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

«название работы»

студента 2 курса, 18209 группы

Большим Максима Антонович

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Матвеев А.С.

Новосибирск 2020

СОДЕРЖАНИЕ

ЦЕЛЬ

Реализовать умножение матриц с помощью MPI.

ЗАДАНИЕ

Реализовать параллельное вычисление подматриц для вычисления произведения матриц A и B, чьё произведение является возможным. Отправку матриц реализовать с помощью MPI_Scatter, MPI_Bcast, сборку реализовать с помощью MPI_Gather Провести тестирование полученной программы на кластере НГУ. Проанализировать полученные результаты.

КОД ПРОГРАММЫ

```
#include <cstdio>

#include <cstdlib>

#include <cerrno>

#include <mpi.h>

#include <cmath>


#define DIMENSIONS 2

#define X 0

#define Y 1


void init_commutators(MPI_Comm *grid_comm, MPI_Comm *rows_comm, MPI_Comm *col_comm, int *coords, int *dims);

void distribute_matrices(const double *matrixA, const double *matrixB, double *partA, double *partB,

      MPI_Comm row_comm, MPI_Comm col_comm, const int coords[DIMENSIONS], const int dims[DIMENSIONS], int n1, int n2, int n3);

void build_result_matrix(const double *partC, double *C, MPI_Comm grid_comm, const int dims[DIMENSIONS], int coords[DIMENSIONS], int n1, int n3, int comm_size);

void fill_matrix(double *matrix, int row, int col, double value);

void matrix_mul(const double *A, const double *B, double *C, int rowsA, int colA, int colB);

void init_matrices(double **A, double **B, double **C, double **C_test, int n1, int n2, int n3, char **argv, int argc);

void print_matrix(const double *matrix, int n1, int n2);

void free_all(double *A, double *B, double *C, double *C_test, double *partA, double *partB, double *partC, MPI_Comm *grid_comm, MPI_Comm *col_comm, MPI_Comm *row_comm, int rank);

double compare_matrices(double *A, double *B, int n1, int n3);
```

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int dims[DIMENSIONS] = {0, 0};
    int coords[DIMENSIONS];
    int size;
    int rank;
    MPI_Comm grid_comm;
    MPI_Comm row_comm;
    MPI_Comm col_comm;

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Dims_create(size, DIMENSIONS, dims);
    init_commutators(&grid_comm, &row_comm, &col_comm, coords, dims);
    MPI_Comm_rank(grid_comm, &rank);

    int n1 = (int)strtol(argv[1], NULL, 10);
    int n2 = (int)strtol(argv[2], NULL, 10);
    int n3 = (int)strtol(argv[3], NULL, 10);

    if(!rank && errno == ERANGE)
    {
        fprintf(stderr, "Invalid arguments!\n");
        return EXIT_FAILURE;
    }

#ifdef DEBUG
        printf("My rank = %d and my coords is %d %d\n", rank, coords[X], coords[Y]);
#endif

    double *A = NULL;
    double *B = NULL;
    double *C = NULL;
```

```
    double *C_test = NULL;

    double *partA = (double*)malloc(sizeof(double) * n1 * n2 / dims[1]);
    double *partB = (double*)malloc(sizeof(double) * n2 * n3 / dims[0]);
    double *partC = (double*)malloc(sizeof(double) * n1 * n3 / (dims[X] * dims[Y]));

    if (!rank)
        init_matrices(&A, &B, &C, &C_test, n1, n2, n3, argv, argc);

    distribute_matrices(A, B, partA, partB, row_comm, col_comm, coords, dims, n1, n2, n3);
    matrix_mul(partA, partB, partC, n1 / dims[Y], n2, n3 / dims[X]);
    build_result_matrix(partC, C, grid_comm, dims, coords, n1, n3, size);

    if (!rank && argc != 7)
        print_matrix(C, n1, n3);
    else if (!rank)
        printf("Error compare: %lf\n", compare_matrices(C_test, C, n1, n3));

    free_all(A, B, C, C_test, partA, partB, partC, &grid_comm, &col_comm, &row_comm, rank);

