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 матриц в MPI при помощи 2D решетки»

Студента 2 курса, группы 21209

Панас Матвея Алексеевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Мичуров Михаил Антонович

ЦЕЛЬ

Ознакомиться с написанием параллельных MPI-программ с использованием построения 2D решетки;

ЗАДАНИЕ

1. *Реализовать параллельный алгоритм умножения матрицы при помощи 2D решетки;*
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.*

ВАРИАНТ ЗАДАНИЯ

Вычисляется произведение $C = A \times B$ *, где A – матрица размера* $n_1 \times n_2$ *, B – матрица* $n_2 \times n_3$ *и C – матрица результатов* $n_1 \times n_3$ *.*

Исходные матрицы первоначально доступны на нулевом процессе, и матрица результатов возвращена в нулевой процесс.

Параллельное выполнение алгоритма осуществляется на двумерной (2D) решетке компьютеров размером $p_1 \times p_2$ *. Матрица A разрезана на* p_1 *горизонтальных полос, матрица B разрезана на* p_2 *вертикальных полос, и матрица результата C разрезана на* $p_1 \times p_2$ *подматрицы.*

Каждый компьютер (i, j) *вычисляет произведение* i*-й горизонтальной полосы матрицы A и* j*-й вертикальной полосы матрицы B, произведение получено в подматрице* (i, j) *матрицы C.*

Последовательные стадии вычисления

1. Матрица A распределяется по горизонтальным полосам вдоль координаты $(x, 0)$ *.*

2. Матрица B распределяется по вертикальным полосам вдоль координаты $(0, y)$ *.*

3. Полосы A распространяются в измерении y*.*

4. Полосы B распространяются в измерении x*.*

5. Каждый процесс вычисляет одну подматрицу произведения.

6. Матрица C собирается из (x, y) *плоскости.*

Исходные данные

Матрица A содержит строки, состоящие из чисел, соответствующих номеру строки.

Матрица B содержит столбцы, состоящие из чисел, соответствующих номеру столбца.

Матрица C содержит на (i, j) *позиции число* $i \times j \times n_2$

ОПИСАНИЕ РАБОТЫ

Описание выполненной работы

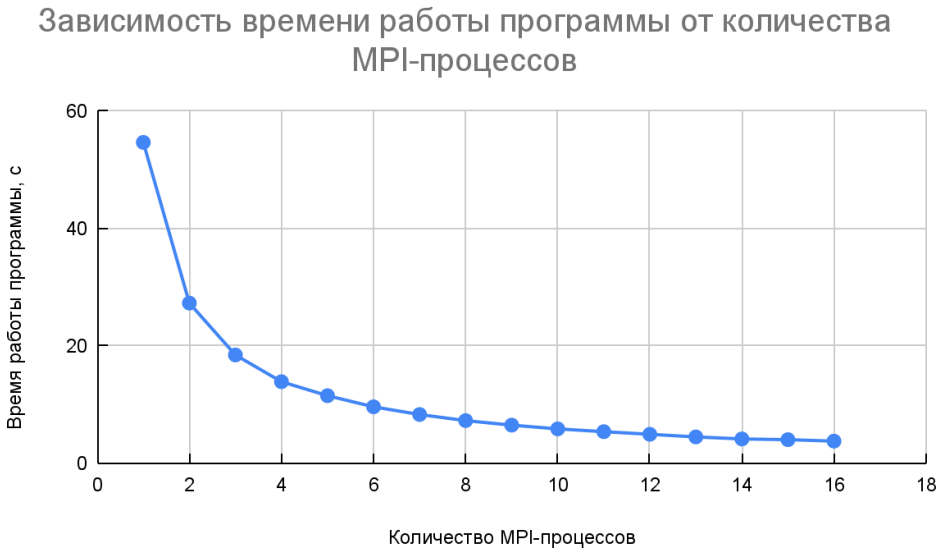
1. *Написал параллельную программу, реализующую умножение матрица, использую построение 2D решетки;*

