**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 3

«Умножение матрицы на матрицу в MPI 2D решетка»

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 12](#_Toc70030782)

[Приложение *Листинг файла lab3.c* 13](#_Toc70030783)

# ЦЕЛЬ

* Написать параллельный алгоритм, осуществляющийся на двумерной решетке, для решение матричного умножения
* Проанализировать полученную программу и выявить зависимость производительности программы от входных данных, варианта решения, количество ядер

# ЗАДАНИЕ

1. Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке
2. Исследовать производительность параллельной программы в зависимости от размера матрицы и размера решетки.
3. Выполнить профилирование программы с помощью MPE при использовании 16-и ядер

# **ОПИСАНИЕ РАБОТЫ**

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

Прежде чем писать полную программу, которая бы делила матрицу на строки и столбцы, я сначала реализовал более оптимальный способ умножения матрицы, которое мне известно ещё с курса по ЭВМ. Суть в том, что мы двигаемся построчно, а не по столбикам, тем самым, не сильно нагружая память.

Далее я протестировал алгоритм умножения на небольших матрицах с помощью калькулятора матриц онлайн. Везде получался верный ответ, значит алгоритм работает верно и его можно применять. Я проверял на маленьких матрицах, например такой: . Такая матрица отлично подходит для разделения до 4-ёх полосок, так как число – в нем содержатся все сомножители. Кстати, потом я сравнивал результаты программы, которая считает без разделения на процессы и с разделением, чтобы понять, насколько верно собралась и посчиталась матрица.

В этой программе процессов. Доступно нам всего 16, значит произведение не должно превосходить это количество. Также я не стал учитывать разделение матрицы на неравные кусочки. В моей программе строго все кусочки равны, для этих целей у меня даже в начале есть проверка на то, чтобы матрица делилась ровно, иначе программа откажется работать.

Для разделения программы на процессы я использовал свой коммуникатор, на основе которого я создавал топологию. Топологию я делил как решетку. Затем я выделял память для матриц и подматриц. По заданию, исходные матрицы и хранятся в 0-ом процессе. А остальные кусочки (подматрицы) хранятся в отдельных процессах. Потом осуществлялось деление процессов на два коммуникатора. Осуществлял я с помощью функции . Затем воспользовался для разделения матрицы на столбики и применил для новых 0-ых процессов. А после, сделал широковещательную рассылку с помощью . Таким образом в каждом процессе получалась нужная нам подматрица.

Далее я применил алгоритм перемножения матриц, который был описан выше. А затем было необходимо собрать всё в кучу обратно. Осуществил я это с помощью двух функций: и . В качестве вектора у меня выступала подматрица , а собирал эти матрицы с помощью второй функции. Пришлось придумать хитроумные смещения, чтобы матрица собиралась корректно.

Теперь самое главное: нужно проверить правильность выполнения сборки матриц. Сравнивать каждую циферку в матрице было бы не очень удобно. Поэтому я придумал как можно было бы быстро проверить сборку. Я суммировал элементы подматриц и суммировал итоговую сумму. А в 0-ом процессе я суммировал готовую матрицу. Если эти две суммы равны, то, вероятно, сборка матриц осуществилась верно. Если бы суммы отличались, это было бы верным признаком неровной сборки матрицы с пустыми местами. Однако проверить полностью, верно ли собрана матрица, не представляется возможным, кроме того, как вывести её полностью или параллельно посчитать произведение матриц без разделения на процессы. А потом просто поэлементно сравнить. Но я всё-таки положился на предположение о том, если на маленьких данных на разных процессах всё работало верно и корректно, то и на больших данных всё должно быть аналогично.

В конце программы происходит очистка памяти и вычисление количества времени работы программы. Время на каждом процессе разное, но я беру минимальное, так как погрешность вычисления на компьютере берётся всегда наименьшее.

