###### 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 ПРАКТИЧЕСКОЙ РАБОТЫ

Умножение матрицы на матрицу в MPI 2D решетке

студента 2 курса, группы 22204

Соломенникова Николая Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

доцент

А.Ю.Власенко

Новосибирск 2023

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#__RefHeading___1)

[ЗАДАНИЕ 3](#__RefHeading___2)

[ОПИСАНИЕ РАБОТЫ 4](#__RefHeading___3)

[ЗАКЛЮЧЕНИЕ 7](#__RefHeading___4)

[ИСПОЛЬЗУЕМЫЕ КОМАНДЫ 8](#__RefHeading___5)

[ЛИСТИНГИ 9](#__RefHeading___6)

[prog1.cpp (последовательная программа) 9](#__RefHeading___7)

[prog2.cpp (параллельная программа) 12](#__RefHeading___8)

# ЦЕЛЬ

Освоении концепции MPI-коммуникаторов и декартовых топологий, а также концепции производных типов данных.

# ЗАДАНИЕ

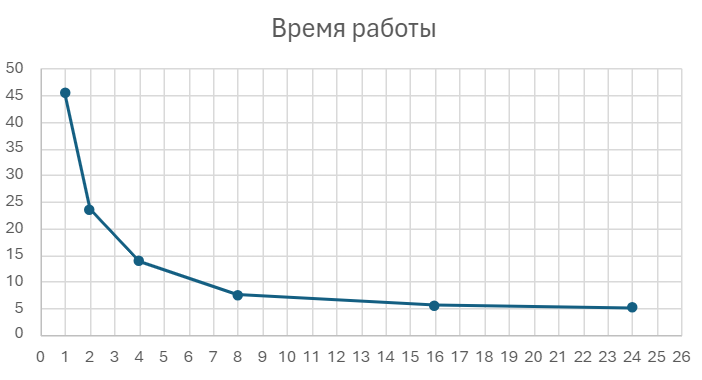
1. Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке процессов с соблюдением требований.
2. Исследовать производительность параллельной программы при фиксированном размере матрицы в зависимости от и размера решетки: 2x12, 3x8, 4x6, 6x4, 8x3, 12x2. Размер матриц подобрать таким образом, чтобы худшее из времен данного набора было не менее 30 сек.
3. Выполнить профилирование программы при использовании 8-и ядер с решетками 2x4, 4x2.

# ОПИСАНИЕ РАБОТЫ

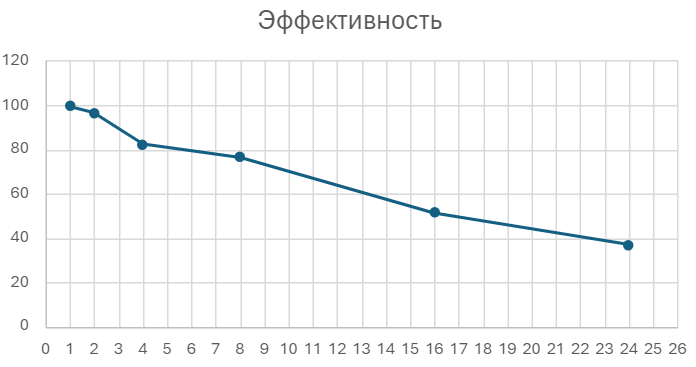
Время работы последовательной программы: T0 = 45,3

Времена работы параллельной программы при разном количестве процессов (размеры матриц при замерах: N1 = N2 = N3 = 1500):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| кол-во процессов | 1 | 2 | 4 | 8 | 16 | 24 |
| сетка | 1x1 | 2x1 | 2x2 | 4x2 | 4x4 | 6x4 |
| время, сек | 45,6 | 23,5 | 13,8 | 7,4 | 5,5 | 5,1 |