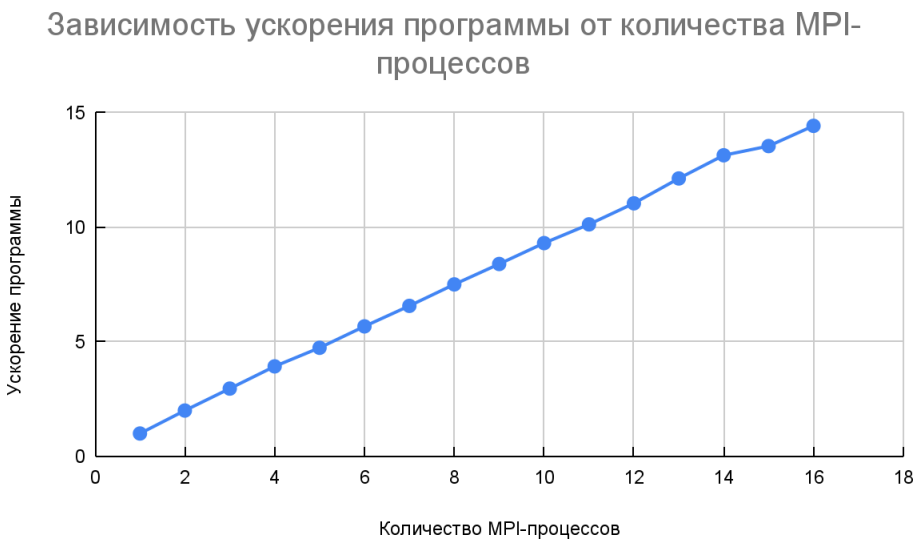
2. *Запустил программу при использовании от 1 до 16 ядер со следующими размерами матриц:* $n_1 = 2000$, $n_2 = 1500$, $n_3 = 2500$ *;*

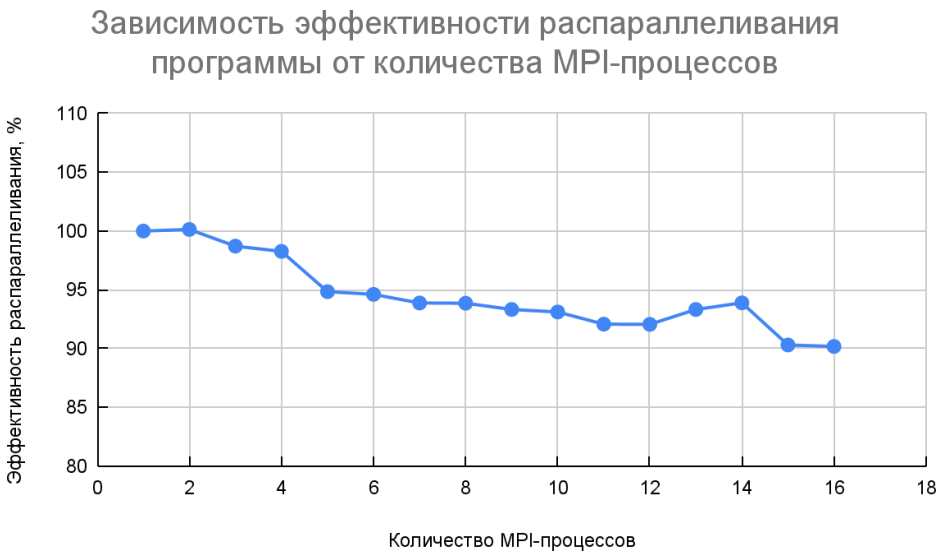
3. *Запустил программу при использовании 16 ядер при различных размерах сетки со следующими размерами матриц:* $n_1 = 4000$, $n_2 = 3000$, $n_3 = 5000$ *;*

4. *Выполнил профилирование программы с помощью MPE при использовании 16 ядер со следующими размерами матриц:* $n_1 = 4000$, $n_2 = 3000$, $n_3 = 5000$ *;*