Таблица с компиляцией программы:

|  |  |  |
| --- | --- | --- |
| Файл | Описание | Команда компиляции |
| [lab3.c](#_Приложение_1) | Код программы | mpiicc lab3.c -std=gnu99 -o lab3.exe |

Пояснение к команде компиляции:

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

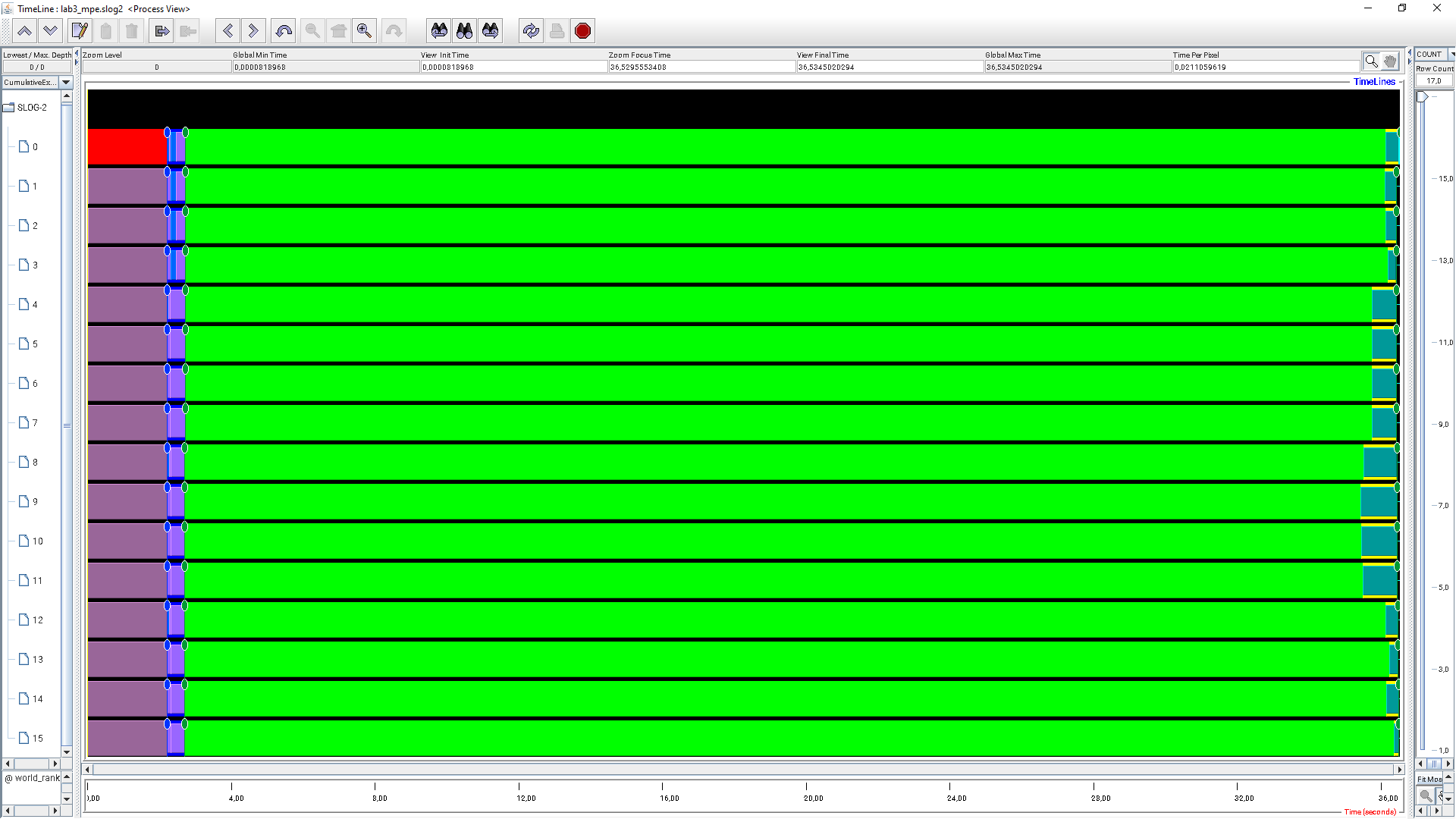
Таблица с результатами при :