    MPI_Finalize();

    return EXIT_SUCCESS;
}

void free_all(double *A, double *B, double *C, double *C_test, double *partA, double *partB, double *partC, MPI_Comm
*grid_comm, MPI_Comm *col_comm, MPI_Comm *row_comm, int rank)
{
    if (!rank)
    {
        free(A);
        free(B);
        free(C);
        if(C_test)
            free(C_test);
```

```
    }
    free(partA);
    free(partB);
    free(partC);

    MPI_Comm_free(grid_comm);
    MPI_Comm_free(col_comm);
    MPI_Comm_free(row_comm);
}

void build_result_matrix(const double *partC, double *C, MPI_Comm grid_comm, const int dims[DIMENSIONS], int
coords[DIMENSIONS], int n1, int n3, int comm_size)
{
    int *recvCounts = (int*)malloc(sizeof(int) * comm_size);
    int *displs = (int*)malloc(sizeof(int) * comm_size);

#ifdef DEBUG
    if (!rank)
    {
        printf("dims %d %d \n", dims[0], dims[1]);
    }
#endif

    MPI_Datatype recv_vector_t, resized_recv_vector_t, send_vector_t;
    PMPI_Type_contiguous(n1 * n3 / (dims[X] * dims[Y]), MPI_DOUBLE, &send_vector_t);
    MPI_Type_commit(&send_vector_t);

    MPI_Type_vector(n1 / dims[Y], n3 / dims[X], n3, MPI_DOUBLE, &recv_vector_t);
    MPI_Type_commit(&recv_vector_t);

    MPI_Type_create_resized(recv_vector_t, 0, n3 / dims[X] * sizeof(MPI_DOUBLE), &resized_recv_vector_t);
    MPI_Type_commit(&resized_recv_vector_t);

    int typeSizeBytes;
    MPI_Type_size(resized_recv_vector_t, &typeSizeBytes);
```

```
#ifdef DEBUG
    printf("%lu\n", typeSizeBytes / sizeof(MPI_DOUBLE) / (n3 / (dims[X])));
#endif

    int typeSize = typeSizeBytes / ( (int)sizeof(MPI_DOUBLE) * (n3 / (dims[X])));

    for (int rank_i = 0; rank_i < comm_size; ++rank_i)
    {
        recvCounts[rank_i] = 1;

        MPI_Cart_coords(grid_comm, rank_i, DIMENSIONS, coords);
        displs[rank_i] = coords[Y] * dims[X] * (int)typeSize + coords[X] * 2;

#ifdef DEBUG
        if (!rank)
            printf("id %d coords: %d %d, result %d\n", rank_i, coords[X], coords[Y], displs[rank_i]);
#endif
    }

#ifdef DEBUG
    for (int l = 0; l < n1 * n3 / (dims[X] * dims[Y]); ++l)
    {
        partC[l] = rank;
    }

    if (!rank)
    {
        for (int m = 0; m < n1 * n3; ++m)
        {
            C[m] = -1;
        }
    }
#endif
```

```
    MPI_Gatherv(partC, 1, send_vector_t, C, recvCounts, displs, resized_recv_vector_t, 0, grid_comm);

    MPI_Type_free(&recv_vector_t);
    MPI_Type_free(&resized_recv_vector_t);
    MPI_Type_free(&send_vector_t);
}

void init_matrices(double **A, double **B, double **C, double **C_test, int n1, int n2, int n3, char **argv, int argc)
{
    *A = (double*)malloc(sizeof(double) * n1 * n2);
    *B = (double*)malloc(sizeof(double) * n2 * n3);
    *C = (double*)malloc(sizeof(double) * n1 * n3);

    if(argc == 7)
    {
        *C_test = (double*)malloc(sizeof(double) * n1 * n3);

