**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
 РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ**

Лабораторная работа 4

«Параллельная реализация метода Якоби в трехмерной области»

Студента 2 курса, 19210 группы

**Пирожков Андрей Константинович**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

П.О.Холявко

Новосибирск 2021

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc70030779)

[ЗАДАНИЕ 3](#_Toc70030780)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc70030781)

[ЗАКЛЮЧЕНИЕ 11](#_Toc70030782)

[Приложение *Листинг файла lab4.c* 13](#_Toc70030783)

# ЦЕЛЬ

* Практическое освоение методов распараллеливания численных алгоритмов на регулярных сетках на примере реализации метода Якоби в трехмерной области
* Проанализировать полученную программу и выявить зависимость производительности программы от входных данных, варианта решения, количество ядер

# ЗАДАНИЕ

1. Написать параллельную программу на языке C/C++ с использованием MPI, реализующую решение уравнения (1) методом Якоби в трехмерной области в случае одномерной декомпозиции области. Уделить внимание тому, чтобы обмены граничными значениями подобластей выполнялись на фоне счета.
2. Измерить время работы программы при использовании различного числа процессорных ядер: 1, 2, 4, 8, 16. Размеры сетки и порог сходимости подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер. По профилю убедиться, что коммуникации происходят на фоне счета

# **ОПИСАНИЕ РАБОТЫ**

Программу решил реализовывать на языке Си стандарта 99-ого года.

Традиционно в начале я писал программу без MPI. Мне нужно было убедиться, что она работает правильно и потом с помощью неё работать с многоядерной программой и сравнивать результаты обоих.

Сначала в программе я объявляю все исходные переменные, которые будут задействованы. Затем я их инициализирую функцией («»), которая присваивает каждой переменной своё значение. Также я решил указать размер сетки в аргументах программы, чтобы в дальнейшем было удобнее тестировать её. И порог сходимости тоже, но только обратной величиной (опять же для удобства). Также я пересчитываю некоторые выражения (константы) заранее, которые возможно, чтобы не тратить время в каждой итерации цикла на их же пересчёт.

Как только все значения переменных известны, я должен задать краевые значения для нашей функции. Для этого тоже есть специальная функция «». Заполняю сначала всё нулями функцией «». А после заполняю каждую грань отдельным циклом. Так должно быть быстрее, чем просто на каждой итерации цикла проверять, а есть ли это край или нет.

Затем начинается основной алгоритм, а также начинается работа таймера. Здесь происходит асинхронная передача границ подблоков. Для это я создал 4 «», каждый из которых отвечает за отправку и получения верхней или нижней границы. Эти границы хранятся в двух массивах: «» и «», названия которых указывает, какая это есть граница.

В то время как они обмениваются, происходит расчёт внутри в функции «». То есть пересчитывается функция везде, кроме верхней и нижней границы каждого блока, так как для подсчёта ей нужны те самые границы, которые в фоне записываются в буфер.

После пересчёта функции внутри, проверяется, записались ли наши границы в буфер. Если они ещё не успели записаться, то ждём, иначе начинаем считать значения функции на границах в «».

В ходе пересчёта функции мы везде проверяем критерий завершённости, и если он сработает, то значение переменной «» станет единичкой (вместо нуля). А с помощью функции «» мы соберём со всех процессов результаты. Если же критерий завершённости сработает, то завершаем алгоритм, иначе идём на следующую итерацию и повторяем пересчёт сначала.

Стоит отметить, что у меня условно 2 массива со значениями функции. Они каждый раз заменяют друг друга и каждый раз кто-то из них предыдущий, а кто-то текущий. Для этих целей есть функция «» и переменные «» и «». В переменных записаны 0 и 1 соответственно. 0 – предыдущий, 1 – текущий. В конце каждой итерации они меняются значением с помощью той функции. Таким образом мы избегаем постоянного переписывания данных в буфер. В памяти, конечно, не выигрываем, а вот в скорости есть выигрыш.

В конце программы мы должны проверить так называемую оценку точности полученного решения. В данном случае она служит как некоторый индикатор корректного счёта. Это очень удобно, когда тестируешь программу на многих процессах. И если значения отличаются, то это может значить только одно: где-то идёт ошибка в вычислениях. Также для этих целей придуман подсчёт итераций. Тоже неплохой индикатор. Обычно если всё плохо, то он вообще равен 1 или 2, вместо 20 и более.

В самом конце программы мы вычисляем время и все необходимые данные выводим. Потом чистим указатели.

Программа компилировалась на intel компиляторе (icc). Его я использую так как он быстрее чем стандартный gcc.

