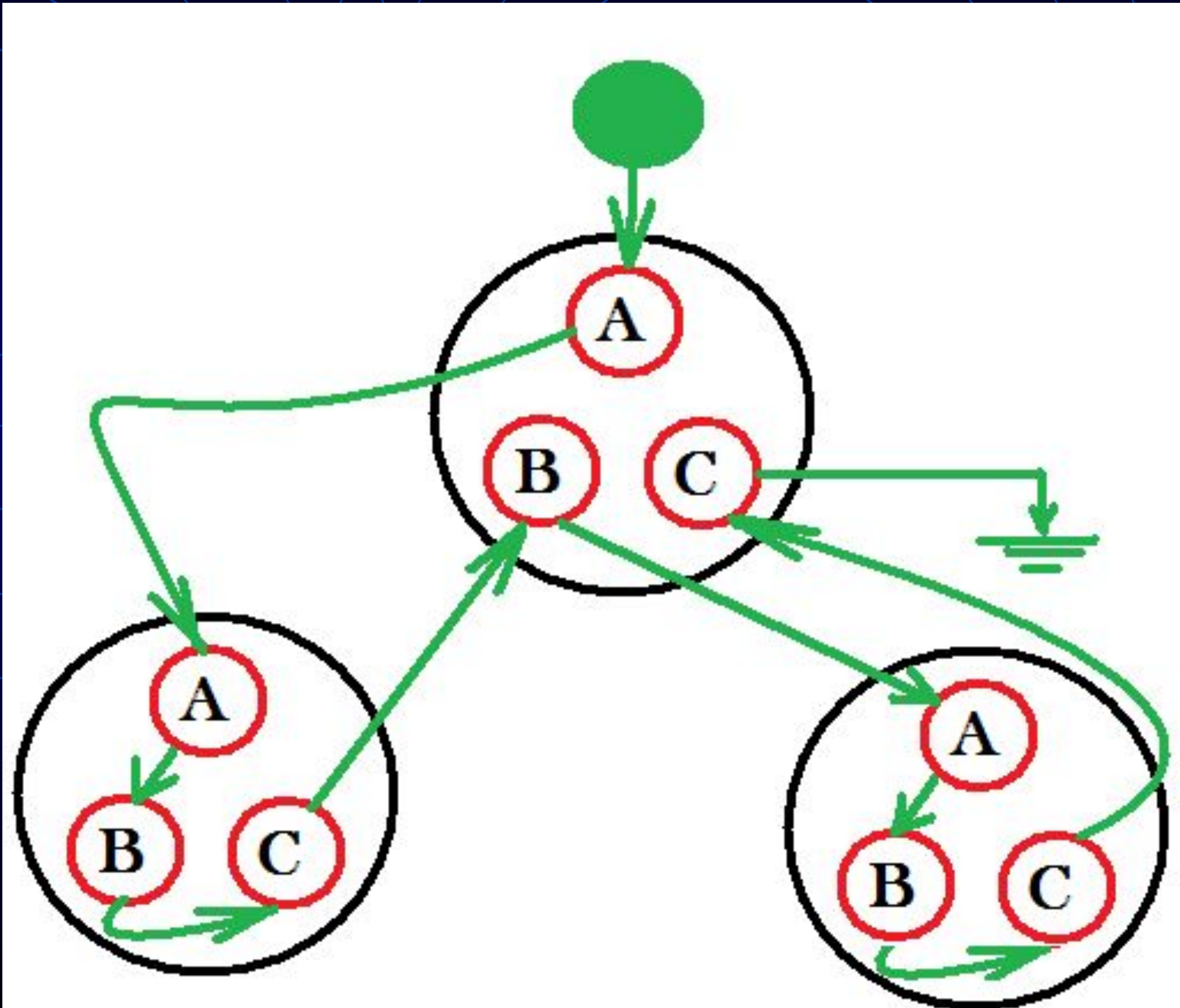



```
if (left  !- NULL) A= left->A ; else A=B;
if (right !-  NILL) B=right->A; else B=C;
if (this == root) C=NULL;
else
    if (this == up->left)  C=up->B; else C=up->C;
```

Обработка деревьев как списков за $\log_2 N$

dataA = +1
dataB = 0
dataC = -1

Глубина вершины равна значению A или B



Задачи на графах – минимальное остовное дерево

(алгоритм Прима – последовательная реализация)

1- выбирается произвольный начальный узел

2 – имеется построенная часть остова

3 – находится минимальное ребро, связывающее построенный остов с внешним узлом

4 - повторяется (3) пока есть не включенные в остов узлы

Сложность $O(E * \log_2 V) \leq O(V * V * \log_2 V)$ при использовании эффективных структур данных