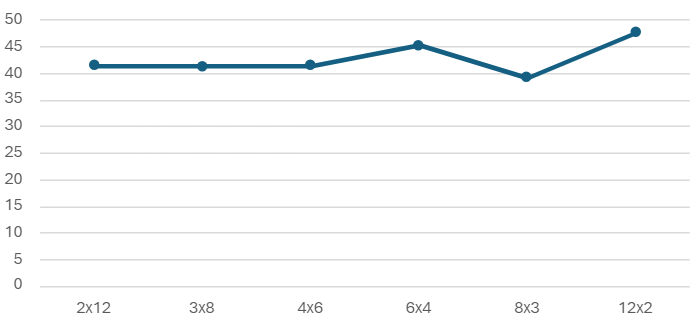




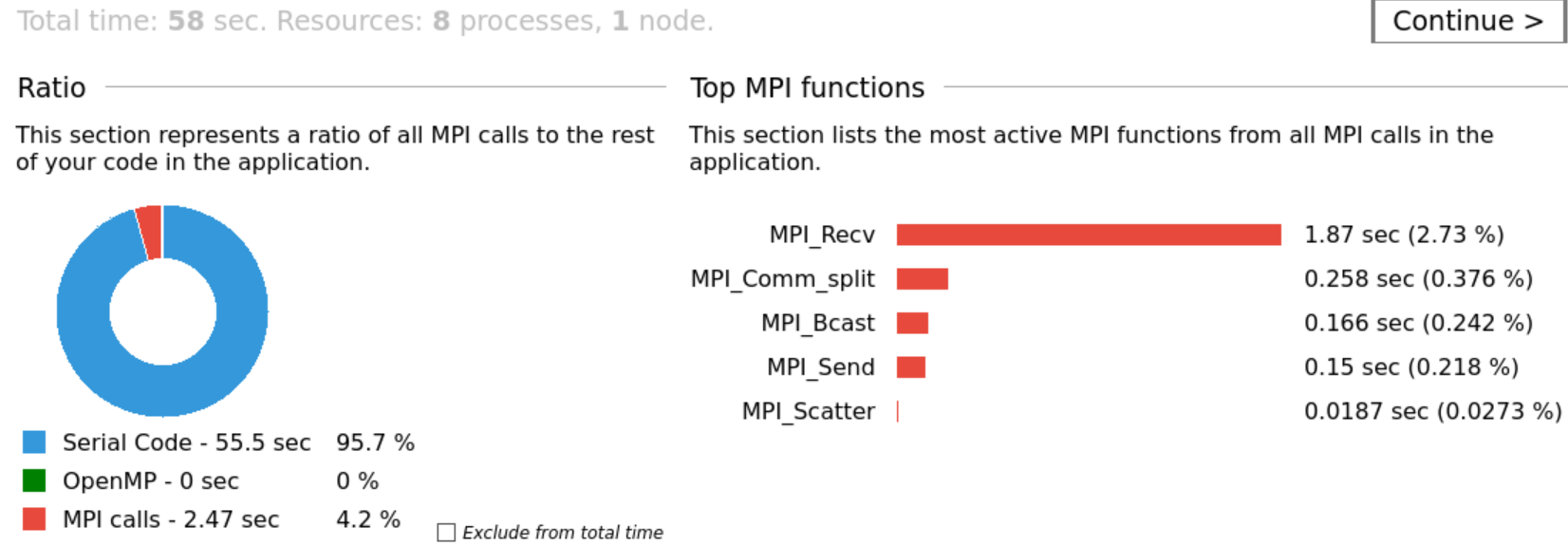


Зависимость времени работы параллельной программы от размеров сетки процессов (размеры матриц при замерах: N1 = N2 = N3 = 3000):

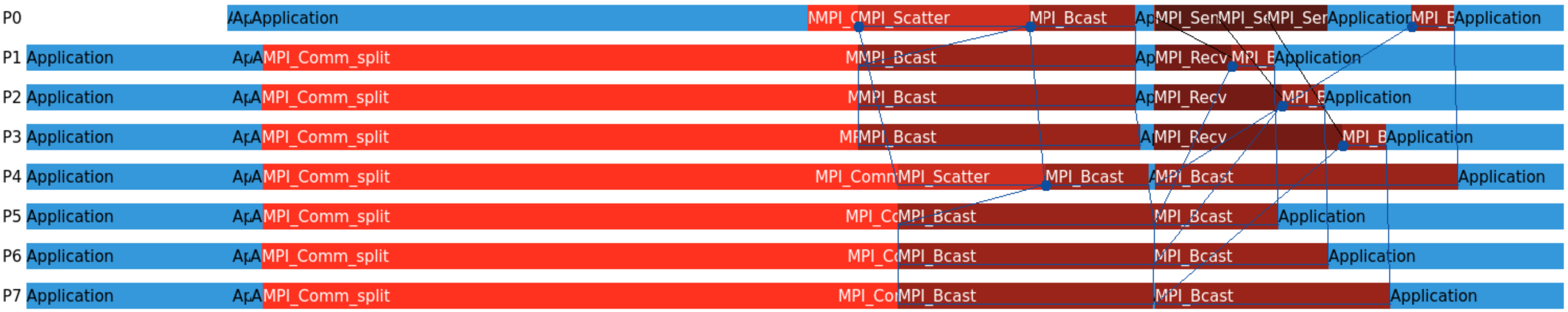
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| сетка | 2x12 | 3x8 | 4x6 | 6x4 | 8x3 | 12x2 |
| время, сек | 41,5 | 41,3 | 41,5 | 45,3 | 39,2 | 47,6 |



Профилирование (N1 = N2 = N3 = 1500, P1 = 2, P2 = 4):

