

**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 матриц средствами MPI на декартовой решётке процессов»

студента 2 курса, группы 20205

Муратова Максима Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

**Преподаватель:
доцент
Власенко А. Ю.**

Новосибирск, 2021

СОДЕРЖАНИЕ

ЦЕЛИ.....	3
ЗАДАНИЕ.....	3
ОПИСАНИЕ РАБОТЫ.....	4
ЗАКЛЮЧЕНИЕ.....	7
Приложение 1. <i>Код программы</i>	8
Приложение 2. <i>Makefile</i>	12
Приложение 3. <i>Скрипты для кластера</i>	13

ЦЕЛИ

1. Найти зависимость времени выполнения параллельной программы от размеров декартовой решётки процессов при одинаковом числе процессов;
2. Проанализировать, как эффективно используется время для выполнения программы.

ЗАДАНИЕ

1. Вычислительное задание – умножение двух матриц A и B размерами $N \times K$ и $K \times M$ соответственно;
2. $N=3000$, $M=3600$, $N=4200$;
3. Число процессов – 24. Решётки – 2×12 , 3×8 , 4×6 , 6×4 , 8×3 , 12×2 ;
4. Профилировщиком Intel Trace Analyzer and Collector (ITAC) пропрофиллировать программу на решётках 2×4 и 4×2 .

ОПИСАНИЕ РАБОТЫ

Исходный код программы содержится в единственном файле `parallel.cpp` (приложение 1). Программа сама позаботится о том, чтобы число процессов было равно размеру решётки. При разных соотношениях длин сторон решётки итоговая матрица $C=A \times B$ собирается по-разному: сборка начинается с более «длинной» стороны. Сборку программы можно осуществить при помощи `Makefile` (приложение 2) одной из следующих команд:

```
$ make
```

или

```
$ make parallel
```

В приложении 3 описаны скрипты для постановки задачи в очередь выполнения кластером, причём как для вычисления на разных решётках, так и для профилирования. Чтобы поставить задачу вычисления на разных решётках в очередь, достаточно выполнить команду:

```
$ qsub run.sh
```

Будет произведено умножение матриц на заданных решётках в 3 круга, их время выполнения записывается в файл `report.txt`. Умножение на разных решётках специально объединено в один скрипт, чтобы умножения производились на одних и тех же узлах кластера. На эту задачу выделяется 18 минут. Скрипт не нуждается в модификации при смене размерности решётки по указанной выше причине.

Для выполнения умножения матриц с профилированием необходимо выполнить следующую команду:

```
$ qsub trace.sh
```

Данный скрипт требует модификации при изменении размерности решётки.

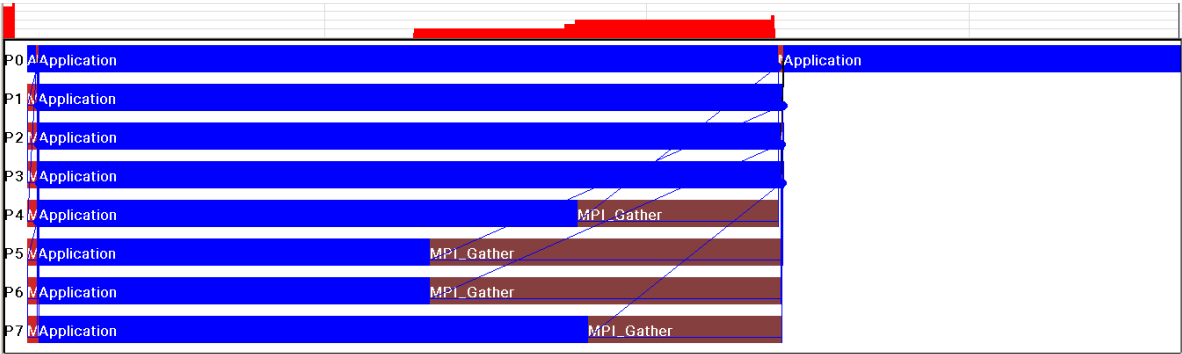
Были получены следующие результаты замеров времени при разных решётках (только лучшие результаты):

Решётка	Время, с
2x12	45,163
3x8	45,145
4x6	37,6696
6x4	36,5377
8x3	41,669
12x2	42,5153

Промежуточный вывод: чем ближе решётка к квадратной, тем меньше время выполнения.

