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

«УМНОЖЕНИЕ МАТРИЦЫ НА МАТРИЦУ В MPI 2D РЕШЕТКА»

студента 2 курса, группы 19212

Хомченко Станислава Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Ажбаков Артём Альбертович

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

Была написана программа (см. Приложение 1), выполняющая умножение матриц с использованием 2D решётки процессов MPI.

Компиляция выполнялась командой:

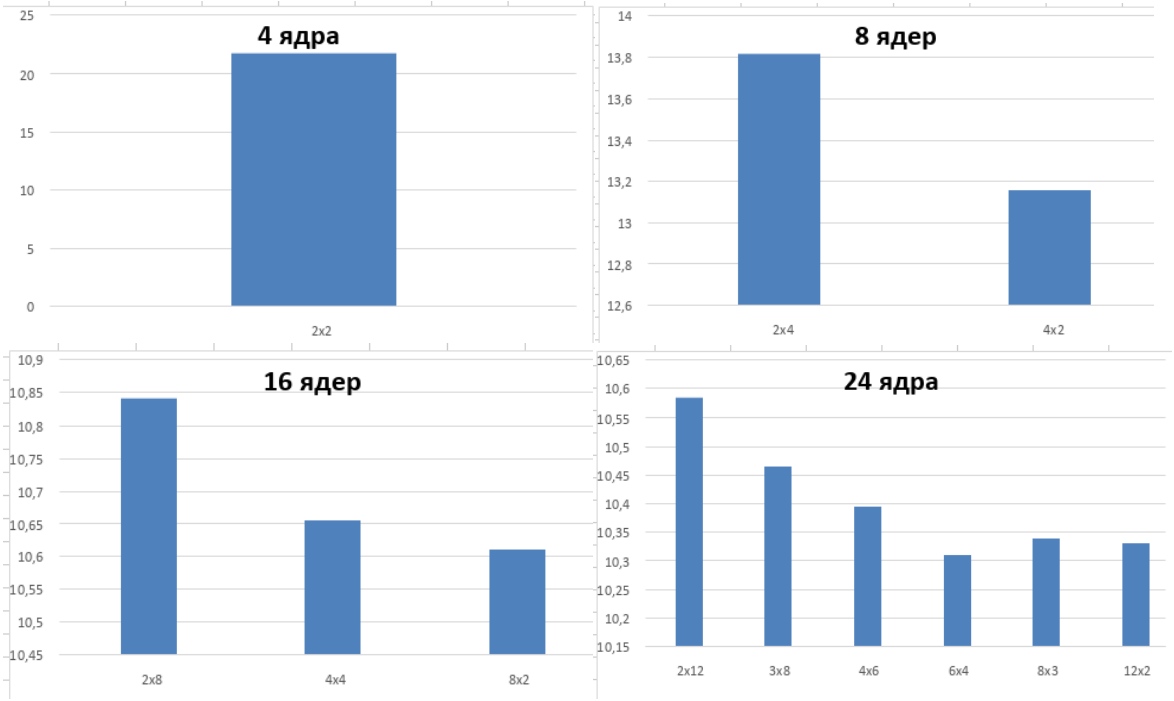
```
mpicxx -O3 lab3.cpp -o lab3 -std=c++11
```

Запуск на кластере выполнялся при помощи скрипта:

```
#!/bin/bash
#PBS -l walltime=00:05:00
#PBS -l select=2:ncpus=12:mpiprocs=12:mem=3000m,place=scatter:exclhost
cd $PBS_O_WORKDIR
MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI process: $MPI_NP"
echo 'File $PBS_NODEFILE:'
cat  $PBS_NODEFILE
echo
mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./lab3 2400 4800 7200 matrix.txt 2 2
```

Выполнен замер времени при различных размерах решетки. Размеры матриц были взяты: A = 2400x4800 и B = 4800x7200. Матрицы считывались из файла, заполненного случайными числами.

Результат можно увидеть на графиках.



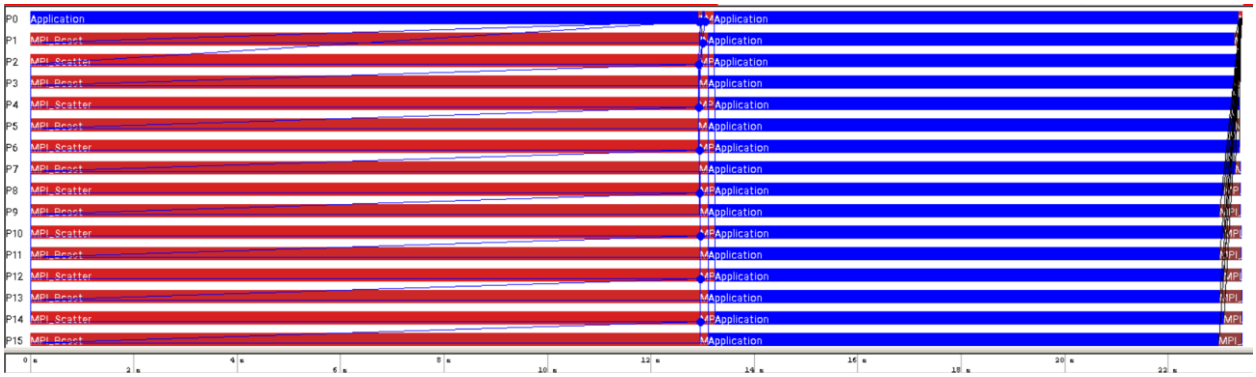
Также было выполнено профилирование программы на 16 ядрах.