|  |  |  |
| --- | --- | --- |
|  | Время: | Эффективность |
| без MPI | 228,3020 | 1,043261 |
| 1 | 238,1786 | 1 |
| 2 | 128,5062 | 1,853441 |
| 3 | 107,1179 | 2,223518 |
| 4 | 82,33971 | 2,892633 |
| 5 | 78,87267 | 3,019786 |
| 6 | 69,34511 | 3,434685 |
| 7 | 75,59625 | 3,150667 |
| 8 | 66,61576 | 3,575409 |
| 9 | 61,25383 | 3,888387 |
| 10 | 53,92911 | 4,416513 |
| 11 | - | - |
| 12 | 44,89134 | 5,305669 |
| 13 | - | - |
| 14 | 38,84266 | 6,131882 |
| 15 | 35,36795 | 6,734306 |
| 16 | 33,97026 | 7,011386 |

Такие входные данные были выбраны неслучайно. Число 5040 отлично делится на все числа от 0 до 16 за исключением 11 и 13 и в то же время является минимальным таким числом. Остальные размеры матрицы тоже были сделаны на основе этого числа, чтобы также всё хорошо равномерно делилось. Элементы матрицы заполняются рандомно командой со значениями от -9 до 9. Ключ генерации не выставлял, но по умолчанию она равен: .

Профилирование корректно произвести не удалось, потому что в конце программы выводится ошибка «Segmentation fault». Хотя в обычной программе такого не было, и программа завершалась успешно. Видимо это особенности MPE.

Вот полный скриншот:



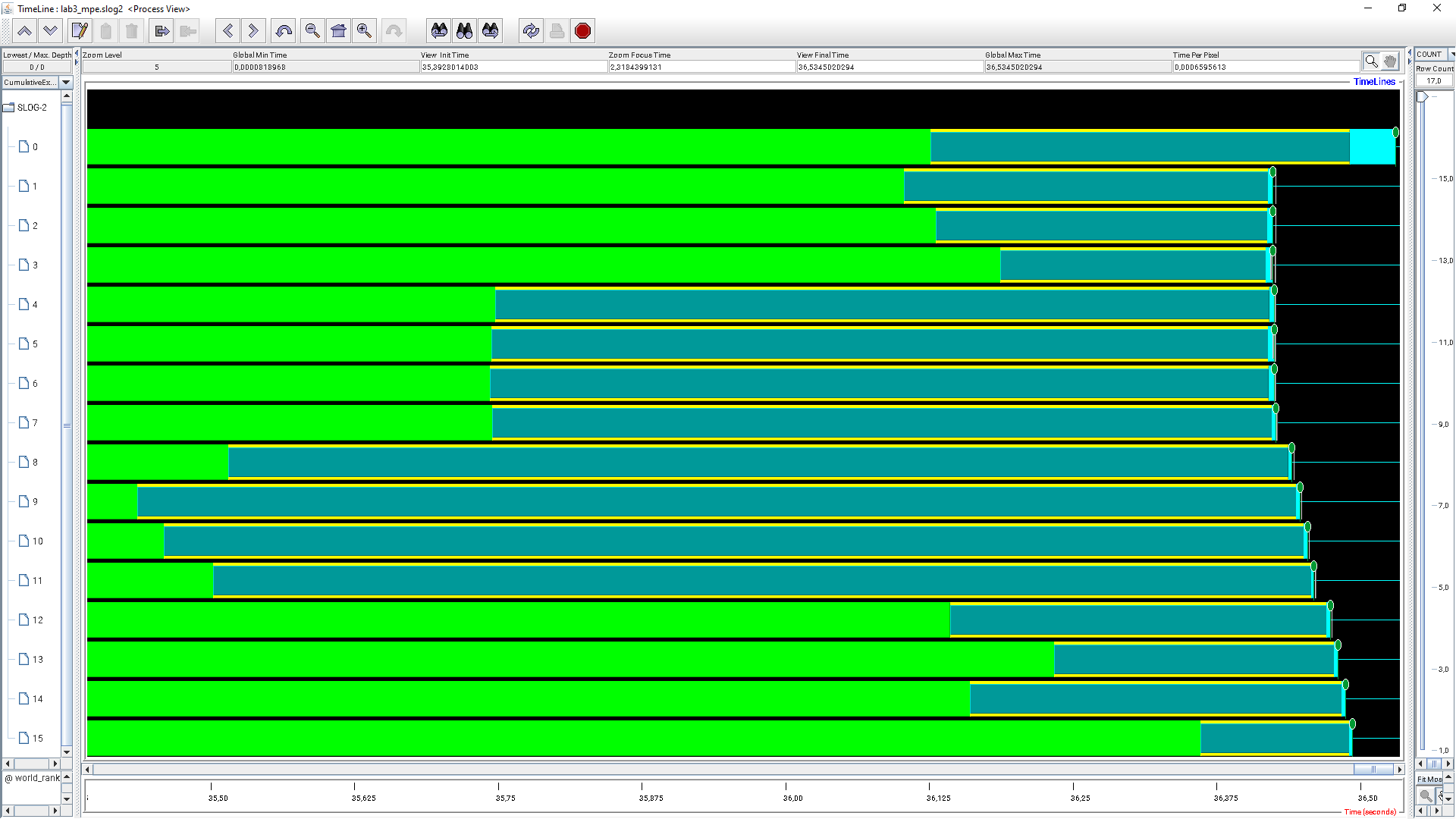
Зелёная зона – это как раз перемножение кусочков матриц.