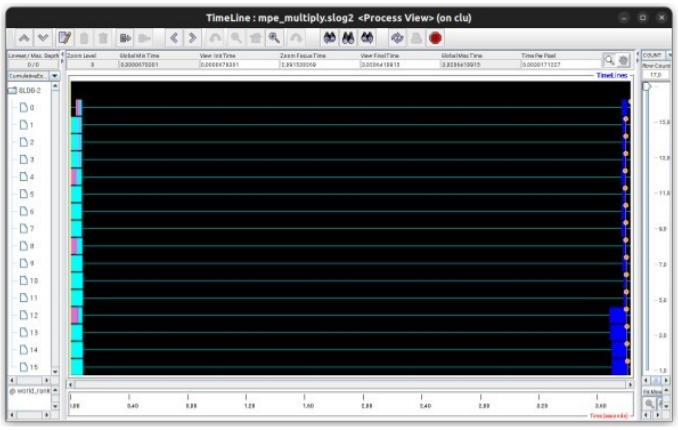
Результаты измерения





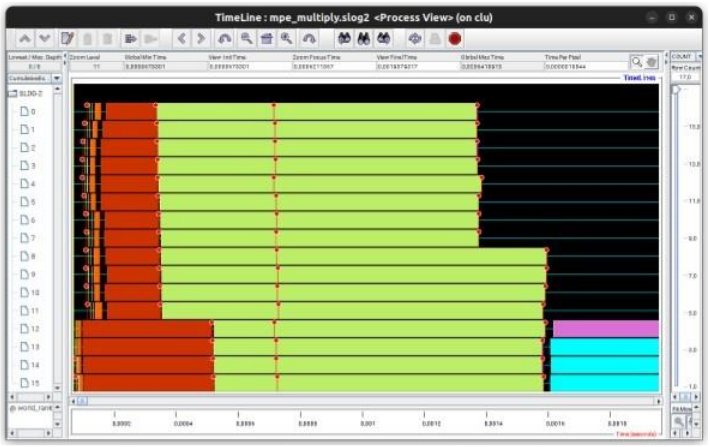

Зависимость эффективности распараллеливания программы от количества MPI-процессов
Эффективность распараллеливания, %
110
105
100
95
90
85
80
0
2
4
6
8
10
12
14
16
18
Количество MPI-процессов


Зависимость времени работы программы от размеров двумерной сетки для 16 MPI-процессов
Время работы программы, с
29,5
29,3
29,1
28,9
28,7
28,5
(1, 16)
(2, 8)
(4, 4)
(8, 2)
(16, 1)
Количество MPI-процессов на каждое измерение

