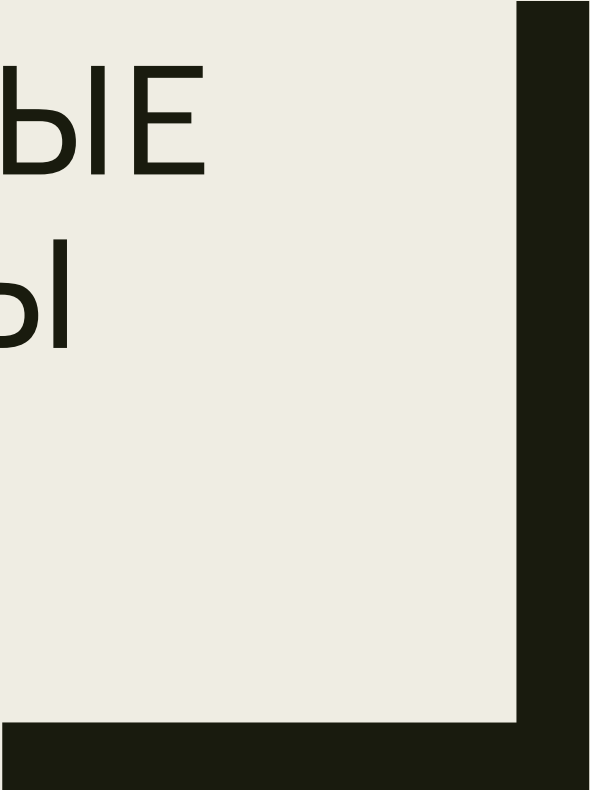




# ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ

Занятие 1. Технология OpenMP



Составитель: Герасимов А.С.

# Учебный кластер МФТИ

head.vdi.mipt.ru

remote.vdi.mipt.ru:52960

ssh login@head.vdi.mipt.ru

- Узлы: 1 головной (head) и 7 вычислительных
- Узлы идентичны: 4 ядра, 15 ГБ ОЗУ
- Система очередей – Torque/PBS

## Пример PBS-задачи

job.sh

```
#!/bin/bash
```

```
#PBS -l walltime=00:10:00,nodes=7:ppn=1
```

```
#PBS -N job_name
```

```
#PBS -q batch
```

```
uname -n
```

## Запуск задачи

```
qsub job.sh
```

### Выход задачи:

- `<job_name>.o<ID>` – выход stdout
- `<job_name>.e<ID>` – выход stderr

### Ограничения:

- 5 заданий / пользователя
- 10 минут выполнения
- 1 ГБ памяти

## Просмотр текущих задач в очереди

qstat

```
[kolya@head mpi]$ qstat
```

Job id	Name	User	Time Use	S	Queue
-----	-----	-----	-----	-	-----
25.localhost	my_job	kolya	0	R	batch
26.localhost	my_job	kolya	0	R	batch
27.localhost	my_job	kolya	0	R	batch
28.localhost	my_job	kolya	0	R	batch
29.localhost	my_job	kolya	0	R	batch

## Удаление задачи

qdel <ID>

# OpenMP

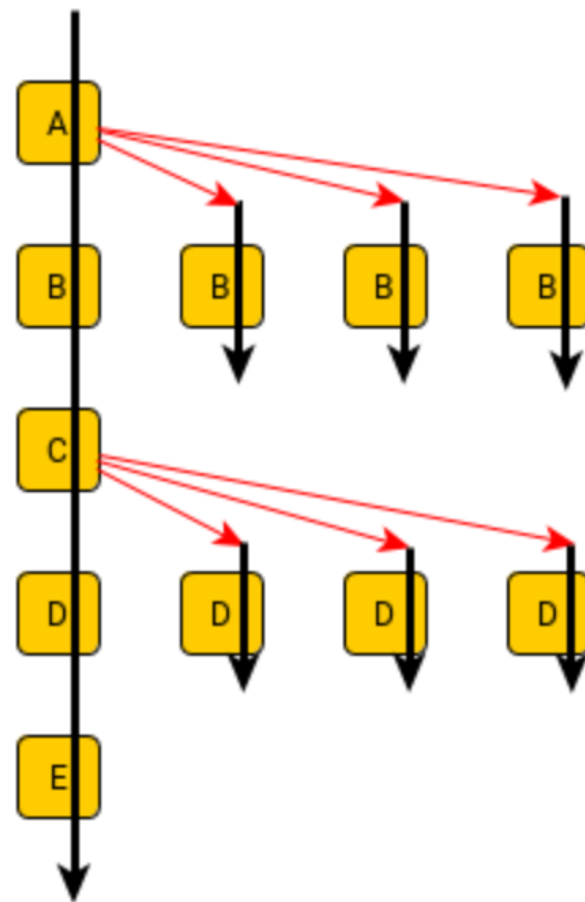
- Библиотека функций и директивы компилятора, предназначенные для написания параллельных программ
- Официально поддерживается *C, C++ и Фортран*, однако можно найти реализации для некоторых других языков, например *Паскаль* и *Java*.
- Поддерживается производителями аппаратуры (*Intel, HP, SGI, Sun, IBM*), разработчиками компиляторов (*Intel, Microsoft, KAI, PGI, PSR, APR, Absoft*)
- Ориентирован на системы с общей памятью
- Основной метод – создание потоков (модель fork-join)