Для профилирования программа компилировалась при помощи команды:

```
mpiicc -O3 lab3.cpp -o lab3 -std=c++11
```

После запуска программы был сгенерирован файл формата stf, который анализировался программой traceanalyzer.

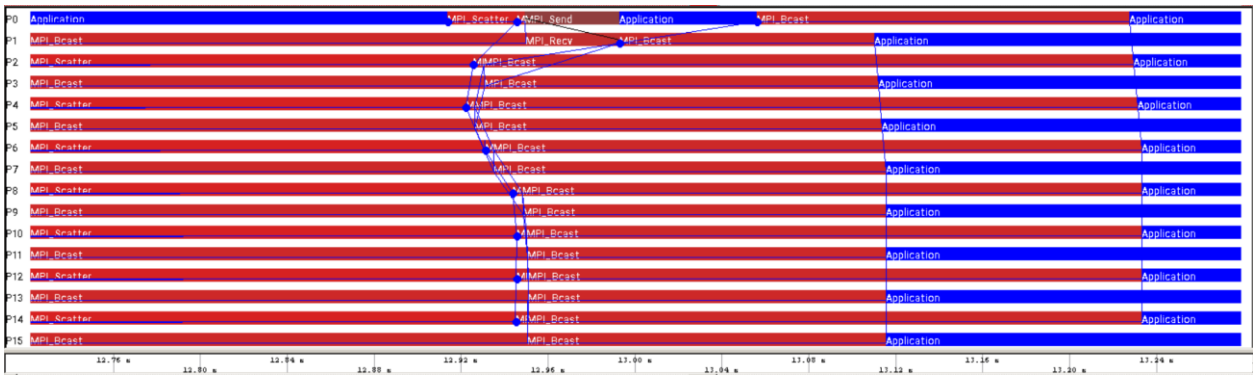
Профилирование решётки 2x8.



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	173.268 s		374.483 s	16	10.8292 s
MPI_Cart_sub	1.629e-3 s		1.629e-3 s	32	50.9062e-6 s
MPI_Comm_size	11e-6 s		11e-6 s	16	687.5e-9 s
MPI_Comm_rank	7e-6 s		7e-6 s	16	437.5e-9 s
MPI_Finalize	5.413e-3 s		5.413e-3 s	16	338.312e-6 s
MPI_Scatter	90.5534 s		90.5534 s	8	11.3192 s
MPI_Bcast	107.072 s		107.072 s	32	3.346 s
MPI_Recv	105.25e-3 s		105.25e-3 s	16	6.57812e-3 s
MPI_Cart_create	2.274e-3 s		2.274e-3 s	16	142.125e-6 s
MPI_Type_free	14e-6 s		14e-6 s	2	7e-6 s
MPI_Type_commit	30e-6 s		30e-6 s	2	15e-6 s
MPI_Cart_coords	154e-6 s		154e-6 s	31	4.96774e-6 s
MPI_Send	3.47428 s		3.47428 s	16	217.142e-3 s
MPI_Type_vector	40e-6 s		40e-6 s	2	20e-6 s

Сначала 0-й процесс выполняет считывание матриц из файла, а остальные процессы сразу переходят к следующим действиям:

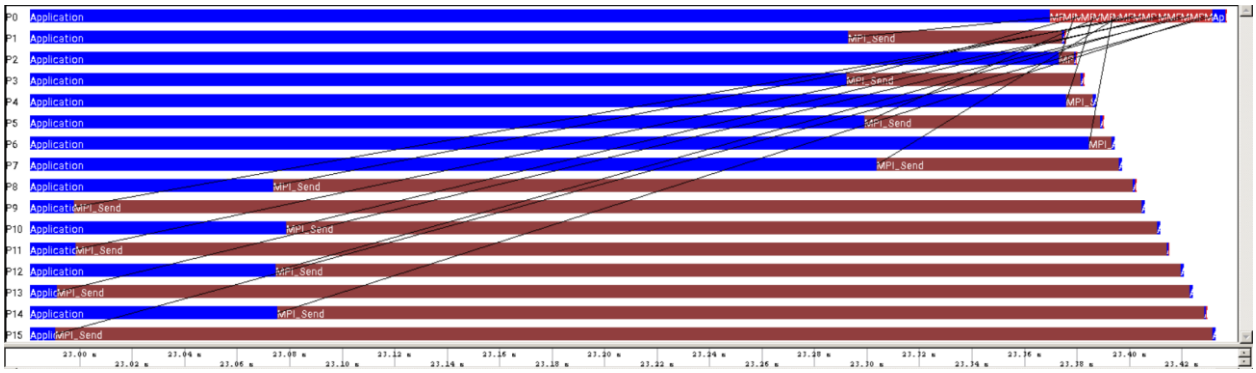
- 2, 4, 6, 8, 10, 12 и 14 процессы ждут, пока 0-й процесс дойдёт до Scatter для разрезания и раздачи матрицы A (именно эти процессы расположены по оси y).
- 1, 3, 5, 7, 9, 11, 13 и 15 процессы ждут Bcast, т.е. когда процессы в соответствующих строках распространят части матрицы A.



Далее, так как в решётке только 2 столбца процессов, то 0-й процесс делает один Send части матрицы B 1-му процессу, который его принимает через Recv.

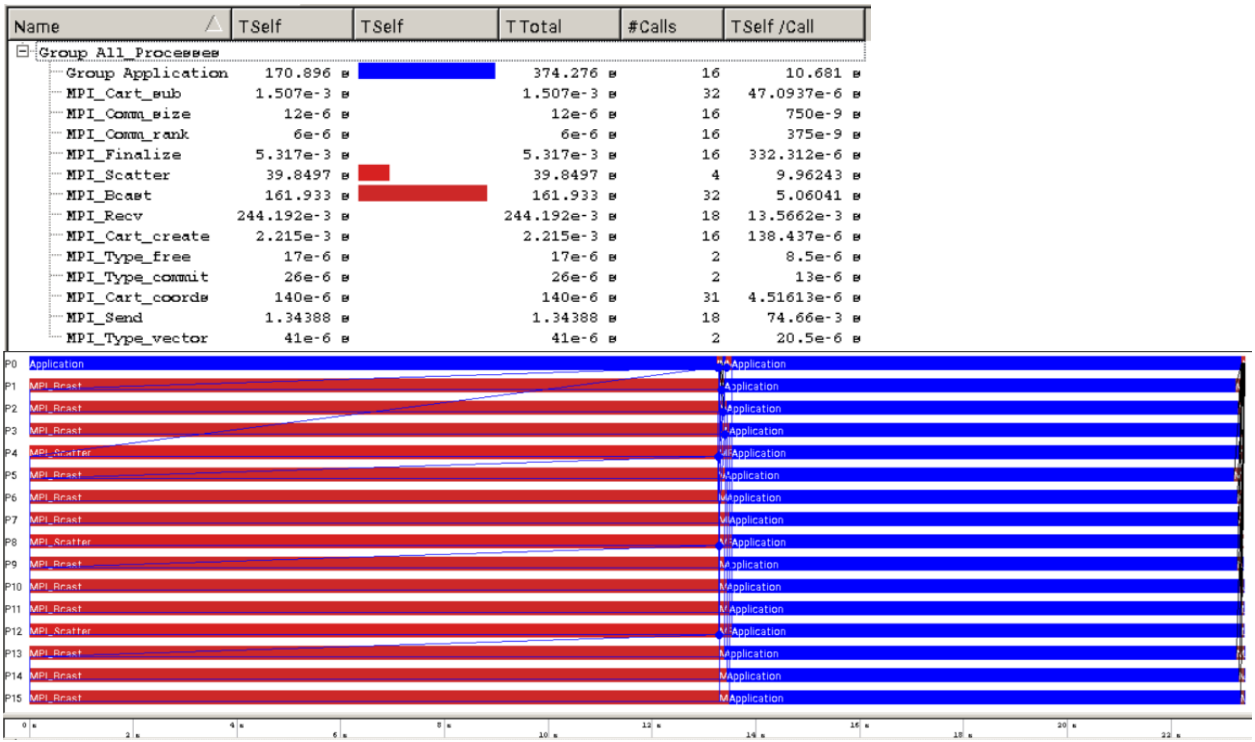
Потом 0 и 1 процессы распространяют свои части матрицы B другим процессам в соответствующих столбцах при помощи Bcast.

Далее процессы перемножают свои части матриц A и B.