Таблица с компиляцией программы:

|  |  |  |
| --- | --- | --- |
| Файл | Описание | Команда компиляции |
| [lab4.c](#_Приложение_1) | Код программы | mpiicc lab4.c -std=gnu99 -o lab4.exe |

Пояснение к команде компиляции:

* -std=gnu99 – используемый мною стандарт языка си
* - mpiicc – компилятор mpi совместно с интеловским компилятором icc

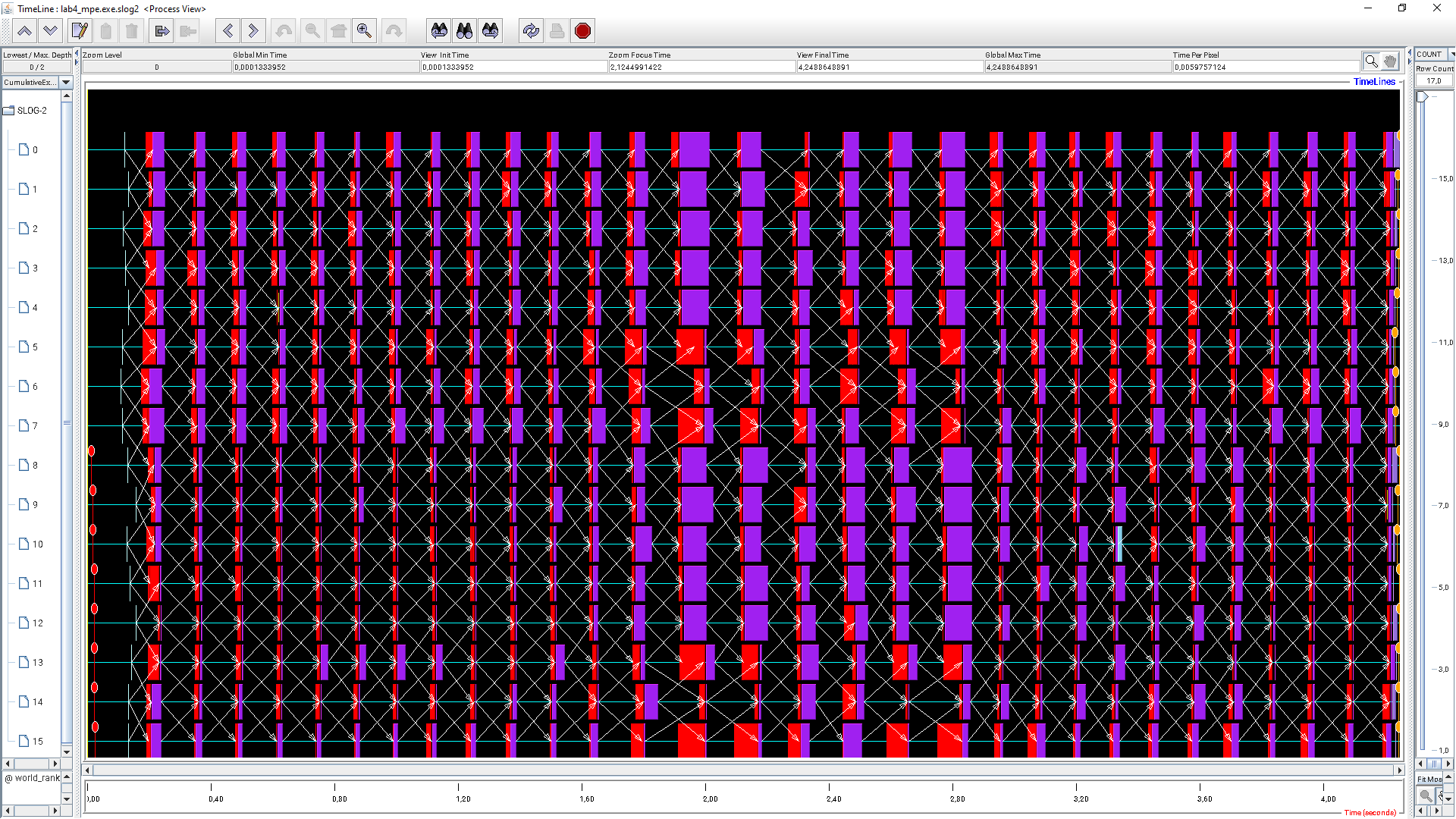
Таблица с результатами при

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество ядер | без MPI | 1 | 2 | 4 | 8 | 16 |
| Время | 37,05203 | 37,09519 | 18,69444 | 9,613881 | 5,099618 | 2,668528 |
| Ускорение | 1 | 0,998837 | 1,981981 | 3,854013 | 7,265648 | 13,88482 |
| Эффективность | 1 | 0,998837 | 0,990991 | 0,963503 | 0,908206 | 0,867801 |