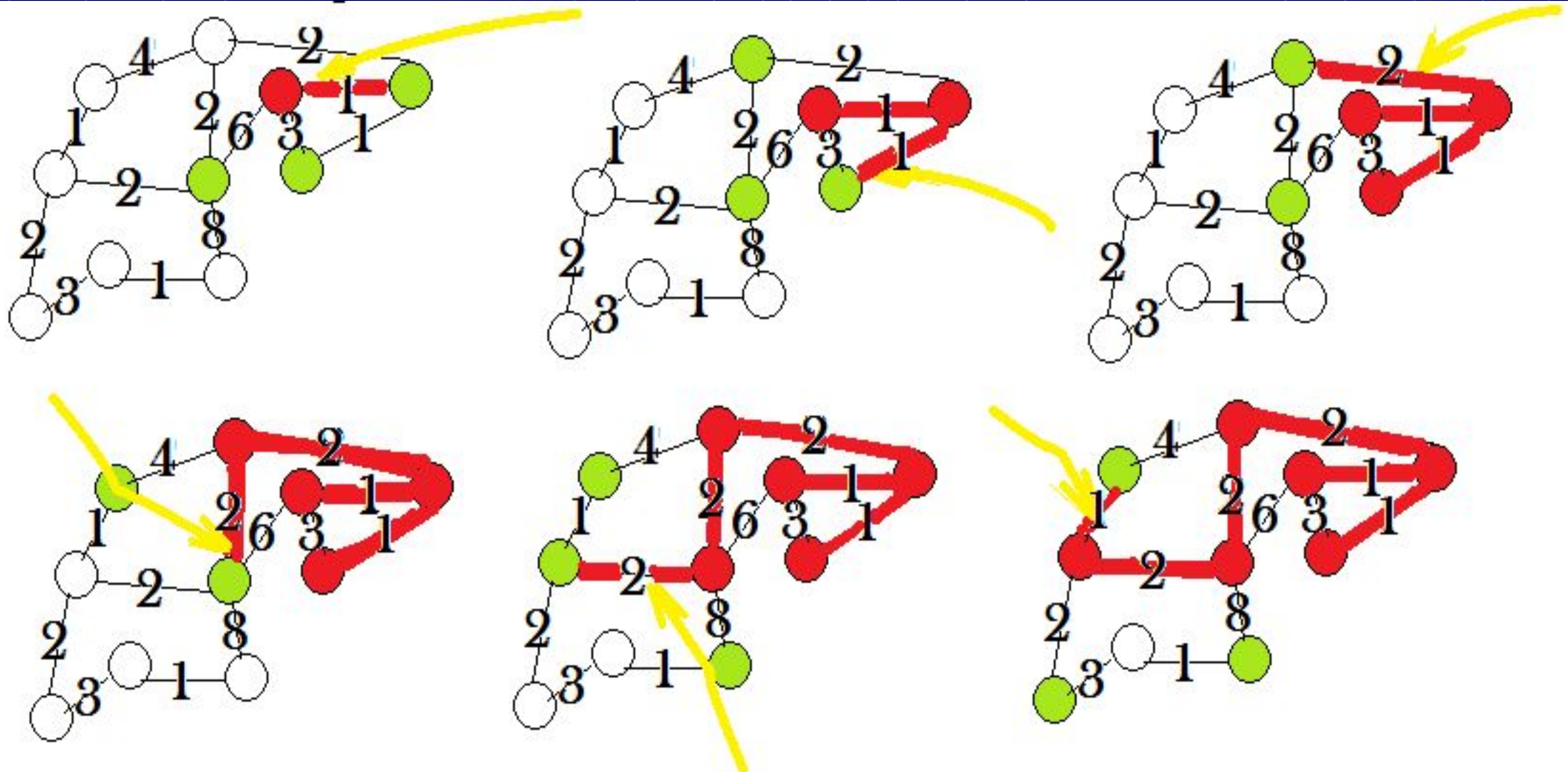
Задачи на графах – минимальное остовное дерево (алгоритм Прима – параллельная реализация)

1- выбирается произвольный начальный узел
2 – параллельно строится «кайма» - узлы, связанные с построенной частью дерева
3 – параллельно извлекается узел из «каймы» с минимальным связывающим ребром
4 - повторяется (2) – (3) пока есть не включенные в остов узлы

Сложность O(V)

Задачи на графах – минимальное остовное дерево (алгоритм Прима)