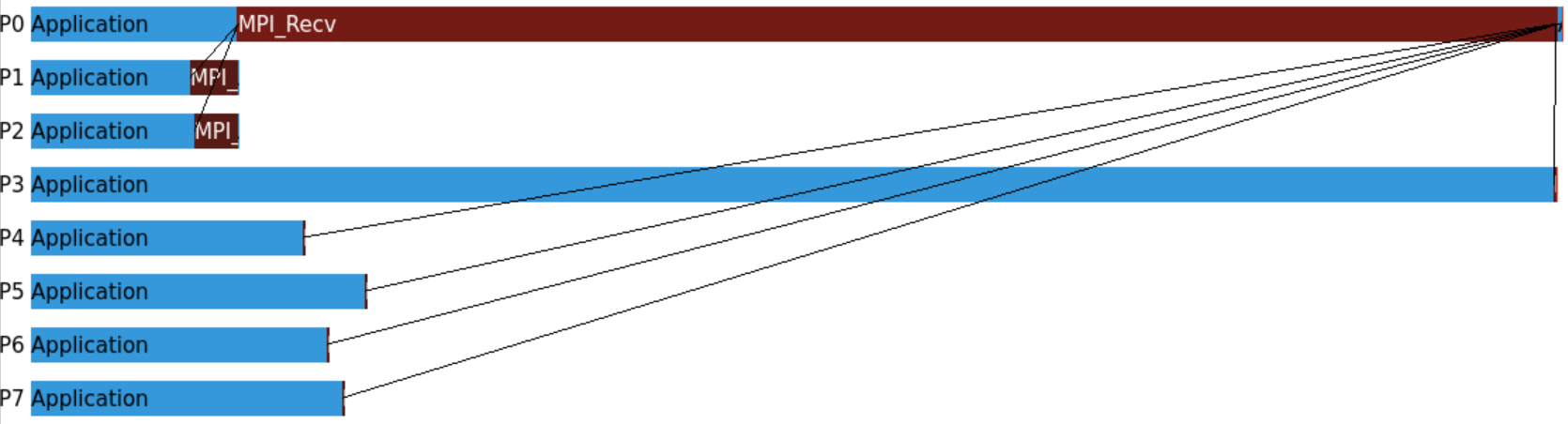


Начало программы (раздача матриц):