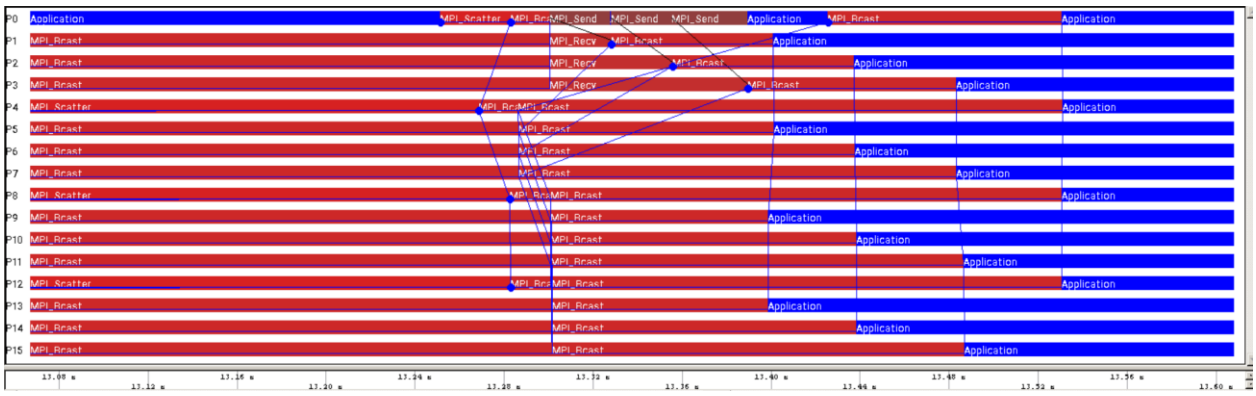
После перемножения все процессы отправляют часть матрицы C 0-му процессу.

Профилирование решётки 4x4.

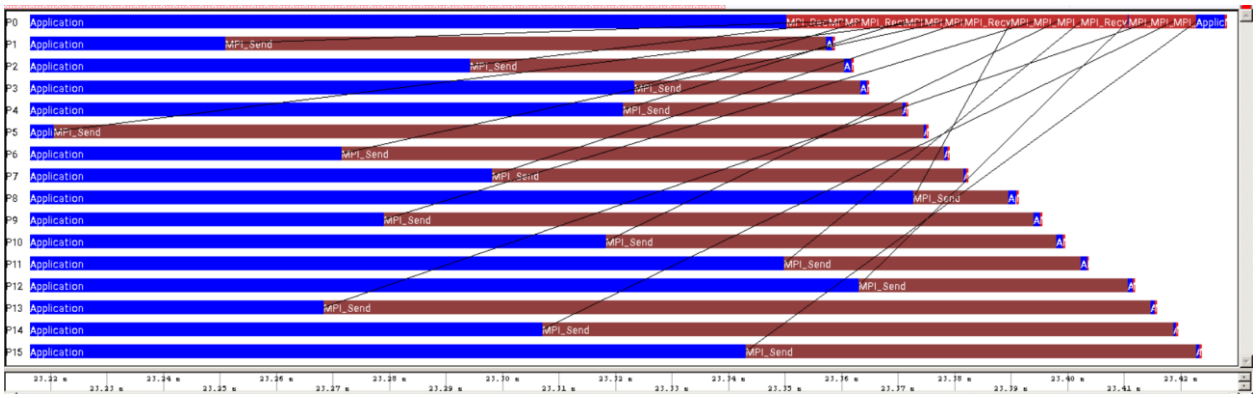


Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	170.896 s		374.276 s	16	10.681 s
MPI_Cart_sub	1.507e-3 s		1.507e-3 s	32	47.0937e-6 s
MPI_Comm_size	12e-6 s		12e-6 s	16	750e-9 s
MPI_Comm_rank	6e-6 s		6e-6 s	16	375e-9 s
MPI_Finalize	5.317e-3 s		5.317e-3 s	16	332.312e-6 s
MPI_Scatter	39.8497 s		39.8497 s	4	9.96243 s
MPI_Bcast	161.933 s		161.933 s	32	5.06041 s
MPI_Recv	244.192e-3 s		244.192e-3 s	18	13.5662e-3 s
MPI_Cart_create	2.215e-3 s		2.215e-3 s	16	138.437e-6 s
MPI_Type_free	17e-6 s		17e-6 s	2	8.5e-6 s
MPI_Type_commit	26e-6 s		26e-6 s	2	13e-6 s
MPI_Cart_coords	140e-6 s		140e-6 s	31	4.51613e-6 s
MPI_Send	1.34388 s		1.34388 s	18	74.66e-3 s
MPI_Type_vector	41e-6 s		41e-6 s	2	20.5e-6 s

Здесь уже 4 строчки в решётке процессов, поэтому Scatter ждут только 3 процесса (4-й, 8-й и 12-й). Остальные процессы ждут, аналогично предыдущей решетке, раздачу частей матрицы A в соответствующих строчках решётки.