# Общие механизмы OpenMP. Инициализация

## Основа программы

```
#include "omp.h"

int main(int argc, char** argv) {
    // A - single thread
    #pragma omp parallel
    {
        // B - many threads
    }
    // C - single thread
}
```



## Компиляция программы

```
gcc -fopenmp superhot.c -o hot
```

## Запуск программы

```
OMP_NUM_THREADS=N ./hot
```



# Общие механизмы OpenMP.

## Параллельные регионы

- Количество порождаемых потоков для параллельных областей контролируется через переменную окружения **OMP\_NUM\_THREADS**, а также может задаваться через вызов функции внутри программы.
- Каждый порожденный поток исполняет код в структурном блоке. По умолчанию синхронизация отсутствует и последовательность выполнения не определена.
- После выполнения параллельного участка все потоки, кроме основного, завершаются.
- Каждый поток имеет уникальный номер, который изменяется от 0 до количества потоков - 1. Идентификатор потока может быть определен с помощью функции **omp\_get\_thread\_num()**.

# Общие механизмы OpenMP.

## Параллельные регионы

```
#pragma omp parallel
{
  myid = omp_get_thread_num();
  if(myid == 0)
    do_something();
  else
    do_something_else(myid);
} ← sync
```

# Общие механизмы OpenMP. Задача 1

- Составить и запустить программу «Hello, world!»
- Вывести размер своего потока
- Потоки распечатывают свои идентификаторы в обратном порядке

# Общие механизмы OpenMP.

## УСЛОВИЯ ВЫПОЛНЕНИЯ

```
#pragma omp parallel \  
shared(var1, var2, ....) \  
private(var1, var2, ...) \  
firstprivate(var1, var2, ...) \  
reduction(оператор:var1, var2, ...) \  
if(выражение) \  
default(shared|none)  
{  
    // parallel block  
}
```

# Общие механизмы OpenMP. Условия выполнения

- **shared(var1, var2, ...)**
  - перечисленные переменные будут разделяться между потоками. Все потоки будут обращаться к одной и той же области памяти.
- **private(var1, var2, ...)**
  - каждый поток должен иметь свою копию переменной на всем протяжении своего исполнения.
- **firstprivate(var1, var2, ...)**
  - инициализируются при входе в параллельный участок кода значением, которое имела переменная до входа в параллельную секцию.
- **lastprivate(var1, var2, ...)**
  - сохраняют свое значение, которое они получили при достижении конца параллельного участка кода.
- **reduction(оператор:var1, var2, ...)**
  - гарантирует безопасное выполнение операций редукции, например, вычисление глобальной суммы.
- **if(выражение)**
  - параллельное выполнение необходимо только если выражение истинно.
- **default(shared | private | none)**
  - область видимости переменных внутри параллельного участка кода по умолчанию.
- **schedule(type[,chunk])**
  - контролируется то, как итерации цикла распределяются между потоками.