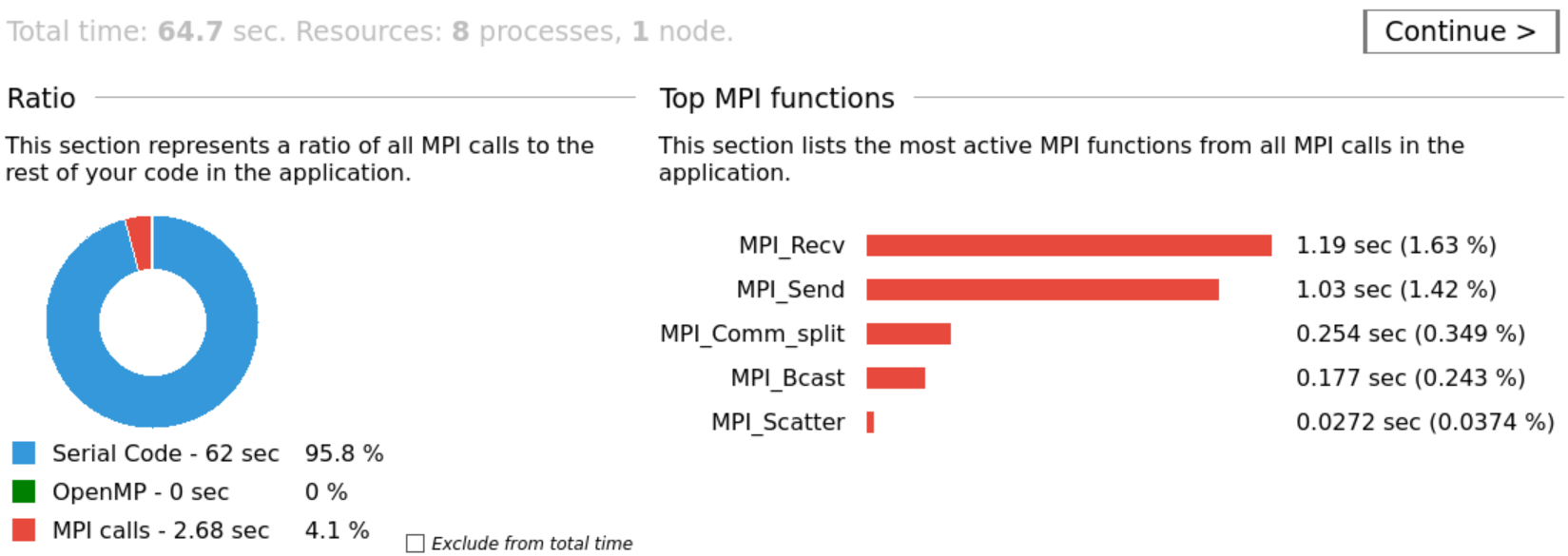


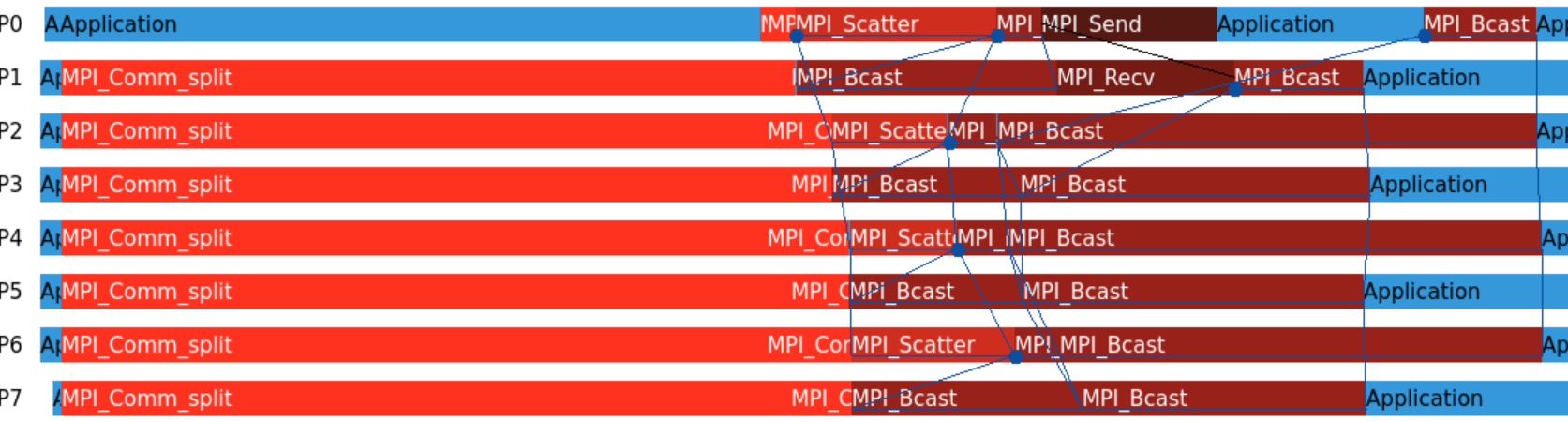
В середине программы – только вычисления (перемножение матриц).

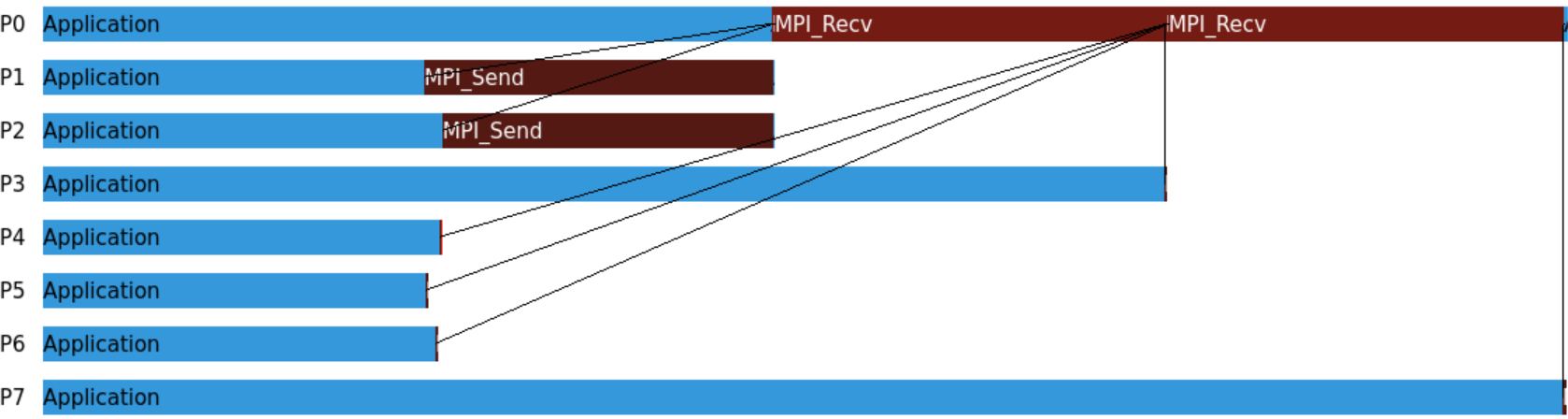
Конец программы (сбор результатов вычислений):