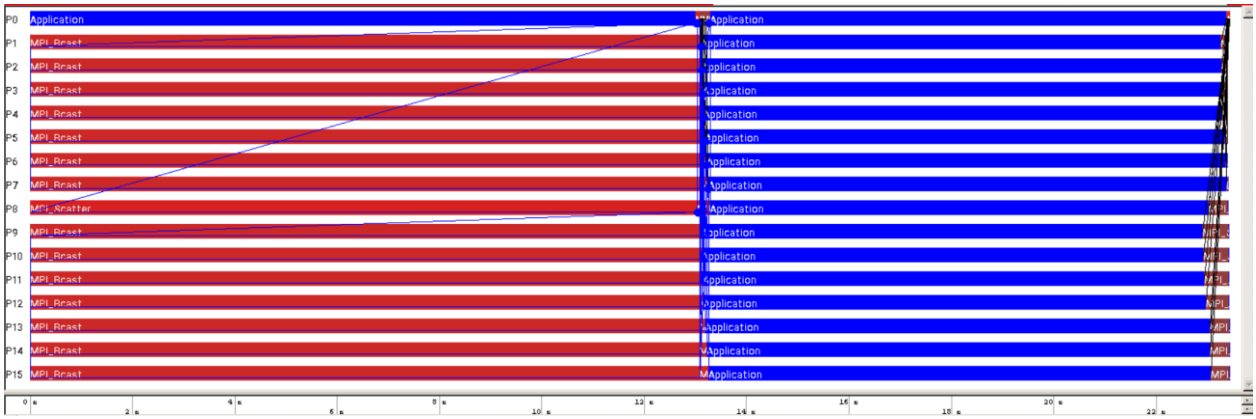
Здесь 0-й процесс отправляет 4-му, 8-му и 12-му процессу части матрицы B, и происходит раздача этих частей по соответствующим столбцам решётки.



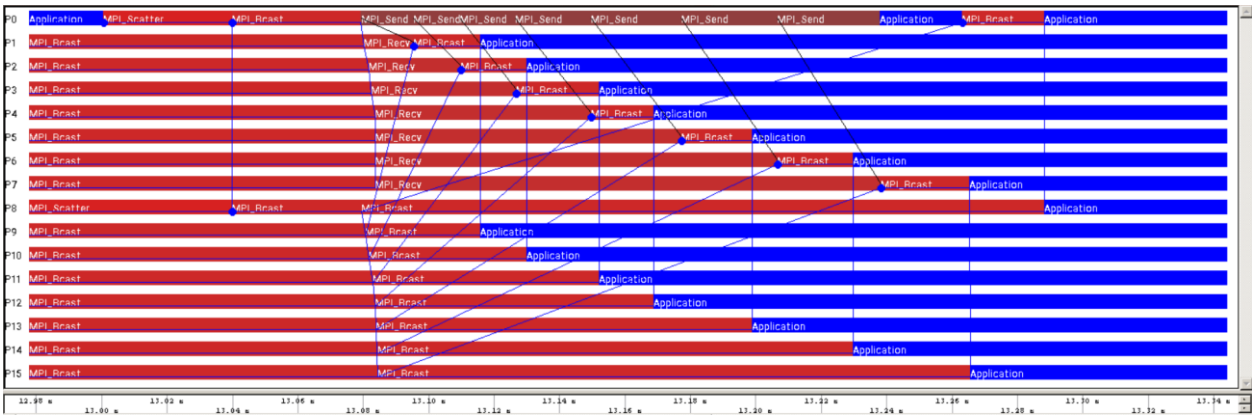
После перемножения все процессы отправляют части матрицы C 0-му процессу.

Профилирование решётки 8x2.