Приближение зоны дробления матрицы на части:



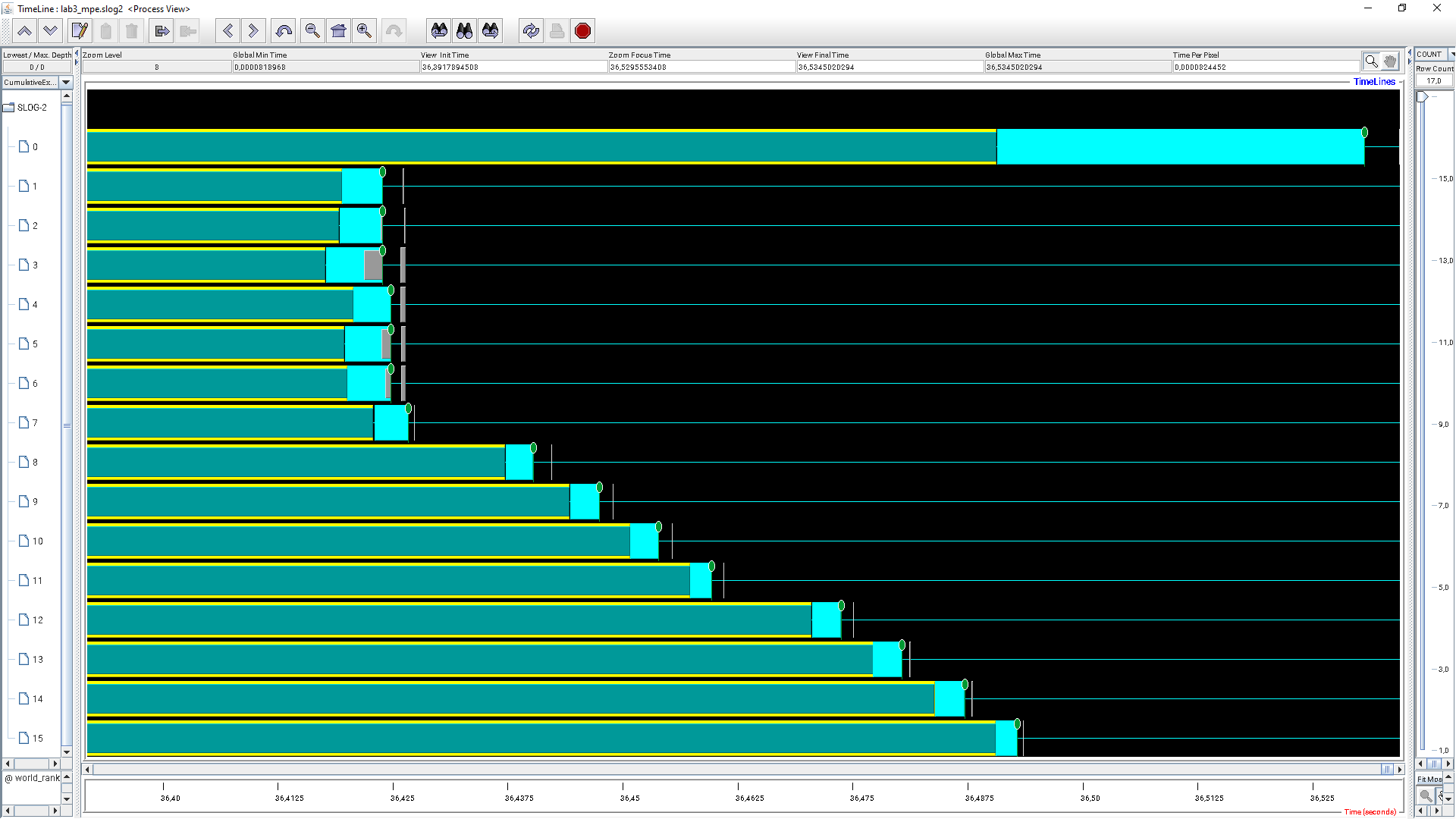
Синим цветом вокруг фиолетового фона отмечена зона дробления матрицы на части. Маленькие синие зоны, вероятно, функции . Остальное фиолетовое – ожидание и .

Приближение зоны сборки матрицы:

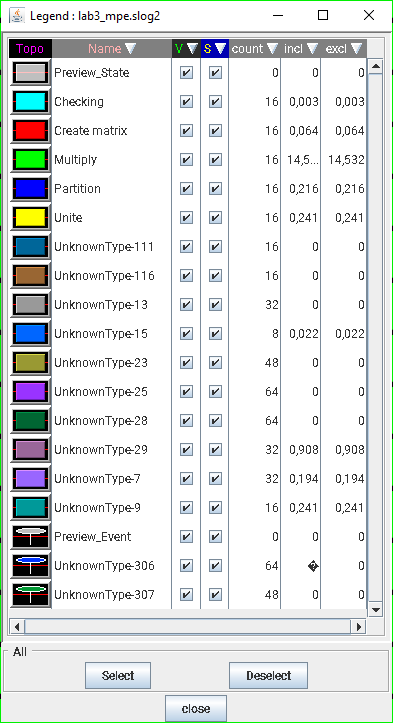


Желтым отмечено ожидание и выполнение сборки матрицы в единое целое. Так как функция блокирующая, то выполнение её начинается с последней зелёной зоны и до первой бирюзовой.

Приближение зоны проверки сборки матрицы:



Бирюзовым цветом отмечен контрольный подсчёт суммы элементов матрицы.

Из-за ошибки в конце программы, профилировщик не может определить названия MPI функций, поэтому мне пришлось самому уже добавлять цвета и модернизировать программу так, чтоб она показала все нужные, работающие зоны.