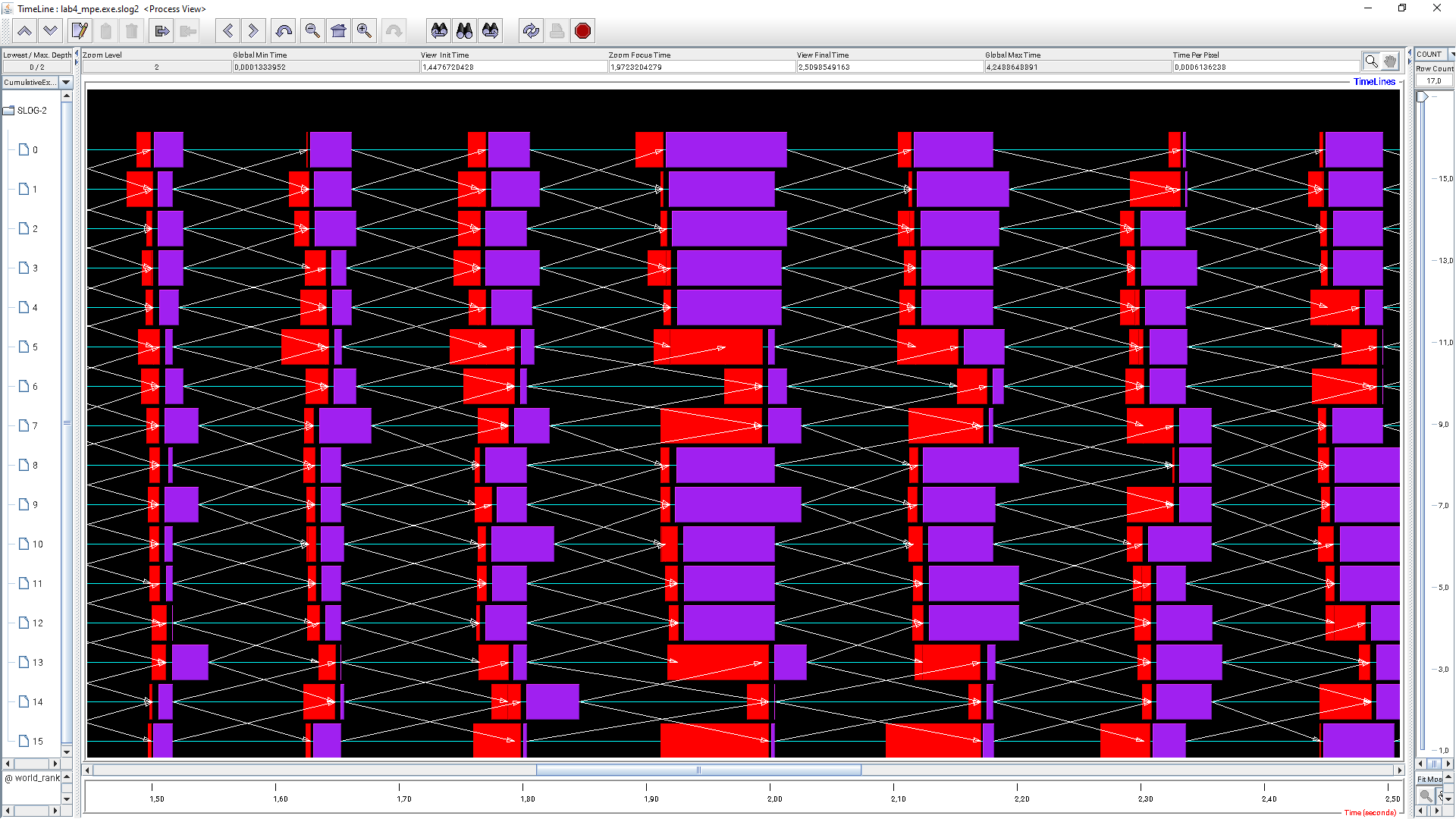
Результаты получилось очень даже неплохими, потому что ускорение достаточно близко к тем значениям, которое соответствует количеству процессов запускаемой нами программы. Эффективность до 4 ядер практически везде составила единицу (на графике показана сбоку шкала, она начинается от 0,8), только сильно просев на 8 и 16 ядрах. Обычно выше, чем 0,5 на предыдущих лабораторных работах выжать не получалось, а тут сразу 0,86.

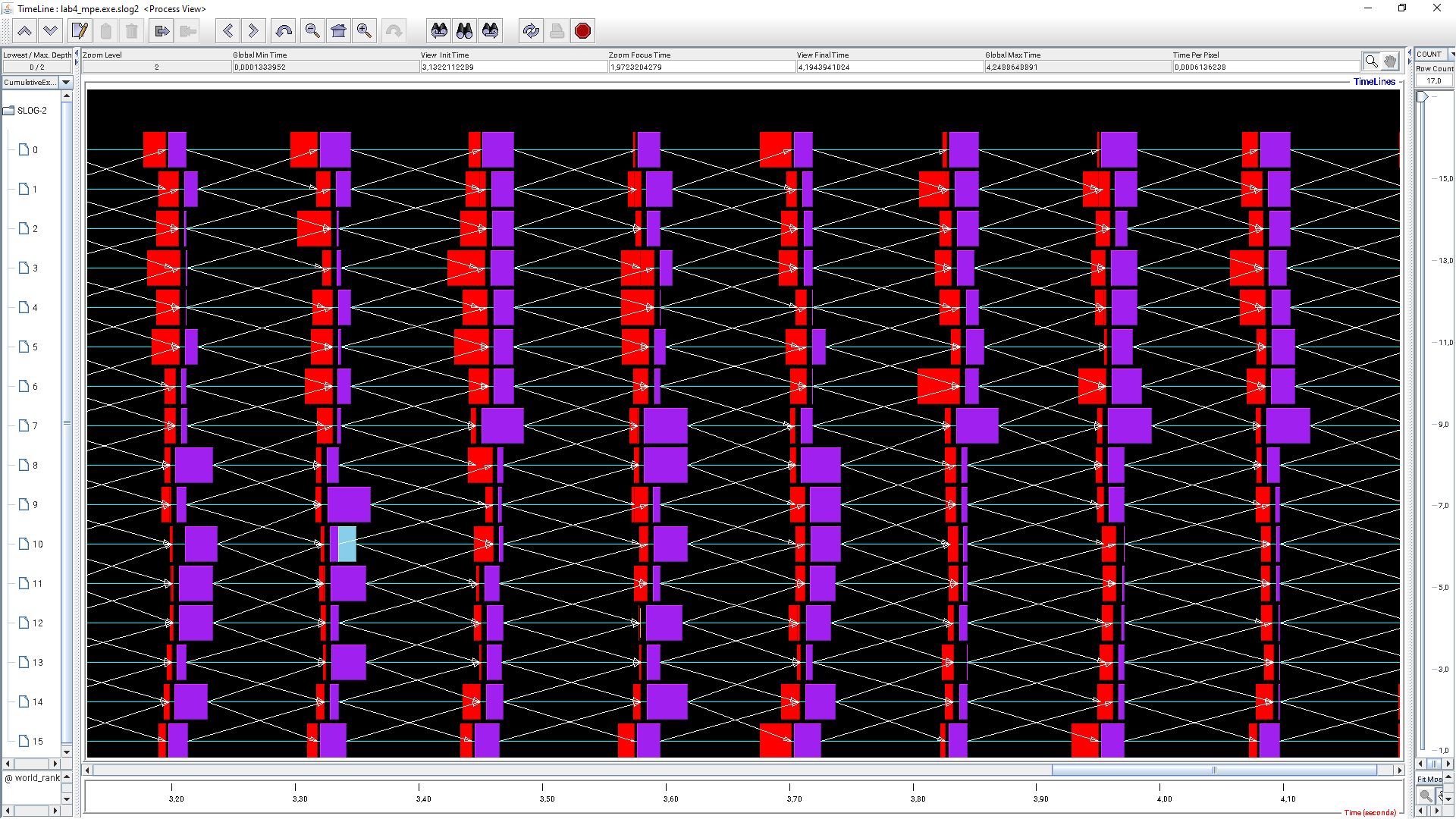
Перейдем теперь к графикам, которые у меня получились:

Теперь скриншоты из профилировщика:



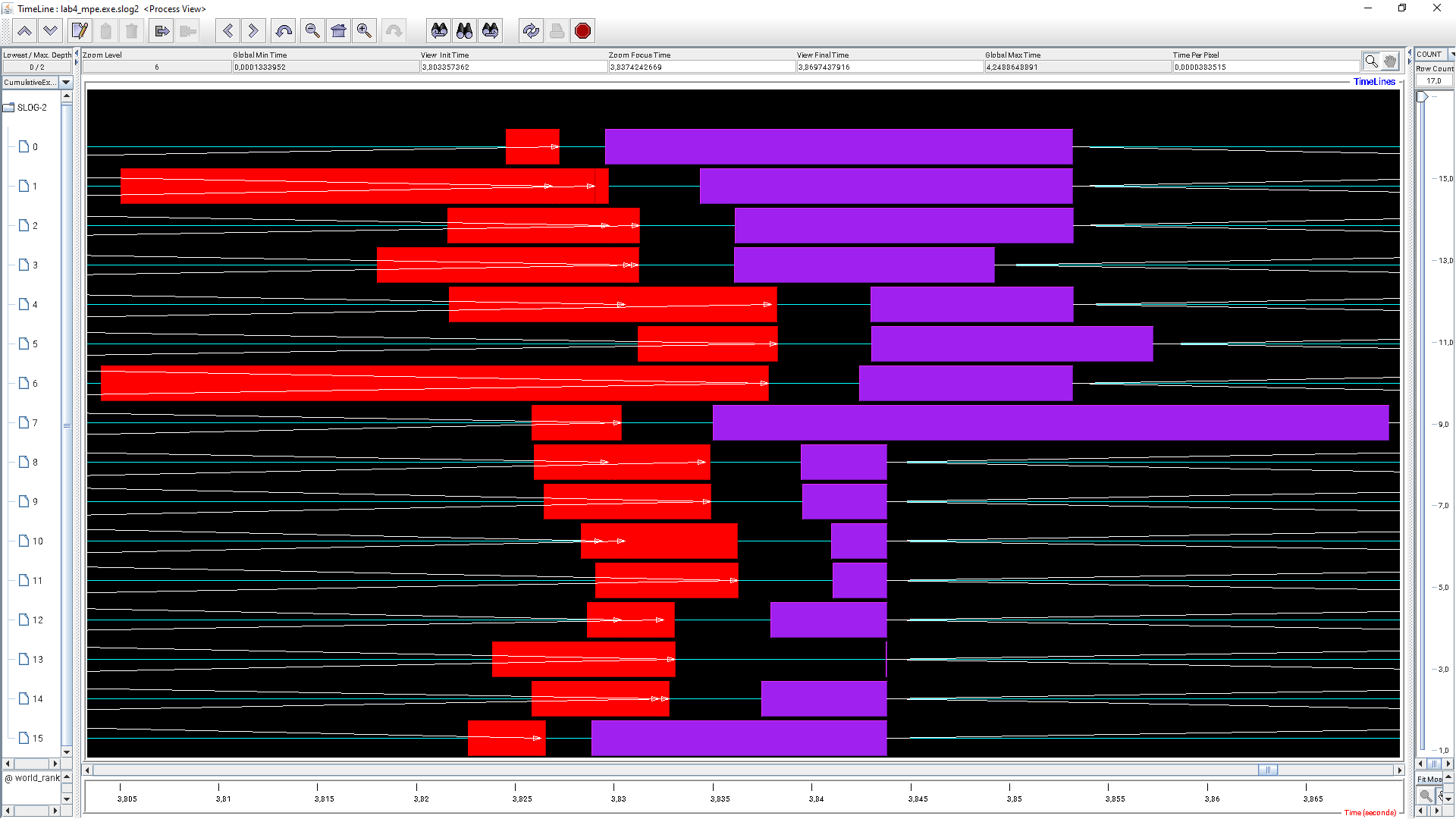
Это общий вид. Тут даже можно посчитать количество итераций – 30. Если посмотреть в целом, то в большинстве случаев всё хорошо, однако местами, особенно в центре, произошли какие-то небольшие сбои.