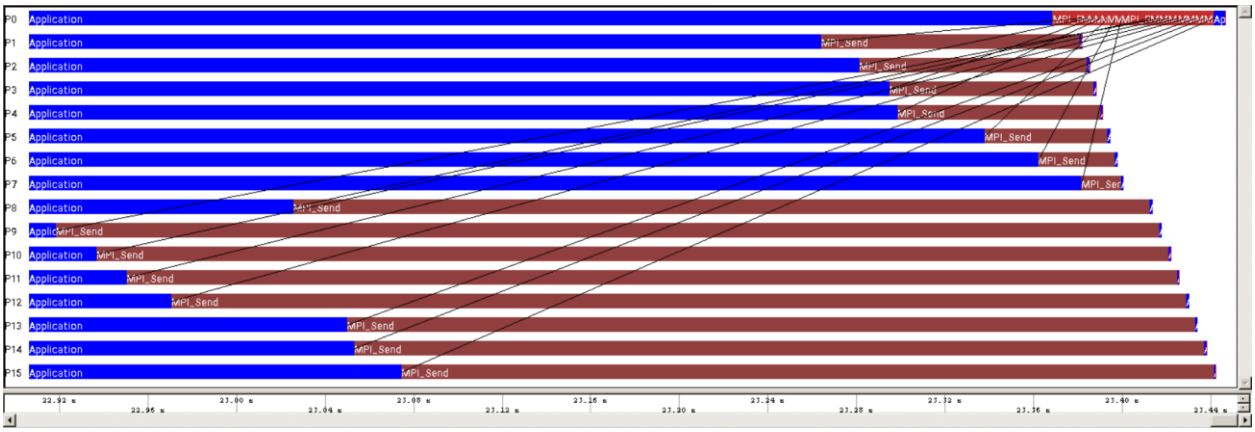
Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	172.602 s		374.607 s	16	10.7876 s
MPI_Cart_sub	1.678e-3 s		1.678e-3 s	32	52.4375e-6 s
MPI_Comm_size	10e-6 s		10e-6 s	16	625e-9 s
MPI_Comm_rank	7e-6 s		7e-6 s	16	437.5e-9 s
MPI_Finalize	5.228e-3 s		5.228e-3 s	16	326.75e-6 s
MPI_Scatter	13.0735 s		13.0735 s	2	6.53673 s
MPI_Bcast	184.219 s		184.219 s	32	5.75684 s
MPI_Recv	597.373e-3 s		597.373e-3 s	22	27.1533e-3 s
MPI_Cart_create	2.309e-3 s		2.309e-3 s	16	144.312e-6 s
MPI_Type_free	14e-6 s		14e-6 s	2	7e-6 s
MPI_Type_commit	43e-6 s		43e-6 s	2	21.5e-6 s
MPI_Cart_coords	138e-6 s		138e-6 s	31	4.45161e-6 s
MPI_Send	4.10583 s		4.10583 s	22	186.629e-3 s
MPI_Type_vector	48e-6 s		48e-6 s	2	24e-6 s



Здесь уже 2 строки процессов в решётке, поэтому два процесса будут участвовать в Scatter (0-й и 8-й).



В разрезании и раздаче матрицы B участвует уже 8 процессов.



После перемножения все процессы отправляют свою часть матрицы C 0-му процессу.

ЗАКЛЮЧЕНИЕ

Я научился работать с решётками процессов в MPI, создавать коммуникаторы и производные типы данных и взаимодействовать с ними.
Выполнен анализ зависимости размера решётки от времени работы программы: чем ближе форма решётки к квадрату, тем быстрее работает программа.