Отмечу, что красным цветом («Create matrix») показано выделение памяти для матриц и заполнение матрицы рандомными значениями, а также обнулением подматриц, отвечающие за произведение.

Синим цветом («Partition») показано разделение основной матрицы на подматрицы и рассылки нужных элементов. Синяя зона отмечена как окантовка, вокруг фиолетовой зоны. Внутри синей зоны, имеются ещё голубые участки. Это вызов функции . Причём вызывается сначала на 0, 4, 8, 12 – что соответствует разделению кусочков матрицы . А затем идёт , вероятно, до следующей голубой зоны в 0 – 3 процессах. Эти процессы уже делят матрицу и длятся они подольше, так как тут используются изменённый тип данных - . А после, возможно, до зелёной зоны работает рассылка этих кусочков.

Зелёная зона («Multiply») – это обыкновенное перемножение, которое, кстати, не очень равномерно завершается. Разница может составить до одной секунды с учётом того, что программа в целом работает чуть больше 30 секунд. Зона перемножения занимает более 90% времени, что с учетом рандомной матрицы и проверки на правильность сборки матрицы очень неплохой результат.

Желтая окантовка («Unite») – зона, где должно совершиться сборка матриц в единое целое. Однако она начинается сразу же по окончанию перемножения. Это нормально для функции , ведь она является блокирующей и начинает работать после последнего завершения зелёной зоны.

Бирюзовая зона («Checking») – проверка корректности сборки матрицы. Это обычная функция, которая суммирует каждый элемент результативной подматрицы и итоговую матрицу. А потом результат выводит в терминал, для понимания, насколько точно собралась матрица обратно.

# **ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы я изучил несколько новых функций MPI, а именно создание топологии и решётки, а также работу с коммуникаторами. Ознакомился с функциями, создающие новых тип данных. Также я проанализировал полученные графики. И по ним можно сделать вывод о том, что умножение матрицы хоть и разделяется на несколько процессов, однако не наблюдается рост производительности в количество процессов, обрабатывающих матрицу. Т.е. если я матрицу разобью на 16 процессов, то программа посчитает её только в 7-8 раз быстрее. Не стоит забывать, что в подсчёт времени включено ещё и создание топологии, деление матрицы на кусочки, выделение памяти, сборка матрицы и проверка на корректность сборки.

# Приложение 1

*Листинг файла lab3.c*

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#include <time.h>

#include "mpi.h"

void random\_matrix(int\* M, int size)

{

for (int i = 0; i < size; i++)

{

M[i] = rand() % 18 - 9;

}

}

void print\_matrix(int\* M, int h, int l, int rank)

{

if (rank == 0)

{

for (int i = 0; i < h; i++)

{

for (int j = 0; j < l; j++)

{

printf("%4d ", M[i \* l + j]);

}

printf("\n");

}

}

}

//Просто проверка, правильно ли посчитала матрица

void check\_result(MPI\_Comm comm2d, int\* C, int\* C\_, int rank, int n1, int n3, int p1, int p2)

{

long long int check = 0;

if (rank == 0)

{

for (int i = 0; i < n1 \* n3; i++)

{

check += C[i];

}

printf("Check: %lld\n", check);

}

long long int check1 = 0;

long long int check2 = 0;

for (int i = 0; i < (n1 / p1) \* (n3 / p2); i++)

{

check1 += C\_[i];

}

MPI\_Reduce(&check1, &check2, 1, MPI\_LONG\_LONG\_INT, MPI\_SUM, 0, comm2d);

if (rank == 0)

{

printf("sum = %lld\n", check2);

}

}

//Сбор всех подматриц в матрицу

void unite\_matrix(MPI\_Comm comm2d, int\* C, int\* C\_, int rank, int size, int n1, int n2, int n3, int p1, int p2)

{

MPI\_Datatype C\_vector;

MPI\_Type\_vector(n1 / p1, n3 / p2, n3, MPI\_INT, &C\_vector);