        FILE* A_f = fopen(argv[4], "rb");
        FILE* B_f = fopen(argv[5], "rb");
        FILE* C_f = fopen(argv[6], "rb");

        float a;
        for (int i = 0; i < n1 * n2; ++i)
        {
            fread(((void *) &a), sizeof(float), 1, A_f);
            (*A)[i] = a;
        }
        for (int i = 0; i < n2 * n3; ++i)
        {
            fread(((void *) &a), sizeof(float), 1, B_f);
            (*B)[i] = a;
        }
        for (int i = 0; i < n1 * n3; ++i)
```

```
        {
            fread(((void *) &a), sizeof(float), 1, C_f);
            (*C_test)[i] = a;
        }
        fclose(A_f);
        fclose(B_f);
        fclose(C_f);
    } else
    {
        fill_matrix(*A, n1, n2, 2);
        fill_matrix(*B, n2, n3, 2);

        printf("matrix A:\n");
        print_matrix(*A, n1, n2);

        printf("matrix B:\n");
        print_matrix(*B, n2, n3);

        printf("----------RESULT----------\n");
    }
}

void init_commutators(MPI_Comm *grid_comm, MPI_Comm *rows_comm, MPI_Comm *col_comm, int *coords, int *dims)
{
    int periods[DIMENSIONS] = {0, 0};
    MPI_Cart_create(MPI_COMM_WORLD, DIMENSIONS, dims, periods, 0, grid_comm);
    int rank;
    MPI_Comm_rank(*grid_comm, &rank);
    MPI_Cart_coords(*grid_comm, rank, DIMENSIONS, coords);

    MPI_Comm_split(*grid_comm, coords[Y], coords[X], rows_comm);
    MPI_Comm_split(*grid_comm, coords[X], coords[Y], col_comm);
}
```

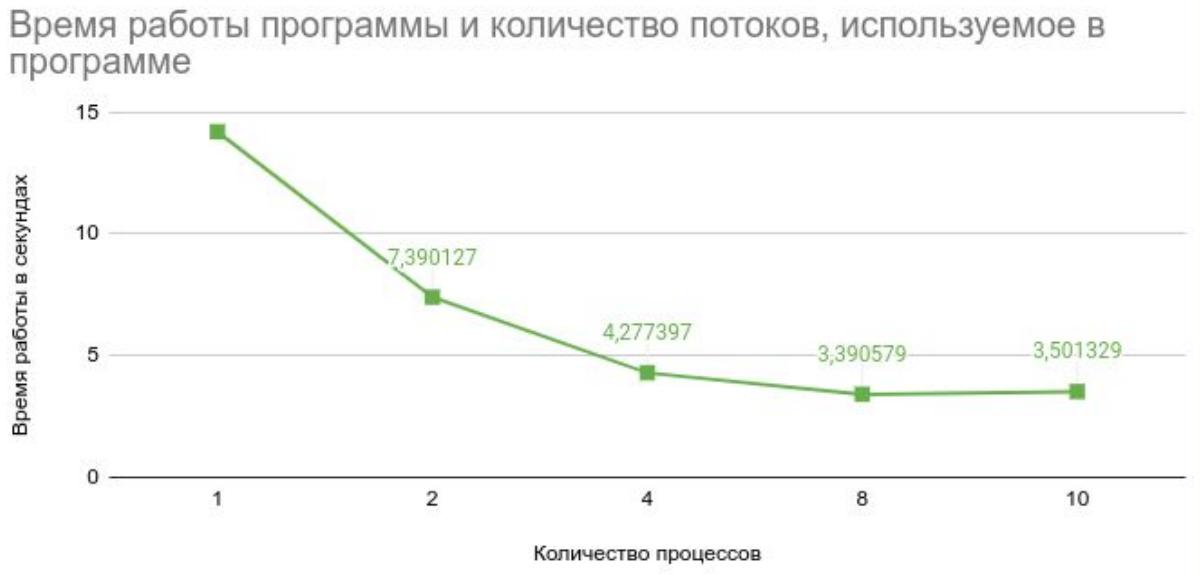
```
void distribute_matrices(const double *matrixA, const double *matrixB, double *partA, double *partB, const MPI_Comm row_comm,
                         const MPI_Comm col_comm, const int coords[DIMENSIONS], const int dims[DIMENSIONS], int n1, int n2, int n3)
{
   if (coords[X] == 0)
      MPI_Scatter(matrixA, n1 * n2 / dims[Y], MPI_DOUBLE, partA, n1 * n2 / dims[Y], MPI_DOUBLE, 0,
                  col_comm);