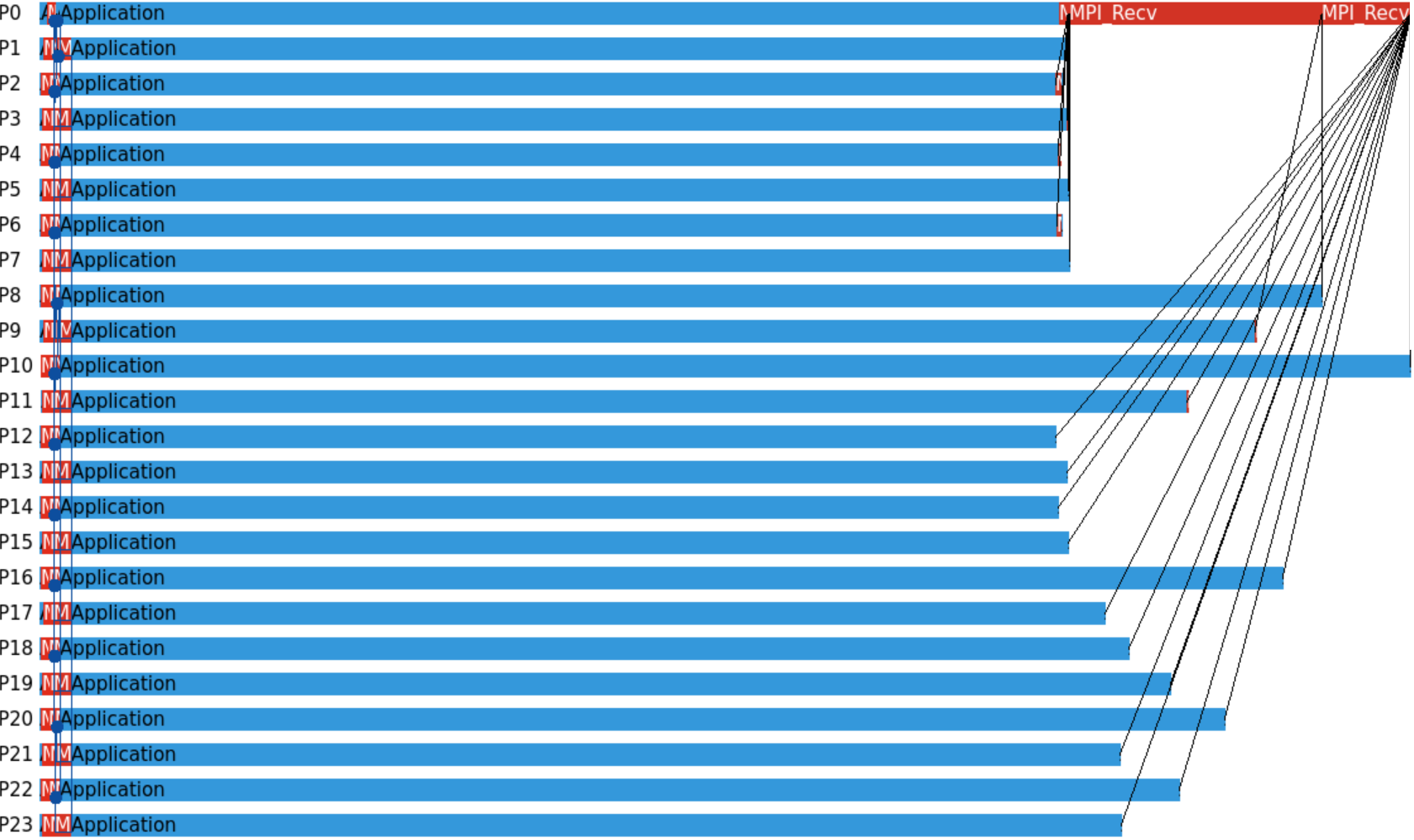
То же самое профилирование, но сетка процессов не 2x4, а 4x2:







Трасса для программы, запущенной на 24 процессах с сеткой 12x2:



# ЗАКЛЮЧЕНИЕ

Про профилирование:

* В самом начале программы нулевой процесс выполняет заполнение матриц A, B, C, поэтому, чтобы далее выполнился split, остальным процессам надо подождать.
* Далее идёт scatter, который отрабатывает только для нулевой колонки процессов (в сетке 2x4 это процессы p0, p4, а в сетке 4x2 это p0, p2, p4 и p6). То есть поведение scatter зависит от размеров сетки.
* Далее – bcast, чтобы раздать соответствующие части матрицы A всем процессам. Bcast не может завершиться, пока не выполнится scatter, поэтому некоторые процессы (на которых scatter не отрабатывает) просто ждут.
* Далее нулевой процесс через коммуникации точка-точка раздаёт вертикальные полосы матрицы B (в этих коммуникациях для сетки 2x4 учавствуют процессы p0, p1, p2 и p3, а для сетки 4x2 – процессы p0, p1).
* Далее делается bcast, чтобы нужные части матрицы B получили все процессы. Можно заметить, что между send и bcast на нулевом процессе есть пользовательский код. Это – копирование части матрицы B в LocB.
* Так как процессы из нулевой строки раздают bcast-ом части матрицы B в своих столбцах остальным процессам, а send-ы на нулевом процессе выполняются последовательно, наблюдается ступенчатая форма графика.
* В конце программы с помощью коммуникаций точка-точка на на нулевом процессе собирается матрица C. Некоторые процессы считают произведение матриц дольше чем остальные, поэтому их проходится ждать, чтобы нулевой процесс сделал recv.

Не совсем понятно почему, но сетки 6x4 и 12x2 отработали хуже всего, а все остальные – примерно с одинаковым временем.

# ИСПОЛЬЗУЕМЫЕ КОМАНДЫ

Компиляция

$ mpicxx src/prog1.cpp -o src/prog1.out

$ mpicxx src/prog2.cpp -o src/prog2.out

Запуск последовательной программы (N1 = N2 = N3 = 1500)

$ mpiexec -n 1 src/prog1.out 1500 1500 1500

Запуск параллельной программы (N1 = N2 = N3 = 3000, P1 = 4, P2 = 6)

$ mpiexec -n 24 src/prog2.out 3000 3000 3000 4 6

Сборка трассы

$ mpiexec -n 8 -trace src/prog2.out 1500 1500 1500 2 4

Просмотр трассы

$ traceanalyzer prog2.out.stf

# ЛИСТИНГИ

### prog1.cpp (последовательная программа)

#include <iostream>

#include <mpi.h>

#include <string>

// N1 \* N2 - размер матрицы A

// N2 \* N3 - размер матрицы B

// N1 \* N3 - размер матрицы C

int N1 = 4;

int N2 = 4;

int N3 = 4;

// Класс исключений для отслеживания ошибок

class myException : public std::exception

{

private:

    const char \*msg\_;

public:

    explicit myException(const char \*msg) : msg\_(msg){};

    const char \*what() const noexcept override { return msg\_; }

};

// Инициализация размеров матриц (по умолчанию есть заданные размеры, но их можно явно указать через аргументы командной строки)

inline void initSizes(const int argc, char \*\*argv)

{

    // Параметры программы: N1 N2 N3 P1 P2

    if (argc >= 4)

    {

        N1 = std::stoi(argv[1]);

        N2 = std::stoi(argv[2]);

        N3 = std::stoi(argv[3]);

    }

}

// Освобождение памяти

inline void deleteData(double \*&A, double \*&B, double \*&C)

{

    free(A);

    free(B);

    free(C);

}

// Инициализация матриц для корневого процесса (именно здесь задаются значения в матрицах)

inline void initData(double \*&A, double \*&B, double \*&C)

{

    A = (double \*)malloc(sizeof(double) \* N1 \* N2);

    B = (double \*)malloc(sizeof(double) \* N2 \* N3);

    C = (double \*)malloc(sizeof(double) \* N1 \* N3);

    if (A == nullptr || B == nullptr || C == nullptr)

    {

        deleteData(A, B, C);

        throw myException("Malloc fail");

    }

    for (int y = 0; y < N1; ++y)

    {

        for (int x = 0; x < N2; ++x)

        {

            A[y \* N2 + x] = y \* N2 + x;

        }

    }

    for (int y = 0; y < N2; ++y)

    {

        for (int x = 0; x < N3; ++x)

        {

            B[y \* N3 + x] = y \* N3 + x;

        }

    }

}

// Вычисления

inline void calculate(double \*const A, double \*const B, double \*const C)

{

    for (int i = 0; i < N1; ++i)

    {

        for (int j = 0; j < N3; ++j)

        {

            C[i \* N3 + j] = 0;

            for (int k = 0; k < N2; ++k)

            {

                C[i \* N3 + j] += A[i \* N2 + k] \* B[k \* N3 + j];

            }

        }

    }

}

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

{

    MPI\_Init(&argc, &argv);

    const double startTime = MPI\_Wtime();

    int rank, size;

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

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

    if (size != 1)

    {

        if (rank == 0)

            std::cerr << "Sequential program must be run by a single process\n";

        return 0;

    }

    double \*A, \*B, \*C;

    double \*LocA, \*LocB, \*LocC;

    try

    {

        initSizes(argc, argv);

        initData(A, B, C);

        calculate(A, B, C);