MPI\_Type\_commit(&C\_vector);

MPI\_Datatype C\_vector\_;

MPI\_Type\_create\_resized(C\_vector, 0, (n3 / p2) \* sizeof(int), &C\_vector\_);

MPI\_Type\_commit(&C\_vector\_);

int\* recvcounts = (int\*)malloc(size \* sizeof(int));

int\* displs = (int\*)malloc(size \* sizeof(int));

for (int i = 0; i < p1; i++)

{

for (int j = 0; j < p2; j++)

{

displs[i \* p2 + j] = j + i \* p2 \* (n1 / p1);

recvcounts[i \* p2 + j] = 1;

}

}

MPI\_Gatherv(C\_, (n1 / p1) \* (n3 / p2), MPI\_INT, C, recvcounts , displs , C\_vector\_, 0, comm2d);

MPI\_Type\_free(&C\_vector);

MPI\_Type\_free(&C\_vector\_);

free(recvcounts);

free(displs);

}

//Умножения матриц

void multiply(int\* C, int\* A, int\* B, int n1, int n2, int n3)

{

for (int i = 0; i < n1; i++)

{

for (int j = 0; j < n2; j++)

{

for (int k = 0; k < n3; k++)

{

C[i \* n3 + k] += A[i \* n2 + j] \* B[j \* n3 + k];

}

}

}

}

//Отправка кусков матриц в процессы

void send\_chunks(MPI\_Comm comm2d, int size, int rank, int\* A, int\* B, int\* C, int\* A\_, int\* B\_, int\* C\_, int n1, int n2, int n3, int p1, int p2)

{

int color\_A = rank / p2;

int color\_B = rank % p2;

int sizeA;

int sizeB;

int rankA;

int rankB;

MPI\_Comm commA;

MPI\_Comm commB;

MPI\_Comm\_split(comm2d, color\_A, color\_B, &commB);

MPI\_Comm\_split(comm2d, color\_B, color\_A, &commA);

MPI\_Comm\_rank(commA, &rankA);

MPI\_Comm\_rank(commB, &rankB);

MPI\_Comm\_size(commA, &sizeA);

MPI\_Comm\_size(commB, &sizeB);

MPI\_Datatype B\_vector;

MPI\_Type\_vector(n2, n3 / p2, n3, MPI\_INT, &B\_vector);

MPI\_Type\_commit(&B\_vector);

MPI\_Datatype B\_vector\_;

MPI\_Type\_create\_resized(B\_vector, 0, (n3 / p2) \* sizeof(int), &B\_vector\_);

MPI\_Type\_commit(&B\_vector\_);

if (rank == rankA \* sizeB)

{

MPI\_Scatter(A, (n1 / p1) \* n2, MPI\_INT, A\_, (n1 / p1) \* n2, MPI\_INT, 0, commA);

}

MPI\_Bcast(A\_, (n1 / p1) \* n2, MPI\_INT, 0, commB);

if (rank == rankB)

{

MPI\_Scatter(B, 1, B\_vector\_, B\_, n2 \* (n3 / p2), MPI\_INT, 0, commB);

}

MPI\_Bcast(B\_, n2 \* (n3 / p2), MPI\_INT, 0, commA);

MPI\_Comm\_free(&commA);

MPI\_Comm\_free(&commB);

MPI\_Type\_free(&B\_vector);

MPI\_Type\_free(&B\_vector\_);

}

//Выделение памяти для процессов

void memory\_matrix(int\*\* A, int\*\* B, int\*\* C, int\*\* A\_, int\*\* B\_, int\*\* C\_, int rank, int n1, int n2, int n3, int p1, int p2)

{

\*A = NULL;

\*B = NULL;

\*C = NULL;

\*A\_ = (int\*)malloc((n1 / p1) \* n2 \* sizeof(int));

\*B\_ = (int\*)malloc(n2 \* (n3 / p2) \* sizeof(int));