## Общие механизмы OpenMP. Условия выполнения

```
#pragma omp parallel shared(a) private(myid, x)
{
    myid = omp_get_thread_num();
    x = work(myid);
    if(x < 1.0)
        a[myid] = x;
}
```

```
#pragma omp parallel default(private) shared(a)
{
    myid = omp_get_thread_num();
    x = work(myid);
    if(x < 1.0)
        a[myid] = x;
}
```

## Общие механизмы OpenMP. Разделение работы

```
#pragma omp for [условие [,условие] ...]
```

```
#pragma omp parallel
```

```
{
```

```
#pragma omp for private(i) shared(a,b)
```

```
    for(i=0; i<10000; i++)
```

```
        a[i] = a[i] + b[i]
```

```
} ← sync
```

## Общие механизмы OpenMP. Разделение работы

**#pragma omp sections [условие [,условие...]]**

```
#pragma omp parallel sections (nowait?)
{
    #pragma omp section
    {
        printf("T%d: foo\n", omp_get_thread_num());
    }
    #pragma omp section
    {
        printf("T%d: bar\n", omp_get_thread_num());
    }
} // omp sections ← sync
```



## Общие механизмы OpenMP. Разделение работы

```
#pragma omp single [условие [, условие ...]]
```

```
#pragma omp single  
    printf("Program finished!\n");
```

## Общие механизмы OpenMP. Условия выполнения

### If(**clause**)

```
#pragma omp parallel
{
  #pragma omp for if(n>2000)
  {
    for(i=0; i<n; i++)
      a[i] = work(i);
  }
}
```

### lastprivate(**var** [, **var2** ...])

```
#pragma omp parallel
{
  #pragma omp for private(i) lastprivate(k)
  for(i=0; i<10; i++)
    k = i*i;
}
printf("k = %d\n", k);
```

## Общие механизмы OpenMP. Условия выполнения

### **reduction(op:var1 [, var2 ...])**

```
#pragma omp parallel
{
  #pragma for shared(x) private(i) reduction(+:sum)
    for(i=0; i<10000; i++)
      sum += x[i];
}
```

```
#pragma omp parallel
{
  #pragma for shared(x) private(i) reduction(min:gsum)
    for(i=0; i<10000; i++)
      gmin = min(gmin, x[i]);
}
```

## Общие механизмы OpenMP. Условия выполнения

**reduction(op:var1 [, var2 ...])**

C/C++

+, -, \*, &, ^, |, &&, ||, min, max

Fortran

+, -, \*, .and., .or., .eqv., .neqv., min, max, iand, ior, ieor

# Общие механизмы OpenMP. Условия выполнения

## `schedule(тип [, размер блока])`

Размер блока задает размер каждого пакета на обработку потоком (количество итераций).

Типы расписания:

- **static** – итерации равномерно распределяются по потокам. Т.е. если в цикле 1000 итераций и 4 потока, то один поток обрабатывает все итерации с 1 по 250, второй – с 251 по 500, третий - с 501 по 750, четвертый с 751 по 1000. Если при этом задан еще и размер блока, то все итерации блоками заданного размера циклически распределяются между потоками.

Статическое распределение работы эффективно, когда время выполнения итераций равно, или приблизительно равно.

- **dynamic** – работа распределяется пакетами заданного размера (по умолчанию размер равен 1) между потоками.

Как только какой-либо из потоков заканчивает обработку своей порции данных, он захватывает следующую.

В отличие от статического планирования, выполняется многократно (во время выполнения программы).

При этом подходе накладные расходы несколько больше, но можно добиться лучшей балансировки загрузки между потоками.

## Общие механизмы OpenMP. Условия выполнения

### `schedule(тип [, размер блока])`

- **guided** – данный тип распределения работы аналогичен предыдущему, однако размер блока изменяется динамически в зависимости от того, сколько необработанных итераций осталось. Размер блока постепенно уменьшается вплоть до указанного значения.

Распределение начинается с некоторого начального размера, зависящего от реализации библиотеки.

При таком подходе можно достичь хорошей балансировки при меньших накладных расходах.

- **runtime** – тип распределения определяется в момент выполнения программы. Удобно в экспериментальных целях для выбора оптимального значения **типа** и **размера блока**.

Тип распределения работ зависит от переменной окружения **OMP\_SCHEDULE**.

По умолчанию считается, что установлен статический метод распределения работ.