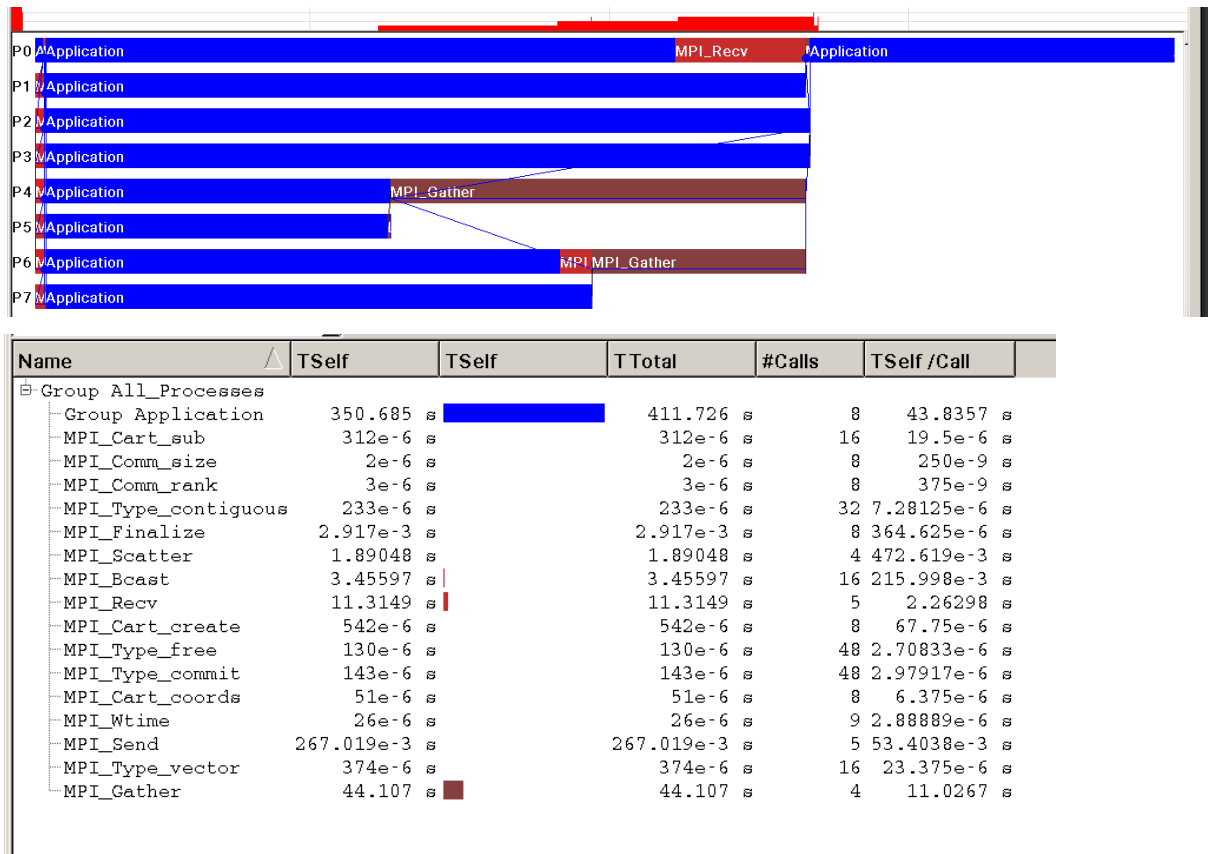
Поговорим о результатах профилирования:

1. 2x4



Name	TSelf	TSelf	T Total	#Calls	TSelf /Call
Group All_Processes					
Group Application	333.452 s		408.872 s	8	41.6816 s
MPI_Cart_sub	326e-6 s		326e-6 s	16	20.375e-6 s
MPI_Comm_size	3e-6 s		3e-6 s	8	375e-9 s
MPI_Comm_rank	2e-6 s		2e-6 s	8	250e-9 s
MPI_Type_contiguous	500e-6 s		500e-6 s	32	15.625e-6 s
MPI_Finalize	2.749e-3 s		2.749e-3 s	8	343.625e-6 s
MPI_Scatter	654.719e-3 s		654.719e-3 s	2	327.359e-3 s
MPI_Bcast	4.41469 s		4.41469 s	16	275.918e-3 s
MPI_Recv	477.338e-3 s		477.338e-3 s	6	79.5563e-3 s
MPI_Cart_create	535e-6 s		535e-6 s	8	66.875e-6 s
MPI_Type_free	129e-6 s		129e-6 s	48	2.6875e-6 s
MPI_Type_commit	152e-6 s		152e-6 s	48	3.16667e-6 s
MPI_Cart_coords	60e-6 s		60e-6 s	8	7.5e-6 s
MPI_Wtime	27e-6 s		27e-6 s	9	3e-6 s
MPI_Send	346.138e-3 s		346.138e-3 s	6	57.6897e-3 s
MPI_Type_vector	75e-6 s		75e-6 s	16	4.6875e-6 s
MPI_Gather	69.5225 s		69.5225 s	8	8.69031 s

