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, 19202 группы

Бакшеева Ивана Дмитриевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
к. т. н., доцент
А. Ю. Власенко

Новосибирск 2021

СОДЕРЖАНИЕ

ЦЕЛЬ

Изучить топологии процессов MPI, изучить метод перемножения матриц на 2D решетках процессов.

ЗАДАНИЕ

1. Реализовать параллельный алгоритм умножения матрицы на матрицу при 2D решетке.
2. Исследовать производительность параллельной программы во зависимости от размера матрицы и размера решетки.
3. Выполнить профилирование программы с помощью MPE при использовании 16-ти ядер.
4. Сделать выводы, найти самую эффективную решетку.

ОПИСАНИЕ РАБОТЫ

1. Реализовал параллельный алгоритм умножения матрицы на матрицу (из задания). При создании решетки не использовал перенумерацию процессов. Корректность программы проверял, повторно считая произведение, но уже последовательным алгоритмом, а потом сравнивая результаты. В качестве начальных значений для матриц брал случайные целые числа. Для перемножения матриц использовал intel mkl cblas_dgemm.
2. Запустил на кластере программу с различным числом процессов, с различными по форме решетками.
3. Провел профилирование программы на 16 процессах с решеткой 4x4.
4. Сделал выводы по результатам измерений.

ЗАКЛЮЧЕНИЕ

Таблица по зависимости времени выполнения от числа процессов:

Число процессов	P1	P2	N1	N2	N3	T (с.)	Sp	Ep
1	1	1	6000	6000	6000	56,67	1,00	100%
2	1	2	6000	6000	6000	27,80	2,04	102%
4	2	2	6000	6000	6000	29,18	1,94	49%
8	2	4	6000	6000	6000	7,13	7,95	99%
12	2	6	6000	6000	6000	8,26	6,86	57%
16	4	4	6000	6000	6000	7,61	7,45	47%
24	4	6	6000	6000	6000	5,71	9,92	41%

Где P1, P2 – размеры решетки, N1-N3 – размеры матриц, T – время перемножения, Sp – ускорение, Ep – эффективность.

По результатам таблицы можно сделать вывод, что решетка 2 на 4 обладает большой эффективностью.

Таблица зависимости времени выполнения от формы решетки:

Число процессов	P1	P2	N1	N2	N3	T (с.)
24	1	24	6000	6000	6000	6,9
24	2	12	6000	6000	6000	5,83
24	3	8	6000	6000	6000	5,78
24	4	6	6000	6000	6000	5,68
24	6	4	6000	6000	6000	5,52
24	8	3	6000	6000	6000	5,64
24	12	2	6000	6000	6000	5,7
24	24	1	6000	6000	6000	6,3

Тут заметно, что матрица, которая ближе к квадратной, отрабатывает быстрее, чем вытянутая матрица процессов.

Таблица зависимости времени выполнения от размеров матриц:

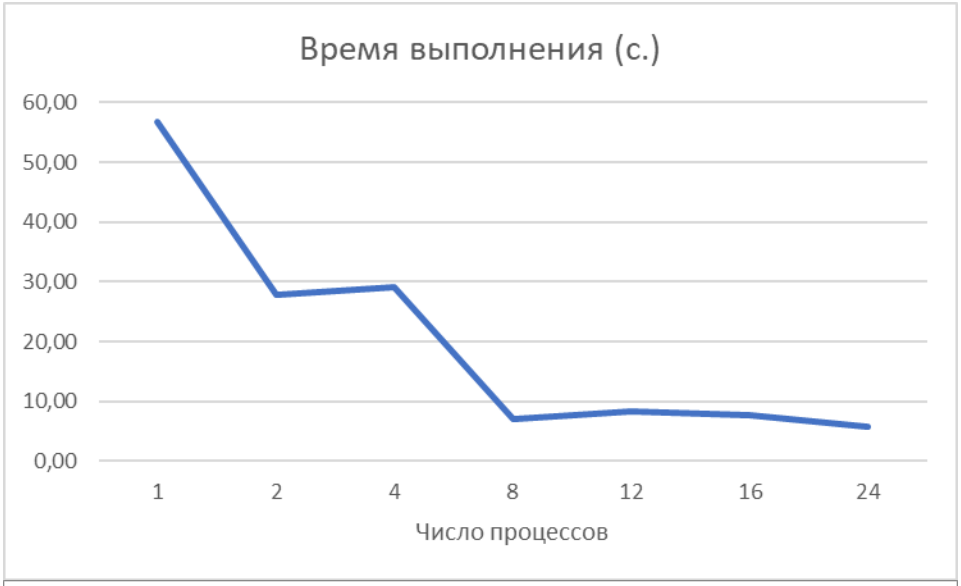
Число процессов	P1	P2	N1	N2	N3	T (с.)	S отн (OP/T)
16	4	4	1600	1600	640	0,12	13,65
16	4	4	3200	3200	1280	0,6	21,85
16	4	4	4800	4800	1920	1,8	24,58
16	4	4	6400	6400	2560	3,93	26,68
16	4	4	8000	8000	3200	7,28	28,13