<https://stackoverflow.com/questions/42970700/openmp-dynamic-vs-guided-scheduling>

## Общие механизмы OpenMP. Задача 2

- Составить параллельную программу, суммирующую все натуральные числа от 1 до  $N$
- Каждый поток получает свой диапазон чисел для суммирования
- $N$  задается аргументом запуска
- Использовать условия `reduction`, `schedule`

# Общие процедуры MPI. Переменные окружения

Переменная	Назначение
<i>OMP_NUM_THREADS</i>	Количество потоков в параллельном блоке. По умолчанию количество потоков равно количеству виртуальных процессоров.
<i>OMP_SCHEDULE</i>	Тип распределения работ в параллельных циклах с типом runtime
<i>OMP_DYNAMIC</i>	Разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации.
<i>OMP_NESTED</i>	Разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию – запрещено.



# Общие механизмы OpenMP. Библиотечные функции

- `void omp_set_num_threads(int num_threads)`
  - задать количество потоков, которое может быть запрошено для параллельного блока
- `int omp_get_num_threads()`
  - получить количество потоков в текущей команде параллельных потоков
- `int omp_get_max_threads()`
  - получить максимальное количество потоков, которое может быть установлено
- `int omp_get_thread_num()`
  - получить номер потока в блоке
- `int omp_get_num_procs()`
  - получить количество физических процессоров, доступных программе
- `int omp_in_parallel()`
  - получить не нулевое значение, если вызвана внутри параллельного блока. В противном случае 0
- `void omp_set_dynamic(expr)`
  - Разрешает/запрещает динамическое выделение потоков
- `int omp_get_dynamic()`
  - получить, разрешено или запрещено динамическое выделение потоков
- `void omp_set_nested(expr)`
  - Разрешает/запрещает вложенный параллелизм
- `int omp_get_nested()`
  - получить, разрешен или запрещен вложенный параллелизм

## Общие механизмы OpenMP. Зависимость по данным

### Data dependency

```
for(i=1; i<8; i++)
```

```
    a[i] = c*a[i-1]; ← зависимость есть
```

```
for(i=1; i<9; i+=2)
```

```
    a[i] = c*a[i-1]; ← зависимости нет
```

#### ■ **Утверждение 1**

Только те переменные, в которые происходит запись на одной итерации и чтение их значения на другой, создают зависимость по данным.

#### ■ **Утверждение 2**

Только разделяемые переменные могут создавать зависимость по данным.

#### ■ **Следствие**

Если переменная не объявлена как приватная, она может оказаться разделяемой.

## Общие механизмы OpenMP. Зависимость по данным

### Function calls

```
double foo(double *a, double *b, int i) {  
    return 0.345*(a[i] + b[2*i]*C); ← зависимость есть  
}
```

```
double bar(double a, double b) {  
    return 0.345*(a + b*C); ← зависимости нет  
}
```

- Функцию необходимо сделать независимой от внешних данных, кроме как от значения параметров.
- В функции так же не должно быть статических переменных (**static**).

## Общие механизмы OpenMP. Зависимость по данным

### Nested loops

```
for(k=0; k<N; k++) ← зависимость есть  
    for(i=0; i<N; i++)  
        for(j=0; j<N; j++)  
            a[i][j] += b[i][k]*c[k,j];
```

```
for(i=0; i<N; i++)  
    for(j=0; j<N; j++)  
        for(k=0; k<N; k++) ← зависимости нет  
            a[i][j] += b[i][k]*c[k,j];
```

- Циклы, в которых есть **выход по условию**, не должны подвергаться распараллеливанию

## Общие механизмы OpenMP. Средства синхронизации

### **#pragma omp critical [(name)]**

```
#pragma omp for private(i) shared(a,xmax)
for(i=0; i<N; i++) {
    #pragma omp critical (xmax)
    {
        if(a[i]>xmax)
            xmax = a[i];
    }
}
```

- Правилom хорошего тона считается, если критическая секция содержит обращения только к одному разделяемому ресурсу.
- Все безымянные секции рассматриваются как одна (очень большая), и если не давать им явно имена, то только в одной из этих секций в один момент времени будет один поток - остальные будут ждать.