Приложение 1. *Код программы*

```
#include <mpi.h>
#include <cstdio>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define IDX(r, c, rows, cols) (r * cols + c)

void init_matrices(const char *fname, double **A, double **B, double **C, int
n1, int n2, int n3) {
    FILE *file = fopen(fname, "r");
    *A = (double *)calloc(n1 * n2, sizeof(double));
    *B = (double *)calloc(n2 * n3, sizeof(double));
    *C = (double *)calloc(n1 * n3, sizeof(double));
    for (int i = 0; i < n1 * n2; i++) fscanf(file, "%lf", &(*A)[i]);
    for (int i = 0; i < n2 * n3; i++) fscanf(file, "%lf", &(*B)[i]);
    fclose(file);
}

void multiply_matrices(const double *A, const double *B, double *C, int n1,
int n2, int n3) {
    for (int i = 0; i < n1; i++)
        for (int k = 0; k < n2; k++)
            for (int j = 0; j < n3; j++)
                C[IDX(i, j, n1, n3)] += A[IDX(i, k, n1, n2)] * B[IDX(k, j,
n2, n3)];
}

int main(int argc, char **argv) {
    if (argc != 7) {
        printf("Usage: %s size1 size2 size3 filename dimx dimy\n", argv[0]);
        return 0;
    }

    MPI_Init(&argc, &argv);

    int proc_cnt; //число процессов
    MPI_Comm_size(MPI_COMM_WORLD, &proc_cnt); //получение числа процессов

    MPI_Comm comm_cart; //коммуникатор, наделенный декартовой топологией
    const int ndims = 2; //размерность декартовой решетки; 2 - плоская
двумерная решётка
    const int DIM_X = atoi(argv[5]); // Dim_x - количество процессов в
решетке вдоль оси x
    const int DIM_Y = atoi(argv[6]);
    int dims[ndims] = { DIM_Y, DIM_X }; //ndims - размерность нового
пространства(декартовой решётки)
                                        //dims - целочисленный массив,
состоящий из ndims элементов, задающий количество процессов в каждом
измерении
    int periods[ndims] = { 0, 0 }; // periods - массив, указывающий
периодичность границ по каждому направлению
    int reorder = 1; // разрешать или нет MPI изменять порядок процессов
    MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder,
&comm_cart); // создание решетки
    /*
    MPI_Cart_create
    • Используется для задания декартовых координат процессам произвольной
размерности пространства
    • Позволяет построить нумерацию процессов по координатам в сетке, а не по
их номеру
    • Позволяет делать циклические условия на границах
```

```
    • В ряде случаев позволяет стоить топологию на основе реального железа
    Функция MPI_CART_CREATE возвращает дескриптор нового коммуникатора, к
которому подключается
    топологическая информация. Если reorder = false, то номер каждого
процесса в новой группе идентичен
    номеру в старой группе. В противном случае функция может
переупорядочивать процессы (возможно,
    чтобы обеспечить хорошее наложение виртуальной топологии на физическую
систему). Если полная
    размерность декартовой решетки меньше, чем размер группы коммуникаторов,
то некоторые процессы
    возвращаются с результатом MPI_COMM_NULL по аналогии с MPI_COMM_SPLIT.
Вызов будет неверным,
    если он задает решетку большего размера, чем размер группы.
     */

    int coords[ndims]; // coords-Одномерный массив, содержащий декартовые
координаты процесса
    int rank, rankx, ranky; // Rank - это просто ранк в топологии. пример:
    // 2x4 -> процессы 0123
    //                 4567
    // у процесса 7 rankx = 3; ranky = 1;
    MPI_Comm_rank(comm_cart, &rank); // Получение номера процесса
    MPI_Cart_coords(comm_cart, rank, ndims, coords); //получение процесса по
его ранку
    /*
    • comm_cart - коммуникатор
    • rank — ранк процессора, координаты которого узнаем
    • ndims— размерность декартовой топологии
    • coords — массив с координатами
    */
    ranky = coords[0]; rankx = coords[1]; //задаем декартовые координаты
процесса

    MPI_Comm comm_row, comm_col;
    int subdims[ndims]; // определяют, принадлежит ли определенное измерение
новому коммуникатору
    subdims[0] = 0; subdims[1] = 1; // изменяется значение по y
    MPI_Cart_sub(comm_cart, subdims, &comm_row);
    /*
    процедура разбиения решетки на подрешетки меньшей размерности
    Еще одним способом является деление сетки на участки меньшей размерности.
Например, можно создать
    коммуникаторы для каждой строки и строки сетки
    Вызов MPI_Cart_sub() создает q новых коммуникаторов. Аргумент subdims
является массивом
    логических значений. Они определяют, принадлежит ли определенное
измерение новому
    коммуникатору. Поскольку в примере создаются коммуникаторы для строк
сетки, то каждый новый
    коммуникатор состоит из процессов, получающих фиксированную координату
строки, при этом
    позволяя координате колонки изменяться. Поэтому subdims[0] принимает
значение 0 - первая
    координата не изменяется, а
    subdims [1] принимает значение 1 - вторая координата изменяется. В каждом
процессе
    возвращается новый коммуникатор row_comm.
    */
    subdims[0] = 1; subdims[1] = 0; // изменяется значение по x
    MPI_Cart_sub(comm_cart, subdims, &comm_col);

    int n1 = atoi(argv[1]), n2 = atoi(argv[2]), n3 = atoi(argv[3]);
    double *A = NULL, *B = NULL, *C = NULL;
```

```
    if (rank == 0) init_matrices(argv[4], &A, &B, &C, n1, n2, n3);

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    int local_n1 = n1 / DIM_Y, local_n3 = n3 / DIM_X;
    /*
     У нас общие размеры матриц n1xn2, n2xn3 и n1xn3.
    Мы матрицу A режем по строчкам, всего n1 строчка, бьём на блоки по
local_n1 строк.