2. 4x2



Вывод: в обоих случаях тратится много времени на Gather, который собирает матрицу C по столбцам или строкам (напомню, что соотношение сторон решётки влияет на алгоритм сборки матрицы C).

ЗАКЛЮЧЕНИЕ

1. При использовании декартовой решётки процессов для перемножения матриц следует использовать решётку с минимальной разницей длин сторон решётки (в идеале – квадратную решётку), чтобы добиться максимальной производительности;
2. Основная причина потери времени – ожидание данных при сборке итоговой матрицы C .

Приложение 1. *parallel.cpp*

Исходный код программы

```
#include <mpi.h>
#include <iostream>
#include <climits>

double random_double() {
    return rand() / (double) INT_MAX * 200 - 100;
}

std::ostream &print_matrix(const double *matrix, int rows, int columns, std::ostream &out)
{
    for(int x = 0; x < rows; x++) {
        out << matrix[x * columns];
        for(int y = 1; y < columns; y++)
            out << '\t' << matrix[x * columns + y];
        out << '\n';
    }
    return out;
}

int main(int argc, char **argv) {
    if(argc != 6) {
        std::cerr << "Got " << argc << " argument(s) instead of 5\n";
        return 1;
    }

    MPI_Init(&argc, &argv);
    double start_time = MPI_Wtime();
    int n = atoi(argv[1]);
    int k = atoi(argv[2]);
    int m = atoi(argv[3]);
    const int DIM = 2;
    int dims[DIM], periods[DIM], reorder = 0;
    int process_count; MPI_Comm_size(MPI_COMM_WORLD, &process_count);
    int non_grid_rank; MPI_Comm_rank(MPI_COMM_WORLD, &non_grid_rank);

    for(int index = 0; index < DIM; index++) {
        dims[index] = atoi(argv[index + 4]);
        periods[index] = 1;
    }
    if(process_count != dims[0] * dims[1]) {
        if(!non_grid_rank)
            std::cerr << "Expected " << (dims[0] * dims[1]) <<
                " processes, got " << process_count << '\n';
        MPI_Finalize();
        return 1;
    }
    if(n % dims[0] != 0) {
        if(!non_grid_rank)
            std::cerr << n << " N==" << n << " must be divisible by " <<
                dims[0] << '\n';
        MPI_Finalize();
        return 1;
    }
    if(m % dims[1] != 0) {
        if(!non_grid_rank)
            std::cerr << "M==" << m << " must be divisible by " << dims[1]
                << '\n';
    }
}
```



```

        MPI_Finalize();
        return 1;
    }
    MPI_Comm grid_comm, column_comm, row_comm;
    MPI_Cart_create(MPI_COMM_WORLD, DIM, dims, periods, reorder, &grid_comm);
    int coords[DIM], sub_dims[DIM];
    MPI_Cart_coords(grid_comm, non_grid_rank, DIM, coords);
    sub_dims[0] = 0; sub_dims[1] = 1;
    MPI_Cart_sub(grid_comm, sub_dims, &row_comm);
    sub_dims[0] = 1; sub_dims[1] = 0;
    MPI_Cart_sub(grid_comm, sub_dims, &column_comm);

    double *A = NULL, *B = NULL, *C = NULL;
    double *sub_A = NULL, *sub_B = NULL, *sub_C = NULL;
    int sub_n = n / dims[0];
    int sub_m = m / dims[1];
    sub_A = new double[sub_n * k];
    sub_B = new double[k * sub_m];
    sub_C = new double[sub_n * sub_m];

    if(!coords[0] && !coords[1]) {
        A = new double[n * k];
        B = new double[k * m];
        C = new double[n * m];
        srand(time(0));

        for(int index = 0; index < n * k; index++)
            A[index] = random_double();
        for(int index = 0; index < k * m; index++)
            B[index] = random_double();
    }

    MPI_Datatype SUB_A;
    MPI_Type_contiguous(sub_n * k, MPI_DOUBLE, &SUB_A);
    MPI_Type_commit(&SUB_A);
    if(!coords[1])
        MPI_Scatter(A, 1, SUB_A, sub_A, 1, SUB_A, 0, column_comm);
    MPI_Bcast(sub_A, 1, SUB_A, 0, row_comm);
    MPI_Type_free(&SUB_A);

    MPI_Datatype SUB_B;
    const int MSG_ID = 12;
    MPI_Type_vector(k, sub_m, m, MPI_DOUBLE, &SUB_B);
    MPI_Type_commit(&SUB_B);
    MPI_Datatype SUB_B_CONTIGUOUS;
    MPI_Type_contiguous(k * sub_m, MPI_DOUBLE, &SUB_B_CONTIGUOUS);
    MPI_Type_commit(&SUB_B_CONTIGUOUS);
    if(!coords[0] && !coords[1]) {
        for(int row = 0; row < k; row++) {
            for(int column = 0; column < sub_m; column++)
                sub_B[row * sub_m + column] = B[row * m + column];
        }
        for(int index = 1; index < dims[1]; index++)
            MPI_Send(B + sub_m * index, 1, SUB_B, index, MSG_ID, row_comm);
    }
    if(!coords[0] && coords[1]) {
        MPI_Status status;
        MPI_Recv(sub_B, 1, SUB_B_CONTIGUOUS, 0, MSG_ID, row_comm, &status);
    }
    MPI_Bcast(sub_B, 1, SUB_B_CONTIGUOUS, 0, column_comm);
    MPI_Type_free(&SUB_B);