А теперь немного приближенный скриншот из места, где происходят какие-то заминки: 

Теперь того же масштаба, но немного в другом месте, где нет таких заминок: 

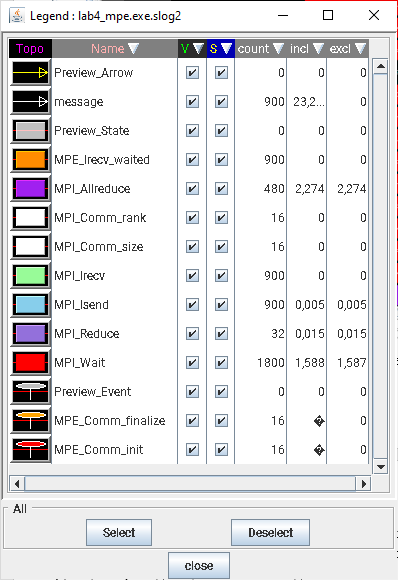
Кстати тут видно, что по каким-то причинам задержалась отправка границы блока.

Ещё ближе скриншот:



Ничего такого информативного здесь не увидеть, наверное.

Теперь сама легенда:

Ну стрелочки просто показывают куда у нас отправляются блоки границ. Красный это ожидание. Фиолетовый это сбор переменных, которые отвечают за критерий завершенности. Странно, что они так долго работают, видимо во время их работы, ещё и засчитывается время работы пересчёта границ.

Интересно, что последние 8 процессов одновременно завершили «». Возможно, это как-то связано с тем, что работают два компьютера, а не один, поэтому происходят такие разбросы.

# **ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы я изучил несколько новые возможности MPI, а именно асинхронные функции. Также я проанализировал полученные графики. И по ним можно сделать вывод о том, что наблюдается хороший рост производительности с увеличением количества процессов, высчитывающих функцию.

# Приложение 1

*Листинг файла lab4.c*

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#include <math.h>

#include <limits.h>

#include <time.h>

#include "mpi.h"

//Эта штука просто меняет предыдущий массив с текущим и наоборот (F[m] F[m+1 и наоборот])

void swapping(int\* swap1, int\* swap2)

{

int temp = \*swap1;

\*swap1 = \*swap2;

\*swap2 = temp;

}

//Функция fi(x,y,z)

double fi(int i, int j, int k, double h\_i, double h\_j, double h\_k, double I\_0, double J\_0, double K\_0)

{

double x = I\_0 + i \* h\_i;

double y = J\_0 + j \* h\_j;

double z = K\_0 + k \* h\_k;

return (double)(x \* x + y \* y + z \* z);

}

//Функция ro(x,y,z)

double ro(int a, int i, int j, int k, double h\_i, double h\_j, double h\_k, double I\_0, double J\_0, double K\_0)

{

return (6 - a \* fi(i, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0));

}

//Оценка точности полученного решения

double accuracy(double\*\* F, int swap, int block, int rank, int a, int N\_i, int N\_j, int N\_k, double h\_i, double h\_j, double h\_k, double I\_0, double J\_0, double K\_0)

{

double max = (double)INT\_MIN;

double c;

for (int i = 1; i < block - 1; i++)

{

int index\_I = i \* N\_j \* N\_k;

int J\_K = N\_j \* N\_k;

for (int j = 1; j < N\_j - 1; j++)

{

int index\_J = j \* N\_k;

for (int k = 1; k < N\_k - 1; k++)

{

c = fabs(F[swap][index\_I + index\_J + k] - fi(i + (block \* rank), j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0));

if (max < c)

{

max = c;

}

}

}

}

return max;

}

//Проверка на критерий завершённости

void check(double\*\* F, int swap1, int swap2, int\* complete, double e, int index\_I, int index\_J, int k)

{

if (fabs(F[swap2][index\_I + index\_J + k] - F[swap1][index\_I + index\_J + k]) < e)

{

\*complete = 1;

}

}

//Итерация - вычисление большой формулы (по краям)

void iteration\_edge(double\*\* F, double\* up, double\* down, int block, int rank, int size, int swap1, int swap2, int\* complete, double e, int N\_i, int N\_j, int N\_k, double coefficient, int a, double h\_i, double h\_j, double h\_k, double h\_i\_2, double h\_j\_2, double h\_k\_2, double I\_0, double J\_0, double K\_0)

{

double F\_i, F\_j, F\_k;

for (int j = 1; j < N\_j - 1; j++)

{

for (int k = 1; k < N\_k - 1; k++)

{

if (rank != 0)

{

int i = 0;

int index\_I = i \* N\_j \* N\_k;

int J\_K = N\_j \* N\_k;

int index\_J = j \* N\_k;

int I = i + block \* rank;