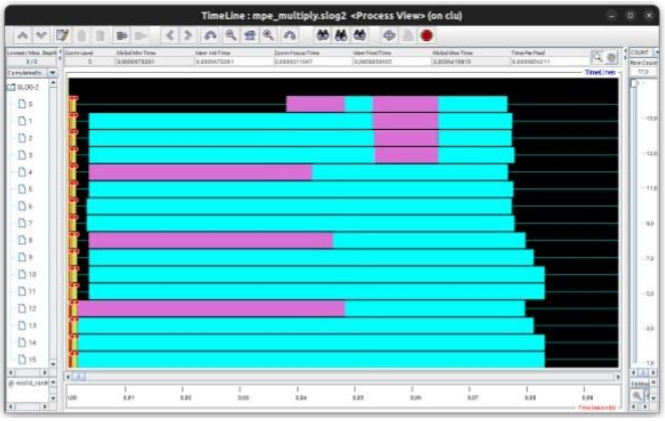


Полная временная шкала

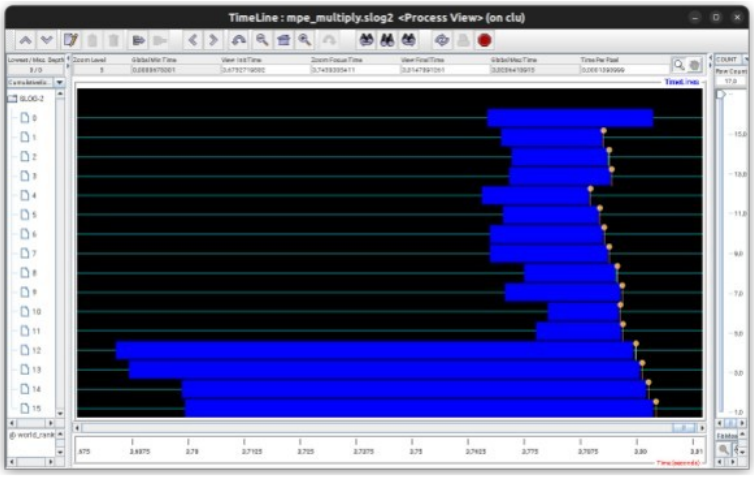
Легенда временной шкалы



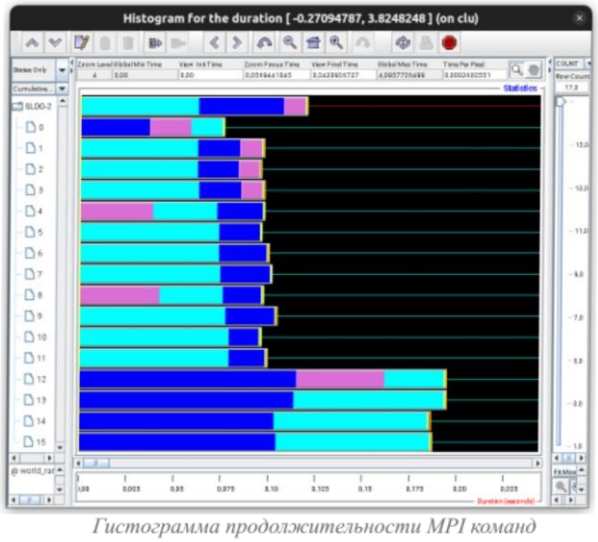
Участок временной шкалы от 0 до 0,0015 секунд



Участок временной шкалы с 0 до 0,09 секунд



Участок временной шкалы от 3,66 до 3,83 секунд



Гистограмма продолжительности MPI команд

Заключение

В ходе выполнения работы я реализовал параллельный алгоритм умножения матрицы с помощью 2D-решетки. Исследовал производительность параллельный программы в зависимости от количества узлов при автоматическом выборе размера решетки, а также при фиксированном количестве узлов. Затем выполнил профилирование программы с помощью MPE при использовании 16 узлов. Анализируя результаты проделанной работы, можно сказать:

Большая часть времени работы программы ушла на умножение блоков матриц.

Эффективность распараллеливания не опускалась ниже 90%, следовательно программа может хорошо масштабироваться.

Время работы программы минимально на 16 узлах при решетке 4x4. Это обуславливается тем, что количество вызовов функции “MPI_Scatter” минимально.

Нулевой процесс заставлял остальные процессы простаивать, так как он принадлежал первому столбцу и строке и в нем дважды вызывался MPI_Scatter, А остальные процессы тем временем вызывали MPI_Scatter или MPI_Bcast.

```c
#include <math.h>
#include <mpi.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define DIMS_COUNT 2
#define X 0
#define Y 1

int check_print_error(double *a, const char *message) {
  if (!a) {
    fprintf(stream: stderr, format: "%s", message);
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
}

double *allocate_matrix(size_t size) {
  double *matrix = malloc(size: size * sizeof(double));
  check_print_error(matrix, message: "FAILED TO ALLOCATE MEMORY TO MATRIX\n");
  return matrix;
}

void generate_matrix(double *matrix, int column, int leading_row, int leading_column, bool onRows) {
  for (int i = 0; i < leading_row; ++i) {
    for (int j = 0; j < leading_column; ++j) {
      matrix[i * column + j] = onRows ? i : j;
    }
  }
}

void split_B(const double *B, double *B_block, int B_block_size, int n_2, int aligned_n3, int coords_x, MPI_Comm comm_rows, MPI_Comm comm_columns) {
  if (coords_x == 0) {
    MPI_Datatype column_not_resized_t;
    MPI_Datatype column_resized_t;

    MPI_Type_vector(count: n_2, blocklength: B_block_size, stride: aligned_n3, oldtype: MPI_DOUBLE, newtype: &column_not_resized_t);
    MPI_Type_commit(datatype: &column_not_resized_t);

    MPI_Type_create_resized(oldtype: column_not_resized_t, lb: 0, extent: B_block_size * sizeof(double), newtype: &column_resized_t);
    MPI_Type_commit(datatype: &column_resized_t);

    MPI_Scatter(sendbuf: B, sendcount: 1, sendtype: column_resized_t, recvbuf: B_block, recvcount: B_block_size * n_2, recvtype: MPI_DOUBLE, root: 0, comm_rows);

    MPI_Type_free(datatype: &column_not_resized_t);
    MPI_Type_free(datatype: &column_resized_t);
  }

  MPI_Bcast(buffer: B_block, count: B_block_size * n_2, datatype: MPI_DOUBLE, root: 0, comm_columns);
}

void split_A(const double *A, double *A_block, int A_block_size, int n_2, int coords_y, MPI_Comm comm_rows, MPI_Comm comm_columns) {
  if (coords_y == 0) {
    MPI_Scatter(sendbuf: A, sendcount: A_block_size * n_2, sendtype: MPI_DOUBLE, recvbuf: A_block, recvcount: A_block_size * n_2, recvtype: MPI_DOUBLE, root: 0, comm_columns);
  }

  MPI_Bcast(buffer: A_block, count: A_block_size * n_2, datatype: MPI_DOUBLE, root: 0, comm_rows);
}

void init_communicators(const int dims[DIMS_COUNT], MPI_Comm *comm_grid, MPI_Comm *comm_rows, MPI_Comm *comm_columns) {
  int reorder = 1;
  int periods[DIMS_COUNT] = {};
  int sub_dims[DIMS_COUNT] = {};

  MPI_Cart_create(comm_old: MPI_COMM_WORLD, ndims: DIMS_COUNT, dims, periods, reorder, comm_cart: comm_grid);