\*C\_ = (int\*)malloc((n1 / p1) \* (n3 / p2) \* sizeof(int));

memset(\*C\_, 0, (n1 / p1) \* (n3 / p2) \* sizeof(int));

if (rank == 0)

{

\*A = (int\*)malloc(n1 \* n2 \* sizeof(int));

\*B = (int\*)malloc(n2 \* n3 \* sizeof(int));

\*C = (int\*)malloc(n1 \* n3 \* sizeof(int));

random\_matrix(\*A, n1 \* n2);

random\_matrix(\*B, n2 \* n3);

}

}

//Создание топологии

void topology(MPI\_Comm\* comm2d, int\* rank, int\* size, int p1, int p2)

{

int dims[2] = { p1,p2 }; //решётка p1 на p2

int periods[2] = { 0,0 }; //переодичность (непереодическая)

int reorder = 1; //указываает, производить ли перенумерацию процессов

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

MPI\_Dims\_create(\*size, 2, dims); //определение размера решётки

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dims, periods, reorder, comm2d); //создание решётки

MPI\_Comm\_rank(\*comm2d, rank); //забираем, вероятно, новые номера процессов

}

//Здесь начинаются все алгоритмы

void go(int n1, int n2, int n3, int p1, int p2)

{

MPI\_Comm comm2d; //коммуникатор решётки

int rank, size;

topology(&comm2d, &rank, &size, p1, p2); //создаём топологию

int\* A, \*B, \*C, \*A\_, \*B\_, \*C\_;

memory\_matrix(&A, &B, &C, &A\_, &B\_, &C\_, rank, n1, n2, n3, p1, p2); //выделяем память и создаём матрицы

send\_chunks(comm2d, size, rank, A, B, C, A\_, B\_, C\_, n1, n2, n3, p1, p2); //рассылка кусочков всем процессам

multiply(C\_, A\_, B\_, n1 / p1, n2, n3 / p2); //перемножение матриц

unite\_matrix(comm2d, C, C\_, rank, size, n1, n2, n3, p1, p2); //сбор матриц в единое целлое

//printf("\nC: \n");

//print\_matrix(C, n1, n3, rank);

check\_result(comm2d, C, C\_, rank, n1, n3, p1, p2); //проверка результатов

MPI\_Comm\_free(&comm2d);

free(A);

free(B);

free(C);

free(A\_);

free(B\_);

free(C\_);

}

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

{

int n1 = 5040;

int n2 = 7560;

int n3 = 10080;

int p1 = 4;

int p2 = 4;

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

{

p1 = atoi(argv[1]);

p2 = atoi(argv[2]);

}

if ((argv[3] != 0) && (argv[4] != 0) && (argv[5] != 0))

{

n1 = atoi(argv[3]);

n2 = atoi(argv[4]);

n3 = atoi(argv[5]);

}

if (p1 \* p2 > 16)

{

printf("Big number of p1 or p2 (p1 = %d; p2 = %d)\n", p1, p2);

return 0;

}

if ((p1 < 1) || (p2 < 1))

{

printf("Small number of p1 or p2 (p1 = %d; p2 = %d)\n", p1, p2);

return 0;

}

if ((n1 % p1 != 0) || (n3 % p2 != 0))

{

printf("Error in p1 or p2!\n");

return 0;

}

MPI\_Init(&argc, &argv);

int size;

int rank;

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

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

printf("Threads: %d\tNumber: %d\n", size, rank);

if (size != p1 \* p2)

{

if (rank == 0)

{

printf("Error! Threads is %d, p1\*p2=%d\n", size, p1 \* p2);

}

}

else

{

float time;

float time\_min;

struct timespec start, finish;

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

go(n1, n2, n3, p1, p2); //все возможные ошибки проверили - можно начинать!

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

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

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

if (rank == 0)

{

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

}

}

MPI\_Finalize();

return 0;

}