```

```

MPI_Type_free(&SUB_B_CONTIGUOUS);

for(int row = 0; row < sub_n; row++) {
    for(int column = 0; column < sub_m; column++) {
        int current_row = row * sub_m;
        sub_C[current_row + column] = 0;
        for(int index = 0; index < k; index++) {
            sub_C[current_row + column] += sub_A[row * k + index] *
                sub_B[index * sub_m + column];
        }
    }
}
delete[] sub_A;
delete[] sub_B;

MPI_Datatype SUB_C_MINOR;
MPI_Type_contiguous(sub_n * sub_m, MPI_DOUBLE, &SUB_C_MINOR);
MPI_Type_commit(&SUB_C_MINOR);

if(dims[0] > dims[1]) {
    MPI_Datatype SUB_C_ROWS, SUB_C;
    MPI_Type_contiguous(sub_n * m, MPI_DOUBLE, &SUB_C_ROWS);
    MPI_Type_commit(&SUB_C_ROWS);
    MPI_Type_vector(sub_n, sub_m, m, MPI_DOUBLE, &SUB_C);
    MPI_Type_commit(&SUB_C);
    double *sub_C_rows = NULL;
    if(!coords[1]) {
        sub_C_rows = new double[sub_n * m];
        for(int row = 0; row < sub_n; row++) {
            for(int column = 0; column < sub_m; column++) {
                sub_C_rows[row * m + column] =
                    sub_C[row * sub_m + column];
            }
        }
        MPI_Status status;
        for(int index = 1; index < dims[1]; index++)
            MPI_Recv(sub_C_rows + sub_m * index, 1, SUB_C, index,
                MSG_ID, row_comm, &status);
    }
    else MPI_Send(sub_C, 1, SUB_C_MINOR, 0, MSG_ID, row_comm);

    if(!coords[1])
        MPI_Gather(sub_C_rows, 1, SUB_C_ROWS, C, 1, SUB_C_ROWS, 0,
            column_comm);

    MPI_Type_free(&SUB_C_ROWS);
    MPI_Type_free(&SUB_C);
    delete[] sub_C_rows;
}
else {
    MPI_Datatype SUB_C_COLUMNS, SUB_C_COLUMNS_CONTIGUOUS;
    MPI_Type_vector(n, sub_m, m, MPI_DOUBLE, &SUB_C_COLUMNS);
    MPI_Type_commit(&SUB_C_COLUMNS);
    MPI_Type_contiguous(n * sub_m, MPI_DOUBLE, &SUB_C_COLUMNS_CONTIGUOUS);
    MPI_Type_commit(&SUB_C_COLUMNS_CONTIGUOUS);
    double *sub_C_columns = NULL;
    if(!coords[0])
        sub_C_columns = new double[n * sub_m];

    MPI_Gather(sub_C, 1, SUB_C_MINOR, sub_C_columns, 1, SUB_C_MINOR, 0,