```

```
  sub_dims[X] = false;
  sub_dims[Y] = true;
  MPI_Cart_sub(*comm_grid, remain_dims: sub_dims, newcomm: comm_rows);

  sub_dims[X] = true;
  sub_dims[Y] = false;
  MPI_Cart_sub(*comm_grid, remain_dims: sub_dims, newcomm: comm_columns);
}

void multiply(const double *A_block, const double *B_block, double *C_block, int A_block_size, int B_block_size, int n_2) {
  for (int i = 0; i < A_block_size; ++i) {
    for (int j = 0; j < B_block_size; ++j) {
      C_block[i * B_block_size + j] = 0;
    }
  }

  for (int i = 0; i < A_block_size; ++i) {
    for (int j = 0; j < n_2; ++j) {
      for (int k = 0; k < B_block_size; ++k) {
        C_block[i * B_block_size + k] += A_block[i * n_2 + j] * B_block[j * B_block_size + k];
      }
    }
  }
}

void compose(const double *C_block, double *C, int A_block_size, int B_block_size, int aligned_n1, int aligned_n3, int proc_count, MPI_Comm comm_grid) {
  MPI_Datatype not_resized_recv_t;
  MPI_Datatype resized_recv_t;

  int dims_x = aligned_n1 / A_block_size;
  int dims_y = aligned_n3 / B_block_size;
  int *recv_counts = malloc(size: sizeof(int) * proc_count);
  int *displs = malloc(size: sizeof(int) * proc_count);

  MPI_Type_vector(count: A_block_size, blocklength: B_block_size, stride: aligned_n3, oldtype: MPI_DOUBLE, newtype: &not_resized_recv_t);
  MPI_Type_commit(datatype: &not_resized_recv_t);

  MPI_Type_create_resized(oldtype: not_resized_recv_t, lb: 0, extent: B_block_size * sizeof(double), newtype: &resized_recv_t);
  MPI_Type_commit(datatype: &resized_recv_t);

  for (int i = 0; i < dims_x; ++i)
    for (int j = 0; j < dims_y; ++j) {
      recv_counts[i * dims_y + j] = 1;
      displs[i * dims_y + j] = j + i * dims_y * A_block_size;
    }

  MPI_Gatherv(sendbuf: C_block, sendcount: A_block_size * B_block_size, sendtype: MPI_DOUBLE, recvbuf: C, recvcounts: recv_counts, displs, recvtype: resized_recv_t, root: 0, comm_grid);

  MPI_Type_free(datatype: &not_resized_recv_t);
  MPI_Type_free(datatype: &resized_recv_t);
  free(ptr: recv_counts);
  free(ptr: displs);
}

bool check_C(const double *C, int column, int leading_row, int leading_column, int n_2) {
  for (int i = 0; i < leading_row; ++i)
    for (int j = 0; j < leading_column; ++j) {
      if (C[i * column + j] != (double)(i * j * n_2)) {
        return false;
      }
    }

  return true;
}

void print_matrix(const double *matrix, int column, int leading_row, int leading_column) {
  for (int i = 0; i < leading_row; i++) {
```

```
    for (int j = 0; j < leading_column; j++) {
      printf(format: "%lf ", matrix[i * column + j]);
    }

    printf(format: "\n");
  }
}

void init_dims(int dims[DIMS_COUNT], int proc_count) {
  MPI_Dims_create(nnodes: proc_count, ndims: DIMS_COUNT, dims);
}

int main(int argc, char **argv) {
  int n_1 = 4000;
  int n_2 = 3000;
  int n_3 = 5000;
  int proc_rank;
  int proc_count;
  int aligned_n1;
  int aligned_n3;
  int A_block_size;
  int B_block_size;
  int dims[DIMS_COUNT] = {};
  int coords[DIMS_COUNT] = {};
  double start_time;
  double finish_time;
  double *A = NULL;
  double *B = NULL;
  double *C = NULL;
  double *A_block = NULL;