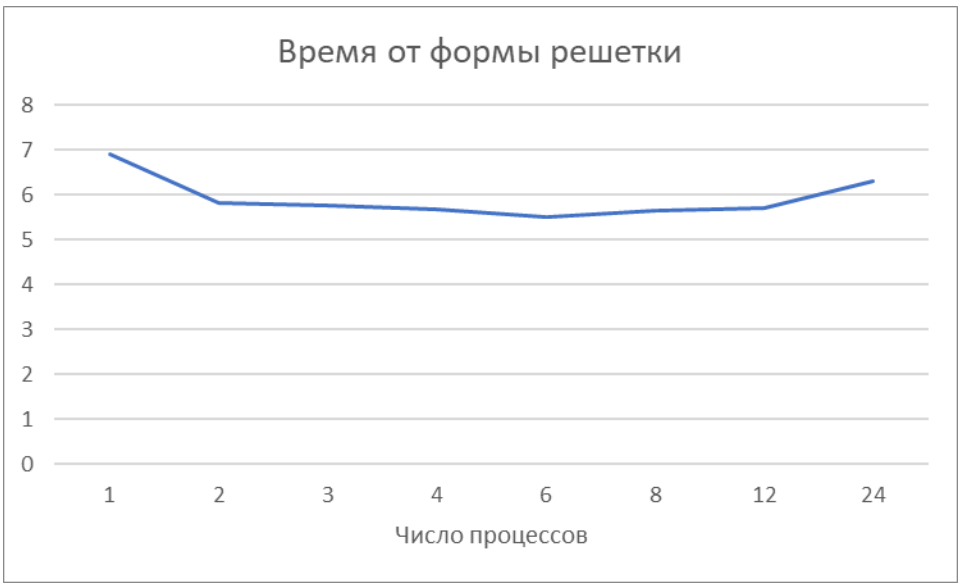
S отн – относительная эффективность, число операций / время.

С ростом размера матрицы время выполнения значительно увеличивается.

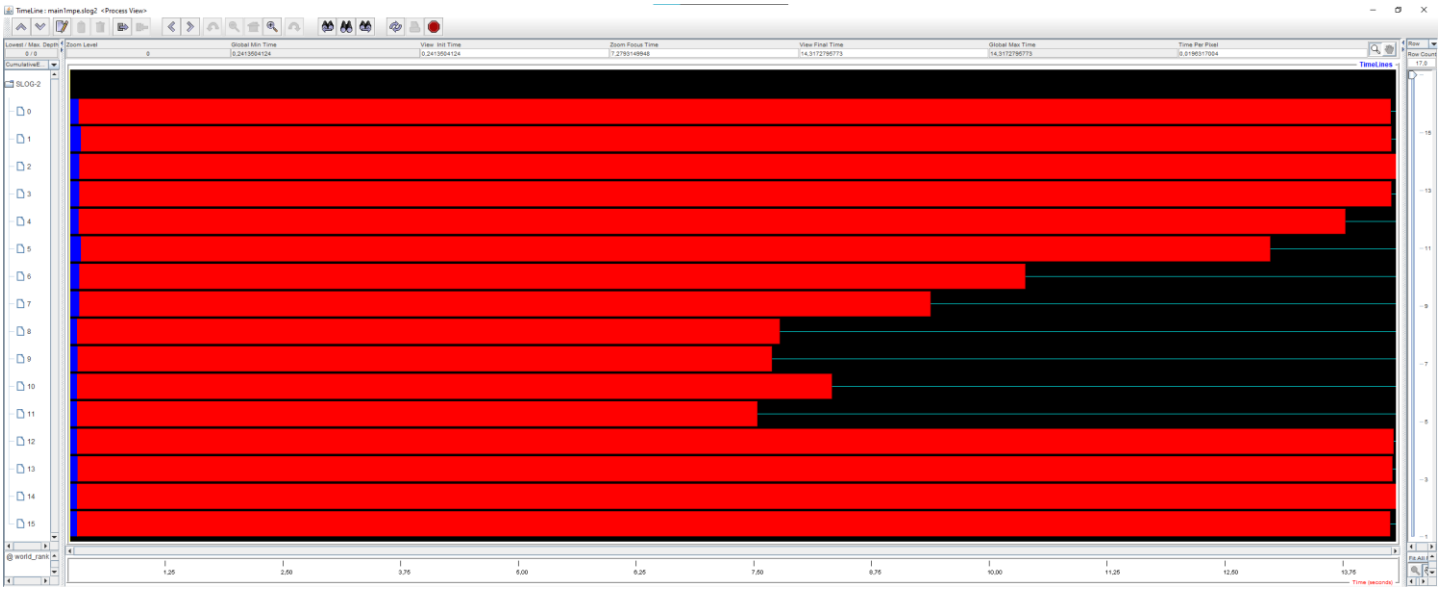
По результатам профилирования можно заметить, что наибольшее время занимает неэффективный наивный алгоритм умножения подматриц (красный цвет). Раздача данных по процессам занимает минимальное время (синий цвет).