   if (coords[Y] == 0)
   {
      MPI_Datatype vector_t;
      MPI_Datatype resized_vector_t;
      MPI_Datatype recv_t;

      MPI_Type_vector(n2, n3 / dims[X], n3, MPI_DOUBLE, &vector_t);
      MPI_Type_commit(&vector_t);

      MPI_Type_create_resized(vector_t, 0, n3 / dims[X] * sizeof(double), &resized_vector_t);
      MPI_Type_commit(&resized_vector_t);

      PMPI_Type_contiguous(n2 * n3 / dims[X], MPI_DOUBLE, &recv_t);
      MPI_Type_commit(&recv_t);

      MPI_Scatter(matrixB, 1, resized_vector_t, partB, 1, recv_t, 0, row_comm);

      MPI_Type_free(&resized_vector_t);
      MPI_Type_free(&vector_t);
      MPI_Type_free(&recv_t);
   }

   MPI_Bcast(partA, n1 * n2 / dims[Y], MPI_DOUBLE, 0, row_comm);
   MPI_Bcast(partB, n2 * n3 / dims[X], MPI_DOUBLE, 0, col_comm);
}
```

```
void fill_matrix(double *matrix, int row, int col, double value)
{
   for (int i = 0; i < row; ++i)
      for (int j = 0; j < col; ++j)
         if (i == j)
            matrix[i * col + j] = 4;
         else
            matrix[i * col + j] = value;
}

void matrix_mul(const double *A, const double *B, double *C, int rowsA, int colA, int colB)
{
   for (int i = 0; i < rowsA; ++i)
   {
      double *c = C + i * colB;
      for (int j = 0; j < colB; ++j)
         c[j] = 0;
      for (int k = 0; k < colA; ++k)
      {
         const double *b = B + k * colB;
         double a = A[i * colA + k];
         for (int j = 0; j < colB; ++j)
            c[j] += a * b[j];
      }
   }
}

void print_matrix(const double *matrix, int n1, int n2)
{
   for (int i = 0; i < n1; ++i)
   {
      for (int j = 0; j < n2; ++j)
         printf("%lf ", matrix[i * n2 + j]);
```

```
        printf("\n");
    }
}

double compare_matrices(double *A, double *B, int n1, int n3)
{
    double error = 0;
    for(int i = 0; i < n1;i++)
        for(int j = 0; j < n3;j++)
            error += fabs(A[i * n3 + j] - B[i * n3 + j]) * fabs(A[i * n3 + j] - B[i * n3 + j]);
    return error;
}
```

ГРАФИК ВРЕМЕНИ РАБОТЫ

Проведём тесты умножения матриц 2400x2400 элементов.



Такая высокая производительность обусловлена тем, что умножение матриц реализовано эффективно, т.е. расчёт указателей в матрицах для умножения происходят более быстро, нежели в наивной реализации.

Вывод

В ходе лабораторной работы я научился умножать матрицы параллельно в разных процессах с помощью MPI. Основной аспект данной лабы заключался в том, что нужно было грамотно реализовать рассылку подматриц с

помощью производных типов, что и было сделано. Программа выдает хорошую скорость и точный результат.