  double *B_block = NULL;
  double *C_block = NULL;
  MPI_Comm comm_grid;
  MPI_Comm comm_rows;
  MPI_Comm comm_columns;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(comm: MPI_COMM_WORLD, &proc_rank);
  MPI_Comm_size(comm: MPI_COMM_WORLD, size: &proc_count);

  init_dims(dims, proc_count);

  init_communicators(dims, &comm_grid, &comm_rows, &comm_columns);

  MPI_Cart_coords(comm_grid, proc_rank, maxdims: DIMS_COUNT, coords);

  A_block_size = ceil(x: (double)n_1 / dims[X]);
  B_block_size = ceil(x: (double)n_3 / dims[Y]);
  aligned_n1 = A_block_size * dims[X];
  aligned_n3 = B_block_size * dims[Y];

  if (coords[X] == 0 && coords[Y] == 0) {
    A = allocate_matrix(size: aligned_n1 * n_2);

    B = allocate_matrix(size: n_2 * aligned_n3);
    C = allocate_matrix(size: aligned_n1 * aligned_n3);

    generate_matrix(matrix: A, column: n_2, leading_row: n_1, leading_column: n_2, onRows: true);
    generate_matrix(matrix: B, column: aligned_n3, leading_row: n_2, leading_column: n_3, onRows: false);
  }

  A_block = allocate_matrix(A_block_size * n_2);
  B_block = allocate_matrix(B_block_size * n_2);
  C_block = allocate_matrix(A_block_size * B_block_size);

  start_time = MPI_Wtime();
```

```
177
178     init_communicators(dims, &comm_grid, &comm_rows, &comm_columns);
179
180     MPI_Cart_coords(comm_grid, proc_rank, maxdims: DIMS_COUNT, coords);
181
182     A_block_size = ceil(x: (double)n_1 / dims[X]);
183     B_block_size = ceil(x: (double)n_3 / dims[Y]);
184     aligned_n1 = A_block_size * dims[X];
185     aligned_n3 = B_block_size * dims[Y];
186
187     if (coords[X] == 0 && coords[Y] == 0) {
188       A = allocate_matrix(size: aligned_n1 * n_2);
189
190       B = allocate_matrix(size: n_2 * aligned_n3);
191       C = allocate_matrix(size: aligned_n1 * aligned_n3);
192
193       generate_matrix(matrix: A, column: n_2, leading_row: n_1, leading_column: n_2, onRows: true);
194       generate_matrix(matrix: B, column: aligned_n3, leading_row: n_2, leading_column: n_3, onRows: false);
195     }
196
197     A_block = allocate_matrix(A_block_size * n_2);
198     B_block = allocate_matrix(B_block_size * n_2);
199     C_block = allocate_matrix(A_block_size * B_block_size);
200
201     start_time = MPI_Wtime();
202
203     split_A(A, A_block, A_block_size, n_2, coords_y: coords[Y], comm_rows, comm_columns);
204     split_B(B, B_block, B_block_size, n_2, aligned_n3, coords_x: coords[X], comm_rows, comm_columns);
205
206     multiply(A_block, B_block, C_block, A_block_size, B_block_size, n_2);
207
208     compose(C_block, C, A_block_size, B_block_size, aligned_n1, aligned_n3, proc_count, comm_grid);
209
210     finish_time = MPI_Wtime();
211
212     if (coords[Y] == 0 && coords[X] == 0) {
213       printf(format: "Time: %lf\n", finish_time - start_time);
214
215       free(ptr: A);
216       free(ptr: B);
217       free(ptr: C);
218     }
219
220     free(ptr: A_block);
221     free(ptr: B_block);
222     free(ptr: C_block);
223     MPI_Comm_free(&comm_grid);
224     MPI_Comm_free(&comm_rows);
225     MPI_Comm_free(&comm_columns);
226
227     MPI_Finalize();
228
229     return EXIT_SUCCESS;
230 }
```