## Общие механизмы OpenMP. Средства синхронизации

### #pragma omp barrier

```
#pragma omp parallel
{
    #pragma omp atomic
    value++;
    #pragma omp barrier
    #pragma omp critical (cout)
    {
        std::cout << value << std::endl;
    }
}
```

## Общие механизмы OpenMP. Средства синхронизации

### #pragma omp ordered

```
#pragma omp parallel private(myid)
{
    myid = omp_get_thread_num();
    #pragma omp for private(i)
        for(i=0; i<8; i++)
            #pragma omp ordered
                printf("T%d: %d\n", myid, i);
}
```

T0: 0

T0: 1

T0: 2

T0: 3

T1: 4

...

## Общие механизмы OpenMP. Средства синхронизации

```
#pragma omp flush(var1[, var2, ...])
```

- Осуществляет немедленный сброс значений разделяемых переменных в память.
- Таким образом гарантируется, что во всех потоках значение переменной будет одинаковое.
- Неявно присутствует в директивах: barrier, начале и конце критических секций, параллельных циклов, параллельных областей, single секций.
- С ее помощью можно посылать сигналы потоком используя переменную как семафор. Когда поток видит, что значение разделяемой переменной изменилось, это говорит о том, что произошло событие и можно продолжить выполнение программы далее.



# Общие механизмы OpenMP. Задача 3

- Написать программу, в которой объявлен массив из 16000 элементов и инициализирован так, что значение элемента массива равно его порядковому номеру.
- Затем создайте новый массив, в котором будут храниться средние значения исходного массива:

$$b[i] = (a[i-1] + a[i] + a[i+1])/3.0$$

- Изменить исходный массив (не создавая новый). Средние значения записываются в центральный элемент «окна»:

$$a[i] = (a[i-1] + a[i] + a[i+1])/3.0$$

# Общие процедуры MPI. Задача 4

- Составить параллельную программу, перемножающую матрицы, значения которых заданы случайными числами.
- Адаптировать программу для перемножения больших матриц ( $>1000 \times 1000$ ). Использовать методы динамического распределения нагрузки, проанализировать эффективность.

## Общие механизмы OpenMP. Расширенные возможности

```
#pragma omp threadprivate(var1[, var2 ...])
```

- Позволяет один раз объявить приватную переменную для всех параллельных секций в рамках одного файла.
- Переменная должна быть объявлена как статическая
- Директива threadprivate должна присутствовать до объявления первой параллельной секции
- Количество потоков в программе должно быть постоянным

# Общие механизмы OpenMP. Расширенные возможности

- `void omp_init_lock(omp_lock_t *lock)`
  - инициализирует блокировку и связывает ее с параметром *lock*
- `void omp_destroy_lock(omp_lock_t *lock)`
  - деинициализирует переменную, связанную с параметром *lock*
- `void omp_set_lock(omp_lock_t *lock)`
  - блокирует выполнение потока до тех пор, пока блокировка на переменную *lock* не станет доступной.
- `void omp_unset_lock(omp_lock_t *lock)`
  - снимает блокировку с переменной *lock*
- `void omp_test_lock(omp_lock_t *lock)`
  - пытается установить блокировку и если операция выполнена успешно, возвращает ненулевое значение
  - в противном случае возвращается ноль
  - функция не блокирующая

# Общие механизмы OpenMP. Расширенные возможности

## Locks

```
#include <omp.h>
int main() {
    omp_lock_t lock;
    int i, p_sum = 0, res = 0;
    omp_init_lock(&lock);
    #pragma omp parallel firstprivate(p_sum)
    {
        #pragma parallel for private(i)
        for(i=0; i<100000; i++)
            p_sum +=i;
        omp_set_lock(&lock);
        res += p_sum;
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
    printf("%d\n", res);
}
```

# Общие механизмы OpenMP. Задача 5

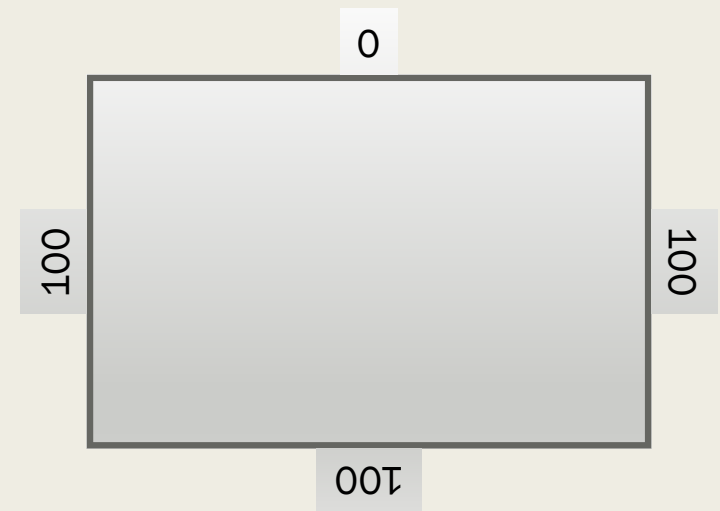
- Составить параллельную программу, вычисляющую сумму ряда (вычисление экспоненты)
- Число слагаемых задается аргументом запуска
- Вычисление более сложного ряда