Красный – остов, зеленый – кайма, желтый – выбираемое ребро



Задачи на графах – поиск кратчайшего пути

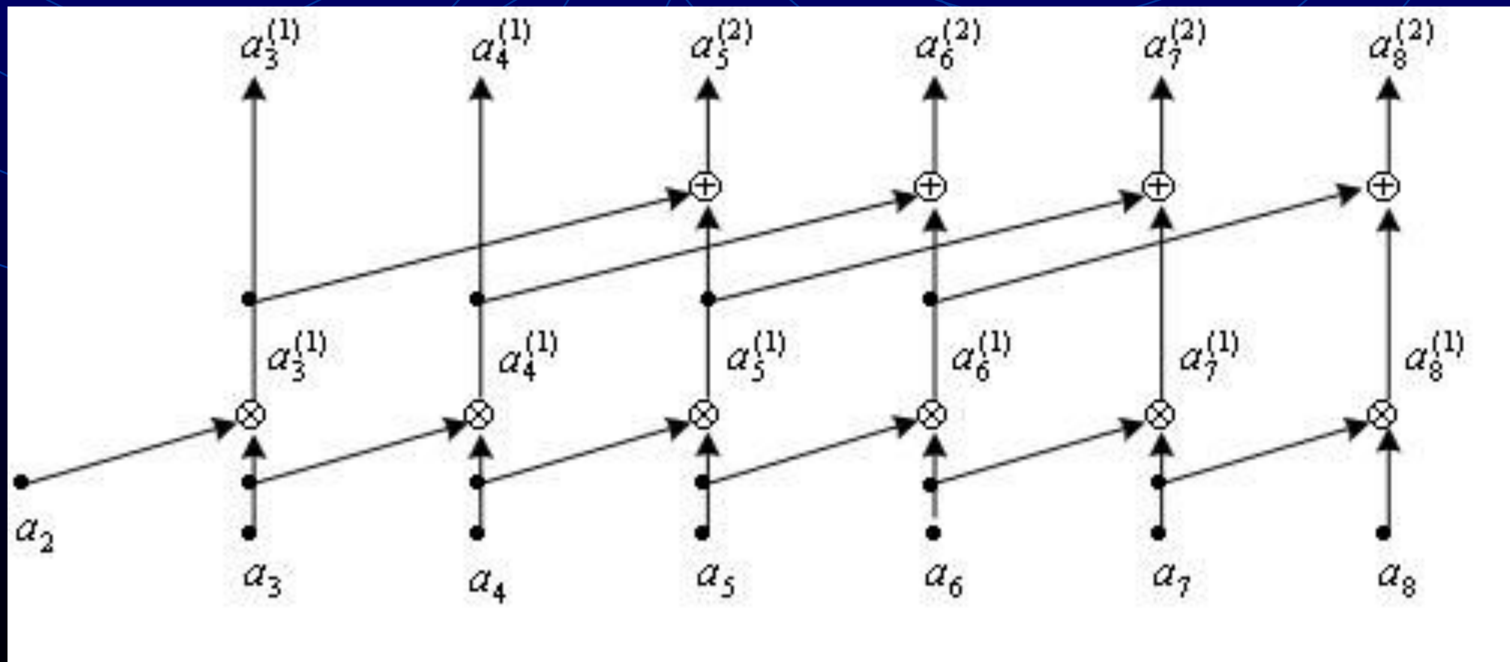
1 – строится матрица D[N][N] связности графа
2 – элемент d[i][j] – расстояние между узлами i и j
3 - D[N][N] * D[N][N] - расстояния длины 2
4 - D[N][N] * D[N][N] * D[N][N] - расстояния длины 3
ИТОГ: сводим задачу к задаче умножения матриц, которая решается параллельно

Параллельный алгоритм вычисления рекурсии

Реализация рекурсии – каскадные алгоритмы.

Пример – вычисление суммы

S[i] = S[i-2] + a[i] * a[i-1]



Проблемы компьютерного зрения

- Сегментация изображений – выделение областей, представляющие собой целые объекты или их крупные элементы
- Семантическое распознавание объектов на 2-D изображении
- Семантический анализ 3-D изображений, реконструкция формы 3-D изображений по их 2-D изображениям
- Обнаружение/распознавание/отслеживание объектов, обладающих определенными свойствами, на статическом изображении и в видеопотоке
- Распознавание и классификации изображений документов
- Автоматический поиск заданного объекта на изображении
- Автоматический подбор параметров для алгоритмов обработки изображений (например, параметры сегментации, выбора опорных точек и т.п.)