F\_i = (F[swap1][(index\_I + index\_J + k) + J\_K] + down[/\*index\_I\*/ + index\_J + k]) / h\_i\_2;

F\_j = (F[swap1][(index\_I + index\_J + k) + N\_k] + F[swap1][(index\_I + index\_J + k) - N\_k]) / h\_j\_2;

F\_k = (F[swap1][(index\_I + index\_J + k) + 1] + F[swap1][(index\_I + index\_J + k) - 1]) / h\_k\_2;

F[swap2][index\_I + index\_J + k] = coefficient \* (F\_i + F\_j + F\_k - ro(a, I, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0));

check(F, swap1, swap2, complete, e, index\_I, index\_J, k);

}

if (rank != size - 1)

{

int i = block - 1;

int index\_I = i \* N\_j \* N\_k;

int J\_K = N\_j \* N\_k;

int I = i + block \* rank;

int index\_J = j \* N\_k;

F\_i = (up[/\*index\_I\*/ + index\_J + k] + F[swap1][(index\_I + index\_J + k) - J\_K]) / h\_i\_2;

F\_j = (F[swap1][(index\_I + index\_J + k) + N\_k] + F[swap1][(index\_I + index\_J + k) - N\_k]) / h\_j\_2;

F\_k = (F[swap1][(index\_I + index\_J + k) + 1] + F[swap1][(index\_I + index\_J + k) - 1]) / h\_k\_2;

F[swap2][index\_I + index\_J + k] = coefficient \* (F\_i + F\_j + F\_k - ro(a, I, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0));

check(F, swap1, swap2, complete, e, index\_I, index\_J, k);

}

}

}

}

//Итерация - вычисление большой формулы (внутри)

void iteration\_in(double\*\* F, int block, int rank, int swap1, int swap2, int\* complete, double e, int N\_i, int N\_j, int N\_k, double coefficient, int a, double h\_i, double h\_j, double h\_k, double h\_i\_2, double h\_j\_2, double h\_k\_2, double I\_0, double J\_0, double K\_0)

{

double F\_i, F\_j, F\_k;

for (int i = 1; i < block - 1; i++) //всего узлов N\_i, т.е. с 0 до N\_i-1 => считаем с 1 до N\_i-2

{

int index\_I = i \* N\_j \* N\_k;

int J\_K = N\_j \* N\_k;

int I = i + block \* rank;

for (int j = 1; j < N\_j - 1; j++)

{

int index\_J = j \* N\_k;

for (int k = 1; k < N\_k - 1; k++)

{

F\_i = (F[swap1][(index\_I + index\_J + k) + J\_K] + F[swap1][(index\_I + index\_J + k) - J\_K]) / h\_i\_2;

F\_j = (F[swap1][(index\_I + index\_J + k) + N\_k] + F[swap1][(index\_I + index\_J + k) - N\_k]) / h\_j\_2;

F\_k = (F[swap1][(index\_I + index\_J + k) + 1] + F[swap1][(index\_I + index\_J + k) - 1]) / h\_k\_2;

F[swap2][index\_I + index\_J + k] = coefficient \* (F\_i + F\_j + F\_k - ro(a, I, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0));

//printf("F[%d][%d][%d] = %lf\t;%d\n", I, j, k, F[swap2][index\_I + index\_J + k], (rank \* block \* N\_j \* N\_k) + index\_I + index\_J + k);

//printf("F[%d][%d][%d] = %lf\t;%d\n", (rank \* block) + I, j, k, F[0][index\_I + index\_J + k], (rank \* block \* N\_j \* N\_k) + index\_I + index\_J + k);

check(F, swap1, swap2, complete, e, index\_I, index\_J, k);

}

}

}

}

//Инициализируем массив-функцию F - задаём краевые условия

void init\_F(double\*\* F, int block, int rank, int size, int N\_i, int N\_j, int N\_k, double h\_i, double h\_j, double h\_k, double I\_0, double J\_0, double K\_0)

{

memset(F[0], 0.0, block \* N\_j \* N\_k \* sizeof(double));

memset(F[1], 0.0, block \* N\_j \* N\_k \* sizeof(double));

if (rank == 0)

{

for (int j = 1; j < N\_j - 1; j++) //заполняем самый низ и вверх

{

for (int k = 1; k < N\_k - 1; k++)

{

F[0][j \* N\_k + k] = fi(0, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0); //для самого низа

F[1][j \* N\_k + k] = fi(0, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

//printf("F[%d][%d][%d] = %lf\t;%d\n", 0, j, k, F[0][j \* N\_k + k], j \* N\_k + k);

}

}

}

if (rank == size - 1)

{

for (int j = 1; j < N\_j - 1; j++) //заполняем самый низ и вверх

{

int index = (block - 1) \* N\_j \* N\_k; //для самого верха

for (int k = 1; k < N\_k - 1; k++)

{

F[0][index + j \* N\_k + k] = fi(N\_i - 1, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[1][index + j \* N\_k + k] = fi(N\_i - 1, j, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

//printf("F[%d][%d][%d] = %lf\t;%d\n", (rank \* block) + (block - 1), j, k, F[0][index + j \* N\_k + k], (rank \* block \* N\_j \* N\_k) + index + j \* N\_k + k);