```

```

        column_comm);

    if(!coords[0]) {
        if(!coords[1]) {
            for(int row = 0; row < n; row++) {
                for(int column = 0; column < sub_m; column++)
                    C[row * m + column] =
                        sub_C_columns[row * sub_m + column];
            }
            MPI_Status status;
            for(int index = 1; index < dims[1]; index++)
                MPI_Recv(C + sub_m * index, 1, SUB_C_COLUMNS,
                    index, MSG_ID, row_comm, &status);
        }
        else MPI_Send(sub_C_columns, 1, SUB_C_COLUMNS_CONTIGUOUS, 0,
            MSG_ID, row_comm);
    }

    delete[] sub_C_columns;
    MPI_Type_free(&SUB_C_COLUMNS);
    MPI_Type_free(&SUB_C_COLUMNS_CONTIGUOUS);
}
MPI_Type_free(&SUB_C_MINOR);

if(!non_grid_rank) {
    print_matrix(C, n, m, std::cout);
    double end_time = MPI_Wtime();
    std::cerr << "It took " << end_time - start_time << " seconds\n";
}

delete[] A;
delete[] B;
delete[] C;
delete[] sub_C;

MPI_Finalize();
return 0;
}

```

Приложение 2. *Makefile*

Файл сборки проекта

```
CXX=g++
CXXP=mpicxx
CFLAGS=-O3 -Wall -c

SERIES_SOURCE=series
PARALLEL_SOURCE=parallel

all: parallel

parallel: $(PARALLEL_SOURCE).o
    $(CXXP) $(PARALLEL_SOURCE).o -o $(PARALLEL_SOURCE)

$(PARALLEL_SOURCE).o: $(PARALLEL_SOURCE).cpp
    $(CXXP) $(CFLAGS) $(PARALLEL_SOURCE).cpp -o $(PARALLEL_SOURCE).o

clean:
    rm -rf *.o
```

Приложение 3. Скрипты для кластера

- run.sh

```
#!/bin/bash

#PBS -l walltime=00:18:00
#PBS -l select=2:ncpus=12:mpiprocs=12:mem=2000m,place=scatter
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI processes: $MPI_NP"

N=3000
K=3600
M=4200

echo 'File $PBS_NODEFILE:'
cat $PBS_NODEFILE

for (( i = 0; i < 3; i++ )); do
    ROWS=2
    COLUMNS=12
    echo -n -e "$ROWS\t$COLUMNS\t" >> report.txt
    mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
        $N $K $M $ROWS $COLUMNS >/dev/null 2>>report.txt

    ROWS=3
    COLUMNS=8
    echo -n -e "$ROWS\t$COLUMNS\t" >> report.txt
    mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
        $N $K $M $ROWS $COLUMNS >/dev/null 2>>report.txt

    ROWS=4
    COLUMNS=6
    echo -n -e "$ROWS\t$COLUMNS\t" >> report.txt
    mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
        $N $K $M $ROWS $COLUMNS >/dev/null 2>>report.txt

    ROWS=6
    COLUMNS=4
    echo -n -e "$ROWS\t$COLUMNS\t" >> report.txt
    mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
        $N $K $M $ROWS $COLUMNS >/dev/null 2>>report.txt

    ROWS=8
    COLUMNS=3
    echo -n -e "$ROWS\t$COLUMNS\t" >> report.txt
    mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
        $N $K $M $ROWS $COLUMNS >/dev/null 2>>report.txt

    ROWS=12
    COLUMNS=2
    echo -n -e "$ROWS\t$COLUMNS\t" >> report.txt
    mpirun -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
        $N $K $M $ROWS $COLUMNS >/dev/null 2>>report.txt
done
```

- `trace.sh`

```
#!/bin/bash

#PBS -l walltime=00:02:00
#PBS -l select=1:ncpus=8:mpiprocs=8:mem=2000m,place=scatter
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI processes: $MPI_NP"

N=3000
K=3600
M=4200

echo 'File $PBS_NODEFILE:'
cat $PBS_NODEFILE
ROWS=4
COLUMNS=2
echo "Count of rows: $ROWS"
echo "Count of columns: $COLUMNS"

mpirun -trace -machinefile $PBS_NODEFILE -np $MPI_NP ./parallel \
      $N $K $M $ROWS $COLUMNS >/dev/null
```