# Общие механизмы OpenMP. Задача 6

- Составить параллельную программу, вычисляющую количество простых чисел от 1 до  $N$
- $N$  задается аргументом запуска
- Не используя простой перебор = используя оптимизированные способы поиска (с обоснованием)

# Общие механизмы OpenMP. Задача 7

- Составить параллельную программу, решающую уравнение теплоты установившегося состояния прямоугольной области для заданной точности  $\varepsilon$ .
- Размер области:  $M * N$
- Начальные условия  $\rightarrow$
- $T_c \leq \frac{T_N + T_S + T_W + T_E}{4}$
- Подсчет времени
- Вывод изменения на каждой итерации





# Общие механизмы OpenMP. Задача 8

- Составить параллельную программу, реализующую механизм быстрой сортировки (Quick Sort).
- Добавить возможность ввода массива через файл
- Добавить возможность ввода массива вручную
- Модифицировать исходный массив, не создавая новых (ограничение по памяти)
- Учесть граничные условия!

# Общие механизмы OpenMP. Задача 9

- Рассмотреть программы с багами.
  - Найти и исправить ошибки.
  - Предоставить рабочий вариант программ.
- 
- *Коды программ находятся в каталоге с презентацией в папке `bugged_progs`.*

# Задача 10

- Рассматривается краевая задача вида

$$y'' - f(x, y) = 0, y(0) = a, y(1) = b.$$

- В 1924 году Б. Нумеров для нелинейных функций, не зависящих от первой производной решения, предложил «компактную» аппроксимацию четвертого порядка. Она имеет вид

$$\frac{y_{m+1} - 2y_m + y_{m-1}}{h^2} = f_m + \frac{1}{12} (f_{m+1} - 2f_m + f_{m-1})$$

- Реализация граничных условий трудностей не вызывает. С помощью аппроксимации Нумерова с линеаризацией по Ньютону и последующей заменой прогонки редукционным алгоритмом решить следующие краевые задачи.

# Задача 10

- **В1 (слабый)**  $y'' - e^y = 0, y(0) = 1, y(1) = b, b$  меняется от 0 до 1 с шагом 0,1. Разбить на 4 исполнителя, количество точек на отрезке от 400 до 4000
- **В2 (средний)**  $y'' - e^{-y} = 0, y(0) = 1, y(1) = b, b$  меняется от 0 до 1 с шагом 0,1. Разбить на 4 исполнителя, количество точек на отрезке от 400 до 4000
  - Подробно исследовать окрестности  $b = 1.5$ . Что происходит в этой окрестности?
- **В3 (средний)**  $y'' - a(y - y^3) = 0, y(-10) = y(10) = \sqrt{2}, a$  меняется от 100 до 1000000, Разбить на несколько исполнителей, количество точек на каждого исполнителя от 400 до 4000, условие на пространственный шаг -  $h \ll \frac{1}{\sqrt{a}}$
- **В4 (тяжелый)**  $y'' - a(y^3 - y) = 0, y(-10) = y(10) = \sqrt{2}, a$  меняется от 100 до 1000000, Разбить на несколько исполнителей, количество точек на каждого исполнителя от 400 до 4000, условие на пространственный шаг -  $h \ll \frac{1}{\sqrt{a}}$
- **В5 (средний)**  $y'' - a(y - |x|)^5 = 0, y(0) = 1, y(1) = b, a$  меняется от 100 до 10000. Разбить на 4 исполнителя, количество точек на отрезке от 400 до 4000
  - Как можно усовершенствовать разбиение области на зоны ответственности, если использовать априорную информацию о существовании в окрестности нуля внутреннего погранслоя?