}

}

}

for (int i = 0; i < block; i++) //заполняем перед и зад

{

int index = i \* N\_j \* N\_k; //для фронтальной части

int index\_ = index + N\_k \* (N\_j - 1); //для задней части

for (int k = 0; k < N\_k; k++)

{

F[0][index + k] = fi(i + block \* rank, 0, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[1][index + k] = fi(i + block \* rank, 0, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[0][index\_ + k] = fi(i + block \* rank, N\_j - 1, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[1][index\_ + k] = fi(i + block \* rank, N\_j - 1, k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

//printf("F[%d][%d][%d] = %lf\t;%d\n", i + (rank \* block), 0, k, F[0][index + k], (rank \* block \* N\_j \* N\_k) + index + k);

//printf("F[%d][%d][%d] = %lf\t;%d\n", i + (rank \* block), N\_j - 1, k, F[0][index\_ + k], (rank \* block \* N\_j \* N\_k) + index\_ + k);

}

}

for (int i = 0; i < block; i++) //заполняем право и лево

{

int index = i \* N\_j \* N\_k; //для правой части (0)

int index\_ = index + N\_k - 1; //для левой части (N\_k)

for (int j = 1; j < N\_j - 1; j++)

{

F[0][index + j \* N\_k] = fi(i + block \* rank, j, 0, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[1][index + j \* N\_k] = fi(i + block \* rank, j, 0, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[0][index\_ + j \* N\_k] = fi(i + block \* rank, j, N\_k - 1, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

F[1][index\_ + j \* N\_k] = fi(i + block \* rank, j, N\_k - 1, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

//printf("F[%d][%d][%d] = %lf\t;%d\n", i + (rank \* block), j, 0, F[0][index + j \* N\_k], (rank \* block \* N\_j \* N\_k) + index + j \* N\_k);

//printf("F[%d][%d][%d] = %lf\t;%d\n", i + (rank \* block), j, N\_k - 1, F[0][index\_ + j \* N\_k], (rank \* block \* N\_j \* N\_k) + index\_ + j \* N\_k);

}

}

}

//Здесь вносятся входные данные и инициализируются массивы (0 - успех, 1 - неудача)

int memory\_init(char\*\* argv, int\* N\_i, int\* N\_j, int\* N\_k, double\* e, int\* a, double\* I\_0, double\* J\_0, double\* K\_0, double\* D\_i, double\* D\_j, double\* D\_k, double\* h\_i, double\* h\_j, double\* h\_k, double\* h\_i\_2, double\* h\_j\_2, double\* h\_k\_2, double\* coefficient, double\*\*\* F, double\*\* up, double\*\* down, int\* swap1, int\* swap2, int\* complete, int\* complete\_all, int\* count, int size, int\* block)

{

\*N\_i = 256;

\*N\_j = 256;

\*N\_k = 256;

if ((argv[1] != 0) && (argv[2] != 0) && (argv[3] != 0))

{

\*N\_i = atoi(argv[1]);

\*N\_j = atoi(argv[2]);

\*N\_k = atoi(argv[3]);

}

if (\*N\_i % size != 0)

{

printf("Error! Use N\_i only multiply of 16!\n");

return 1;

}

\*block = \*N\_i / size;

if (\*block < 3)

{

printf("Error! Use N\_i more than now if you want a lot of threads!\n");

return 1;

}

\*e = 100000000;

if (argv[4] != 0)

{

\*e = atoi(argv[4]);

}

\*e = 1 / \*e;

\*a = 100000;

\*I\_0 = -1.0;

\*J\_0 = -1.0;

\*K\_0 = -1.0;

\*D\_i = 2.0;

\*D\_j = 2.0;

\*D\_k = 2.0;

\*h\_i = \*D\_i / (double)(\*N\_i - 1);

\*h\_j = \*D\_j / (double)(\*N\_j - 1);

\*h\_k = \*D\_k / (double)(\*N\_k - 1);

\*h\_i\_2 = (\*h\_i) \* (\*h\_i);

\*h\_j\_2 = (\*h\_j) \* (\*h\_j);

\*h\_k\_2 = (\*h\_k) \* (\*h\_k);

\*coefficient = 1.0 / (2.0 / (\*h\_i\_2) + 2.0 / (\*h\_j\_2) + 2.0 / (\*h\_k\_2) + (\*a));

\*F = (double\*\*)malloc(2 \* sizeof(double\*));

(\*F)[0] = (double\*)malloc((\*block) \* (\*N\_j) \* (\*N\_k) \* sizeof(double));

(\*F)[1] = (double\*)malloc((\*block) \* (\*N\_j) \* (\*N\_k) \* sizeof(double));

\*up = (double\*)malloc((\*N\_j) \* (\*N\_k) \* sizeof(double));

\*down = (double\*)malloc((\*N\_j) \* (\*N\_k) \* sizeof(double));

\*swap1 = 0;

\*swap2 = 1;

\*complete = 0;

\*complete\_all = 0;

\*count = 0;

return 0;

}

int main(int argc, char\*\* argv)