Приложение 1. Графики по таблице




Время от формы решетки
8
7
6
5
4
3
2
1
0
1
2
3
4
6
8
12
24
Число процессов

Приложение 2. Профилирование (скриншот)



Приложение 3. Код программы

```
#include <iostream>
#include <vector>
#include <cassert>
#include <math.h>
#include <fstream>
#include <cstdlib>
#include <mpi.h>
// #include <cblas.h>
#include <mkl_cblas.h>

// #define CHECK

#define DEBUG(var) \
            do { std::cout << p_rank << " has " << #var << ": " << var <
< std::endl; } while (0)

void fill(double* x, int N, double value) {
    for (int i = 0; i < N; i++) {
        x[i] = value;
    }
}

enum Type {
    DIAG,
    RAND,
    REDIAG,
    INC
};

void fillmat(double* A, int m, int n, Type type, int value) {
    if (type == DIAG) {
        fill(A, m*n, 0.);
        for (int i = 0; i < std::min(n, m); i++) {
            A[i * n + i] = value;
        }
    }
    if (type == REDIAG) {
        fill(A, m*n, 0.);
        for (int i = 0; i < std::min(n, m); i++) {
            int x = n - i - 1;
            A[i * n + x] = value;
        }
    }
    if (type == INC) {
        fill(A, m*n, 0.);
        for (int i = 0; i < std::min(n, m); i++) {
            A[i * n + i] = i;
        }
```

```
    }
    if (type == RAND) {
        for (int i = 0; i < n*m; i++) {
            A[i] = rand() % 100;
        }
    }
}

void printmat(double* A, int m, int n, const char* name, int rank=0) {
    printf("%s (rank %d):\n", name, rank);
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            printf("%4.0f ", A[i*n + j]);
        }
        printf("\n");
    }
}

void mat_mat_mul(int m, int n, int k, double* AA, double* BB, double* CC
) {
    #define AA(i,j) AA[k*i + j]
    #define BB(i,j) BB[n*i + j]
    #define CC(i,j) CC[n*i + j]

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            CC(i, j) = 0.0;
            for (int g = 0; g < k; g++) {
                CC(i, j) += AA(i, g) * BB(g, j);
            }
        }
    }
}

void mpi_mat_mat_mul(int m, int n, int k, double* A, double* B, double*
C, MPI_Comm comm, int* p) {
    int size;
    MPI_Comm_size(comm, &size);

    int periods[2] = {0, 0};
    MPI_Comm comm2d;
    MPI_Dims_create(size, 2, p);
    MPI_Cart_create(comm, 2, p, periods, 0, &comm2d);

    int rank;
    MPI_Comm_rank(comm2d, &rank);

    int coords[2];
    MPI_Cart_get(comm2d, 2, p, periods, coords);

    MPI_Comm comm1d[2];
```

```
    int dims1[] = {0, 1};
    MPI_Cart_sub(comm2d, dims1, &comm1d[1]);
    int dims2[] = {1, 0};
    MPI_Cart_sub(comm2d, dims2, &comm1d[0]);

    int nn[2];

    // should be divisible without remainder
    nn[0] = m / p[0];
    nn[1] = n / p[1];

    double* AA = new double[nn[0] * k];
    double* BB = new double[nn[1] * k];
    double* CC = new double[nn[0] * nn[1]];

    if (coords[1] == 0) {
        MPI_Scatter(A, k * nn[0], MPI_DOUBLE, AA, k * nn[0], MPI_DOUBLE,
 0, comm1d[0]);
    }

    MPI_Datatype y_send_vector_t;
    MPI_Datatype y_padded_vector_t;
    MPI_Datatype y_recv_cont_t;

    int *dispc, *countc;
    if (coords[0] == 0) {
        MPI_Type_vector(k, nn[1], n, MPI_DOUBLE, &y_send_vector_t);
        MPI_Type_commit(&y_send_vector_t);

        MPI_Type_create_resized(y_send_vector_t, 0, nn[1] * sizeof(doubl
e), &y_padded_vector_t);
        MPI_Type_commit(&y_padded_vector_t);

        MPI_Type_contiguous(k * nn[1], MPI_DOUBLE, &y_recv_cont_t);
        MPI_Type_commit(&y_recv_cont_t);

        MPI_Scatter(B, 1, y_padded_vector_t, BB, 1, y_recv_cont_t, 0, co