    }

    catch (const std::exception &e)

    {

        std::cerr << e.what() << '\n';

    }

    const double endTime = MPI\_Wtime();

    std::cout << "Total time: " << endTime - startTime << "\n";

    deleteData(A, B, C);

    return 0;

}

### prog2.cpp (параллельная программа)

#include <iostream>

#include <mpi.h>

#include <string>

// P1 \* P2 - размеры двумерной решётки процессов

// N1 \* N2 - размер матрицы A

// N2 \* N3 - размер матрицы B

// N1 \* N3 - размер матрицы C

int P1 = 2; // Количество горизонтальных полос, на которые делатся матрица A

int P2 = 2; // Количество вертикальных полос, на которые делатся матрица B

int N1 = 4;

int N2 = 4;

int N3 = 4;

int DY = N1 / P1; // Высота горизонтальных полос

int DX = N3 / P2; // Ширина вертикальных полос

// Класс исключений для отслеживания ошибок

class myException : public std::exception

{

private:

    const char \*msg\_;

public:

    explicit myException(const char \*msg) : msg\_(msg){};

    const char \*what() const noexcept override { return msg\_; }

};

// Инициализация размеров матриц (по умолчанию есть заданные размеры, но их можно явно указать через аргументы командной строки)

inline void initSizes(const int argc, char \*\*argv)

{

    if (argc >= 6)

    {

        N1 = std::stoi(argv[1]);

        N2 = std::stoi(argv[2]);

        N3 = std::stoi(argv[3]);

        P1 = std::stoi(argv[4]);

        P2 = std::stoi(argv[5]);

    }

    if (N1 % P1 != 0 || N3 % P2 != 0)

        throw myException("The dimensions of the matrix must be a multiple of the dimensions of the two-dimensional grid");

    DY = N1 / P1;

    DX = N3 / P2;

}

// Освобождение памяти

inline void deleteData(double \*&A, double \*&B, double \*&C, double \*&LocA, double \*&LocB, double \*&LocC)

{

    free(A);

    free(B);

    free(C);

    free(LocA);

    free(LocB);

    free(LocC);

}

// Инициализация матриц для корневого процесса (именно здесь задаются значения в матрицах)

inline void initRootData(double \*&A, double \*&B, double \*&C, double \*&LocA, double \*&LocB, double \*&LocC)

{

    A = (double \*)malloc(sizeof(double) \* N1 \* N2);

    B = (double \*)malloc(sizeof(double) \* N2 \* N3);

    C = (double \*)malloc(sizeof(double) \* N1 \* N3);

    LocA = (double \*)malloc(sizeof(double) \* DY \* N2);

    LocB = (double \*)malloc(sizeof(double) \* N2 \* DX);

    LocC = (double \*)malloc(sizeof(double) \* DY \* DX);

    if (A == nullptr || B == nullptr || C == nullptr || LocA == nullptr || LocB == nullptr || LocC == nullptr)

    {

        deleteData(A, B, C, LocA, LocB, LocC);

        throw myException("Malloc fail");

    }

    for (int y = 0; y < N1; ++y)

    {

        for (int x = 0; x < N2; ++x)

        {

            A[y \* N2 + x] = y \* N2 + x;

        }

    }

    for (int y = 0; y < N2; ++y)

    {

        for (int x = 0; x < N3; ++x)

        {

            B[y \* N3 + x] = y \* N3 + x;

        }

    }

}

// Инициализация матриц для не корневых процессов

inline void initSideData(double \*&LocA, double \*&LocB, double \*&LocC)

{

    LocA = (double \*)malloc(sizeof(double) \* DY \* N2);

    LocB = (double \*)malloc(sizeof(double) \* N2 \* DX);

    LocC = (double \*)malloc(sizeof(double) \* DY \* DX);

    if (LocA == nullptr || LocB == nullptr || LocC == nullptr)

    {

        free(LocA);

        free(LocB);

        free(LocC);

        throw myException("Malloc fail");

    }

}

// Вычисления (LocC = LocA \* LocB)

inline void calculate(double \*const LocA, double \*const LocB, double \*const LocC)

{

    for (int i = 0; i < DY; ++i)

    {

        for (int j = 0; j < DX; ++j)

        {

            LocC[i \* DX + j] = 0;

            for (int k = 0; k < N2; ++k)

            {

                LocC[i \* DX + j] += LocA[i \* N2 + k] \* LocB[k \* DX + j];

            }

        }

    }

}

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

{

    MPI\_Init(&argc, &argv);

    const double startTime = MPI\_Wtime();

    int rank, size;