{

//Необходимые входные данные и их описание

int N\_i, N\_j, N\_k; //размеры узлов

double e; //порог сходимости (указывается обратной величниной, для удобства)

int a; //параметр уравнения

double I\_0, J\_0, K\_0; //начальные координаты

double D\_i, D\_j, D\_k; //размеры моделирования

double h\_i, h\_j, h\_k; //шаги сетки

double h\_i\_2, h\_j\_2, h\_k\_2; //шаги сетки в квадрате

double coefficient; //посчитаем его, чтобы больше не пересчитывать

double\*\* F = NULL; //тут будет результат итеранционного алгортима (текущая и предыдущие и будут по очереди меняться)

int swap1, swap2; //переменные которые отвечают за очередность в F[0] и F[1]

double\* up = NULL, \*down = NULL; //это буфера верхних и нижних значений

int complete, complete\_all; //проверка на завершение работы алгоритма (критерий завершёности)

int count; //просто счётчик итераций (для интереса)

int block; //высота каждого I-ого блока

//Начало алгоритма

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Request send\_down\_to\_up; //отвечает за доставку исходной НИЖНЕЙ части в качестве ВЕРХНЕЙ для предыдущего по счёту rank

MPI\_Request recv\_up\_from\_down; //отвечает за приём ВЕРХНЕЙ части, пришедшей из НИЖНЕЙ последующего по счёту rank

MPI\_Request send\_up\_to\_down; //отвечает за доставку исходной ВЕРХНЕЙ части в качестве НИЖНЕЙ для предыдущего по счёту rank

MPI\_Request recv\_down\_from\_up; //отвечает за приём НИЖНЕЙ части, пришедшей из ВЕРХНЕЙ последующего по счёту rank

if (memory\_init(argv, &N\_i, &N\_j, &N\_k, &e, &a, &I\_0, &J\_0, &K\_0, &D\_i, &D\_j, &D\_k, &h\_i, &h\_j, &h\_k, &h\_i\_2, &h\_j\_2, &h\_k\_2, &coefficient, &F, &up, &down, &swap1, &swap2, &complete, & complete\_all, &count, size, &block) == 0)

{

init\_F(F, block, rank, size, N\_i, N\_j, N\_k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

struct timespec start, end;

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &start);

while (complete\_all == 0)

{

//Обмен границами

if (rank != 0)

{

MPI\_Isend(&(F[swap1][0]), N\_j \* N\_k, MPI\_DOUBLE, rank - 1, 0, MPI\_COMM\_WORLD, &send\_down\_to\_up); //отправляем низ предыдущему блоку (для вверха)

MPI\_Irecv(down, N\_j \* N\_k, MPI\_DOUBLE, rank - 1, 1, MPI\_COMM\_WORLD, &recv\_down\_from\_up); //примнимаем снизу вверх предыдущего блока

}

if (rank != size - 1)

{

MPI\_Isend(&(F[swap1][(block - 1) \* N\_j \* N\_k]), N\_j \* N\_k, MPI\_DOUBLE, rank + 1, 1, MPI\_COMM\_WORLD, &send\_up\_to\_down); //отправляем вверх следующему блоку (для низа)

MPI\_Irecv(up, N\_j \* N\_k, MPI\_DOUBLE, rank + 1, 0, MPI\_COMM\_WORLD, &recv\_up\_from\_down); //принимаем вверх снизу следующего блока

}

//Считаем внутри

iteration\_in(F, block, rank, swap1, swap2, &complete, e, N\_i, N\_j, N\_k, coefficient, a, h\_i, h\_j, h\_k, h\_i\_2, h\_j\_2, h\_k\_2, I\_0, J\_0, K\_0);

//Ожидаем, пока они не перекинуться данными

if (rank != 0)

{

MPI\_Wait(&recv\_down\_from\_up, MPI\_STATUS\_IGNORE);

MPI\_Wait(&send\_down\_to\_up, MPI\_STATUS\_IGNORE);

}

if (rank != size - 1)

{

MPI\_Wait(&recv\_up\_from\_down, MPI\_STATUS\_IGNORE);

MPI\_Wait(&send\_up\_to\_down, MPI\_STATUS\_IGNORE);

}

//Считаем края

iteration\_edge(F, up, down, block, rank, size, swap1, swap2, &complete, e, N\_i, N\_j, N\_k, coefficient, a, h\_i, h\_j, h\_k, h\_i\_2, h\_j\_2, h\_k\_2, I\_0, J\_0, K\_0);

//Собираем в кучу результаты

MPI\_Allreduce(&complete, &complete\_all, 1, MPI\_INT, MPI\_MAX, MPI\_COMM\_WORLD);

count++;

swapping(&swap1, &swap2);

}

clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);

double accuracy\_ = accuracy(F, swap1, block, rank, a, N\_i, N\_j, N\_k, h\_i, h\_j, h\_k, I\_0, J\_0, K\_0);

double accuracy\_max;

double time = end.tv\_sec - start.tv\_sec + 0.000000001 \* (end.tv\_nsec - start.tv\_nsec);

double time\_min;

MPI\_Reduce(&time, &time\_min, 1, MPI\_DOUBLE, MPI\_MIN, 0, MPI\_COMM\_WORLD);

MPI\_Reduce(&accuracy\_, &accuracy\_max, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

if (rank == 0)

{

printf("count = %d\n", count);

printf("Accuracy = %e\n", accuracy\_max);

printf("Time: %lf \n", time);

}

//Завершаемся

free(F[0]);

free(F[1]);

free(F);

free(up);

free(down);

}

MPI\_Finalize();

return 0;

}