mm1d[1]);

        dispc = new int[p[0] * p[1]];
        countc = new int[p[0] * p[1]];

        for (int i = 0; i < p[0]; i++) {
            for (int j = 0; j < p[1]; j++) {
                dispc[i*p[1] + j] = (i * p[1] * nn[0] + j);
                countc[i*p[1] + j] = 1;
            }
        }

        MPI_Type_free(&y_send_vector_t);
```

```
        MPI_Type_free(&y_padded_vector_t);
        MPI_Type_free(&y_recv_cont_t);
    }


    MPI_Bcast(AA, nn[0] * k, MPI_DOUBLE, 0, comm1d[1]);
    MPI_Bcast(BB, k * nn[1], MPI_DOUBLE, 0, comm1d[0]);

    // mat_mat_mul(nn[0], nn[1], k, AA, BB, CC);
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, nn[0], nn[1],
 k, 1, AA, k, BB, nn[1], 1, CC, nn[1]);

    MPI_Datatype c_recv_vector_t, c_send_vector_t, c_padded_recv_vector_
t;

    MPI_Type_contiguous(nn[0] * nn[1], MPI_DOUBLE, &c_send_vector_t);
    MPI_Type_commit(&c_send_vector_t);

    MPI_Type_vector(nn[0], nn[1], n, MPI_DOUBLE, &c_recv_vector_t);
    MPI_Type_commit(&c_recv_vector_t);

    MPI_Type_create_resized(c_recv_vector_t, 0, nn[1] * sizeof(double),
&c_padded_recv_vector_t);
    MPI_Type_commit(&c_padded_recv_vector_t);

    MPI_Gatherv(CC, 1, c_send_vector_t, C, countc, dispc, c_padded_recv_
vector_t, 0, comm2d);

    MPI_Type_free(&c_recv_vector_t);
    MPI_Type_free(&c_padded_recv_vector_t);
    MPI_Type_free(&c_send_vector_t);
    MPI_Comm_free(&comm1d[0]);
    MPI_Comm_free(&comm1d[1]);
    MPI_Comm_free(&comm2d);
    delete[] AA;
    delete[] BB;
    delete[] CC;
    if (rank == 0) {
        delete[] dispc;
        delete[] countc;
    }
}

int main(int argc, char** argv) {
    int p_rank;
    int p_count;

    // INIT MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &p_rank);
```

```
    const int N1 = atoi(argv[1]);
    const int N2 = atoi(argv[2]);
    const int N3 = atoi(argv[3]);
    const int P1 = atoi(argv[4]);
    const int P2 = atoi(argv[5]);


    double* matrix_A;
    double* matrix_B;
    double* matrix_C;
    double* matrix_C1;

    double start, end;

    assert(P1 * P2 == p_count);

    // LOAD MATRIX
    if (p_rank == 0) {
        matrix_A = new double[N1*N2];
        fillmat(matrix_A, N1, N2, RAND, 0.);
        matrix_B = new double[N2*N3];
        fillmat(matrix_B, N2, N3, RAND, 0.);
        matrix_C = new double[N1*N3];
#ifdef CHECK
        matrix_C1 = new double[N1*N3];
        // cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, nn[0],
 nn[1], k, 1, AA, k, BB, nn[1], 1, CC, nn[1]);
        mat_mat_mul(N1, N3, N2, matrix_A, matrix_B, matrix_C1);
#endif
    }

    int p[2] = {P1, P2};
    if (p_rank == 0) {
        start = MPI_Wtime();
    }

    mpi_mat_mat_mul(N1, N3, N2, matrix_A, matrix_B, matrix_C, MPI_COMM_W
ORLD, p);
    if (p_rank == 0) {
        end = MPI_Wtime();
        printf("%d\t%d\t%d\t%d\t%d\t%d\t", P1*P2, P1, P2, N1, N2, N3);
        printf("%.2f\n", end - start);
    }

    if (p_rank == 0) {
        // printmat(matrix_C, N1, N3, "C");
        // printmat(matrix_C1, N1, N3, "C1");
        #ifdef CHECK
        for (int i = 0; i < N1*N3; i++) {
            if (fabs(matrix_C[i] - matrix_C1[i]) > 0.0001) {
```

```
                printf("C1 != C2, program is not correct...\n");
                break;
            }
        }
        printf("checked\n");
        delete[] matrix_C1;
        #endif

        delete[] matrix_A;
        delete[] matrix_B;
        delete[] matrix_C;
    }

    MPI_Finalize();

    return 0;
}
```