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

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

    double \*A = nullptr, \*B = nullptr, \*C = nullptr;

    double \*LocA = nullptr, \*LocB = nullptr, \*LocC = nullptr;

    try

    {

        initSizes(argc, argv);

        if (rank == 0 && P1 \* P2 != size)

            throw myException("The grid size should correspond to the number of processes");

        if (rank == 0)

            initRootData(A, B, C, LocA, LocB, LocC);

        else

            initSideData(LocA, LocB, LocC);

    }

    catch (const std::exception &e)

    {

        if (rank == 0)

            std::cerr << e.what() << '\n';

        return 0;

    }

    // Создание коммуникаторов для строк и столбцов

    int rankX = rank % P2;

    int rankY = rank / P2;

    MPI\_Comm commColumn; // Коммуникатор по столбцу

    MPI\_Comm commRow;    // Коммуникатор по строке

    MPI\_Comm\_split(MPI\_COMM\_WORLD, rankX, rank, &commColumn);

    MPI\_Comm\_split(MPI\_COMM\_WORLD, rankY, rank, &commRow);

    // Нулевой процесс разделяет матрицу A на горизонтальные полосы и передаёт их нулевому столбцу процессов

    if (rankX == 0)

        MPI\_Scatter(A, DY \* N2, MPI\_DOUBLE, LocA, DY \* N2, MPI\_DOUBLE, 0, commColumn);

    // Горизонтальные полосы матрицы A раздаются от каждого элемента нулевого столбца процессов всем процессам в соответствующей строке

    MPI\_Bcast(LocA, DY \* N2, MPI\_DOUBLE, 0, commRow);

    // Новый тип для хранения целого столбца матрицы B

    MPI\_Datatype columnType;

    // count - число блоков; blocklength - число элементов базового типа в каждом блоке

    // stride - шаг между началами соседних блоков, измеренный числом элементов базового типа; oldtype - базовый тип данных

    MPI\_Type\_vector(N2, DX, N3, MPI\_DOUBLE, &columnType);

    MPI\_Type\_commit(&columnType);

    // Нулевой процесс разделяет матрицу B на вертикальнные столбцы и передаёт их нулевой строке процессов

    // Процессы из первой строки принимают данные

    if (rank == 0)

    {

        for (int p = 1; p < P2; ++p)

            MPI\_Send(B + DX \* p, 1, columnType, p, p, commRow);

        // Копирование нулевого столбца матрицы B в LocB для нулевого процесса

        for (int y = 0; y < N2; ++y)

        {

            for (int x = 0; x < DX; ++x)

                LocB[y \* DX + x] = B[y \* N3 + x];

        }

    }

    else if (rank < P2)

    {

        MPI\_Recv(LocB, N2 \* DX, MPI\_DOUBLE, 0, rank, commRow, MPI\_STATUS\_IGNORE);

    }

    // Вертикальные полосы матрицы B раздаются от каждого элемента нулевой строки процессов всем процессам в соответствующем столбце

    MPI\_Bcast(LocB, N2 \* DX, MPI\_DOUBLE, 0, commColumn);

    calculate(LocA, LocB, LocC);

    // Новый тип для того, чтобы можно было удобно собирать части матрицы C без использования промежуточного буфера

    MPI\_Datatype submatrixType; // По сути это 1/P1 часть от columnType

    // count - число блоков; blocklength - число элементов базового типа в каждом блоке

    // stride - шаг между началами соседних блоков, измеренный числом элементов базового типа; oldtype - базовый тип данных

    MPI\_Type\_vector(DY, DX, N3, MPI\_DOUBLE, &submatrixType);

    MPI\_Type\_commit(&submatrixType);

    // Сбор частей матрицы C в одну большую матрицу C на нулевом процессе

    if (rank != 0)

        MPI\_Send(LocC, DY \* DX, MPI\_DOUBLE, 0, rank, MPI\_COMM\_WORLD);

    else

    {

        // Копирование LocC в левый верхний угол C для нулевого процесса

        for (int y = 0; y < DY; ++y)

        {

            for (int x = 0; x < DX; ++x)

                C[y \* N3 + x] = LocC[y \* DX + x];

        }

        for (int p = 1; p < size; ++p)

        {

            const int x = p % P2;

            const int y = p / P2;

            MPI\_Recv(C + N3 \* DY \* y + DX \* x, 1, submatrixType, p, p, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        }

    }

    // Вывод времени работы программы

    // MPI\_Barrier(MPI\_COMM\_WORLD);

    const double endTime = MPI\_Wtime();

    if (rank == 0)

        std::cout << "Total time: " << endTime - startTime << "\n";

    deleteData(A, B, C, LocA, LocB, LocC);

    MPI\_Type\_free(&columnType);

    MPI\_Type\_free(&submatrixType);

    MPI\_Finalize();

    return 0;

}