Потоки :
создание и барьеры

Создание потока

```
static HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpsa,
    //параметры защиты
   // NULL – атрибуты защиты по умолчанию
    DWORD dwStackSize,     // размер стека потока
    LPTHREAD_START_ROUTINE pfnThreadProc,
                    // функция  потока
    void* pvParam,      // передаваемые параметры
    DWORD dwCreationFlags,   // флаги потока
   // 0 – исполнение потока начнется немедленно
  // CREATE_SUSPENDED – поток задержан
    DWORD* pdwThreadId ) ;
          // указатель на  идентификатор потока
```

Пример – описание тела потока

```
 struct Param{int start, fin, indexResult;};
                              // передаваемые потоку параметры
int data[MAXN];   // суммируемый массив
// доступ по чтению  - нет гонки данных

DWORD WINAPI ProducerThread(LPVOID lpParam) {
                              //        тело потока:
Param * myParam = (Param*)lpParam;
                              // переданные потоку параметры
int indexSum = myParam->indexResult;
sum[indexSum] = 0;
for (int index=start; index<=fin; index++)
               sum[index] += data[indexSum];
}
```

Пример – создание двух потоков

```
int main (){
HANDLE    threadFirst,  threadSecond;
DWORD     dwNetThreadIdFirst, dwNetThreadIdSecond;
int len=MAXN; // длина суммируемого массива
param paramFirst={0,len/2,0},
          paramSecond={len/2+1,len,1];
threadFirst = CreateThread(NULL, 0, ProducerThread,
               &paramFirst, 0, &dwNetThreadIdFirst);
threadSecond = CreateThread(NULL, 0,ProducerThread,
               &paramSecond, 0, &dwNetThreadIdSecond);
// организуем барьер:
WaitForSingleObject (threadFirst, INFINITE);
WaitForSingleObject (threadSecond, INFINITE);
// после завершения потоков вычисляем результат:
int sumResult = sum[0] + sum[1];
return 0;
}
```

Барьер

Барьер - метод синхронизации, при помощи которого потоки из одного набора поддерживают взаимодействие. При помощи барьера поток из рабочего набора должен ждать завершения всех других потоков этого набора для того, чтобы получить возможность перейти к следующему шагу выполнения. Данный метод гарантирует, что ни один поток не пройдет за некую логическую точку выполнения до тех пор, пока все потоки не придут в эту логическую точку.

Барьер

WaitForMultipleObjects(k, threadProducer, true, INFINITE);

//ждем завершения всех потоков в

//массиве threadProducer,

Или

for(inti=0; i<k; i++) {

WaitForSingleObjects(threadProducer [i] , INFINITE);

//в цикле ждем завершения *i* – ого потока

}

Параллельное программирование в Visual Studio 2017

(Библиотека параллельных шаблонов PPL)

Возможности PPL

Предоставляет универсальные безопасные алгоритмы и контейнеры, работающие параллельно:

- ***Параллелизм задач*: работает поверх ThreadPool для выполнения нескольких рабочих задач**
- ***Параллельные алгоритмы*: универсальные алгоритмы, работающие с коллекциями данных параллельно**
- ***Параллельные контейнеры и объекты*: универсальные типы контейнеров, предоставляющие безопасный одновременный доступ к элементам**

Параллельные алгоритмы

- Алгоритм parallel_for
- Алгоритм parallel_for_each
- Алгоритм parallel_invoke
- Алгоритмы parallel_transform и parallel_reduce
- Алгоритм parallel_transform
- Алгоритм parallel_reduce
- Секции
- Параллельная Сортировка

Лабораторная работа 2 «Параллельные вычислительные алгоритмы»

1. Написать программу для последовательного алгоритма
2. Вычислить теоретическую временную сложность последовательного алгоритма , провести эксперименты
3. Предложить параллельный алгоритм решения задачи, оценить временную сложность и написать программу
4. Рассмотреть реализацию алгоритма с использованием пула потоков C++. (ОБЯЗАТЕЛЬНО ДЛЯ ПИ)
5. Вычислить временную сложность параллельного алгоритма
6. Провести эксперименты по определению влияния количества создаваемых потоков на скорость работы
7. Сделать выводы

Литература по дисциплине

1. Эхтер Ш., Робертс Дж. Многоядерное программирование. – СПб.: Питер. 2010
2. Рихтер Дж. Widows для профессионалов: создание эффективных Win-32 приложений с учетом специфики 64-разрядной версии Windows. - СПб.: Питер. 2008
3. Фленов М.Е. Программирование на C++ глазами хакера. - СПб.: БХВ-Петербург, 2009

Литература по предмету

4. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования
5. Фаулер М., Скотт К. UML. Основы
6. Крючкова Е.Н., Старолетов С.М. Методическое пособие