    Аналогично с матрицей B - ее режем по столбцам.
    В каждом процессе блок матрицы A. Количество не изменилось - n2, а
количество стрк - local_n1
     */
    double *partA = (double *)calloc(local_n1 * n2, sizeof(double));
//подматрица (блок)
    double *partB = (double *)calloc(n2 * local_n3, sizeof(double));

    if (rankx == 0) MPI_Scatter(A, local_n1 * n2, MPI_DOUBLE, partA, local_n1
* n2, MPI_DOUBLE, 0, comm_col);
    /*
    Раздача матрицы A по горизонтальным полосам на вертикальную линейку
процессов (0;0), (1;0), (2;0), …,
    (p1 - 1; 0) при помощи MPI_Scatter

    IN   A начальный адрес буфера рассылки (альтернатива, используется
         только корневым процессом)

    IN   Local_n1*n2 количество элементов, посылаемых каждому процессу
(целое,
         используется только корневым процессом)

    IN   MPI_DOUBLE тип данных элементов в буфере посылки (дескриптор,
         используется только корневым процессом)

    OUT  partA адрес буфера процесса-получателя (альтернатива)

    IN   local_n1 * n2 количество элементов в буфере корневого (целое)

    IN   MPI_DOUBLE тип данных элементов приемного буфера (дескриптор)

    IN   0 номер процесса-получателя (целое)

    IN   comm_col коммуникатор (дескриптор)
    */
    MPI_Bcast(partA, local_n1 * n2, MPI_DOUBLE, 0, comm_row);
    /*
    Каждый из процессов в левой вертикальной колонке ( (1;0), (2;0), …, (p1 -
1; 0) ) при помощи MPI_Bcast
    раздает свою полосу матрицы A всем процессам своей горизонтали. Т.е.
процесс (1;0) раздает свою полосу
    процессам (1;1), (1;2),…
    */

    if (rank == 0) {
        MPI_Datatype MAT_B_BLOCK;
        MPI_Type_vector(n2, local_n3, n3, MPI_DOUBLE, &MAT_B_BLOCK);
        /*
        IN count  число блоков (неотрицательное целое)
        IN blocklength    число элементов в каждом блоке (неотрицательное
целое)
        IN stride число элементов между началами каждого блока (целое)
        IN oldtype    старый тип данных (дескриптор)
        OUT    newtype    новый тип данных (дескриптор)
```

```
        */
        MPI_Type_commit(&MAT_B_BLOCK); // регистрация нового типа

        for (int i = 1; i < DIM_X; i++) {
            MPI_Send(B + i * local_n3 , 1, MAT_B_BLOCK, i, 120, comm_row);
        }
        /*
        На вход принимает адрес буфера с сообщением buf, количество данных в
буфере count, тип
        данных datatype, ранг процесса получателя dest, тег сообщения tag,
коммуникатор в котором
        происходит обмен comm, оба процесса(отправитель и получатель) должны
находиться в этом
        коммуникаторе.
        */
        for (int i = 0; i < n2; i++) {
            for (int j = 0; j < local_n3; j++) {
                partB[IDX(i, j, n2, local_n3)] = B[IDX(i, j, n2, n3)];
            }
        }

        MPI_Type_free(&MAT_B_BLOCK);
    }
    else if (ranky == 0) {
        MPI_Recv(partB, n2 * local_n3, MPI_DOUBLE, 0, 120, comm_row,
MPI_STATUS_IGNORE);
        /*
        Ловить с другой стороны наш буфер будет функция MPI_Recv
        На вход принимает адрес буфера buf для полученного сообщения, который
должен смочь вместить в себя
        count элементов типа datatype, ранг процесса источника source, тег
сообщения tag, коммуникатор comm.
        Отправитель и получатель должны находиться в одном коммуникаторе.
Последний аргумент status — статус
        посылки.
        */
    }
    MPI_Bcast(partB, n2 * local_n3, MPI_DOUBLE, 0, comm_col);

    double *partC = (double *)calloc(local_n1 * local_n3, sizeof(double));
    multiply_matrices(partA, partB, partC, local_n1, n2, local_n3);

    if (rank == 0) {
        MPI_Datatype MAT_C_BLOCK;
        MPI_Type_vector(local_n1, local_n3, n3, MPI_DOUBLE, &MAT_C_BLOCK);
        /*
        Создает новый тип, элементы которого представляют собой несколько
        равноудаленных друг от друга блоков из одинакового числа смежных
        элементов базового типа.
        Local_n1- число блоков
        Local_n3- число элементов базового типа в каждом блоке
        N3- шаг между началами соседних блоков, измеренный числом элементов
        базового типа
        Mpi_double- базовый тип данных
        &amp;mat_c_block- новый производный тип данных
        */
        MPI_Type_commit(&MAT_C_BLOCK);

        for (int i = 1; i < proc_cnt; i++) {
            MPI_Cart_coords(comm_cart, i, ndims, coords);
            int procy = coords[0], procx = coords[1];
            MPI_Recv(C + procy * local_n1 * n3 + procx * local_n3, 1,
MAT_C_BLOCK, i, 121, comm_cart, MPI_STATUS_IGNORE);
        }
```

```
        MPI_Type_free(&MAT_C_BLOCK); //уничтожает описатель производного типа

        for (int i = 0; i < local_n1; i++) {
            for (int j = 0; j < local_n3; j++) {
                C[IDX(i, j, n1, n3)] = partC[IDX(i, j, local_n1, local_n3)];
            }
        }
    }
    else MPI_Send(partC, local_n1 * local_n3, MPI_DOUBLE, 0, 121, comm_cart);

    clock_gettime(CLOCK_MONOTONIC, &end);

    if (rank == 0) {
        printf("Time taken: %lf sec.\n", end.tv_sec - start.tv_sec +
0.000000001 * (end.tv_nsec - start.tv_nsec));
        if (1) {
            for (int i = 0; i < n1; i++) {
                for (int j = 0; j < n3; j++)
                    printf(" %lf ", C[IDX(i, j, n1, n3)]);
                printf("\n");
            }
        }
        free(A); free(B); free(C);
    }
    free(partA); free(partB); free(partC);

    MPI_Finalize();
    return 0;
}
```