1 Отчет

1.1 Постановка задачи

Разработать процедуру, вычисляющую определитель вещественной матрицы A порядка N (первый шаг – приведение матрицы к верхнему треугольному виду). Обсновать проектное решение (выбор алгоритма). Обеспечить равномерную загрузку процессоров. Результат вывести в текстовый файл построчно. Использовать зависимость времени счета от размерности задачи и количества процессоров.

1.2 Процедура, вычисляющая определитель матрицы

В листинге 1 представлена программа на ЯП C, вычисляющую определитель вещественной матрицы.

Листинг 1 — Вычисление определителя вещественной матрицы

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

#define N 512

static double **allocate_matrix(int size)
{
        int i, j;
        double **matrix;

        matrix = malloc((size) * sizeof(double *));
        for (i = 0; i < size; i++)
        matrix[i] = malloc((size) * sizeof(double));

        return matrix;
}

static void **fill_random_matrix(double **matrix, int size)
{
        int i, j;

        for (i = 0; i < size; i++)
```

```
        for (j = 0; j < size; j++)
        matrix[i][j] = (rand() % 10) + 1;
}

int main(int argc, char** argv)
{
        double **A = allocate_matrix(N);

        fill_random_matrix(A, N);

        double det = 1;
        double norm = 0;
        int i, j, k;

        for (k = 0; k < N - 1; k++) {

                double max_val = abs(A[k][k]);
                int max_row = k;
                for (i = k + 1; i < N; i++) {
                        if (abs(A[i][k]) > max_val) {
                                max_val = abs(A[i][k]);
                                max_row = i;
                        }
                }

                if (max_row != k) {
                        for (j = k; j < N; j++) {
                                double temp = A[k][j];
                                A[k][j] = A[max_row][j];
                                A[max_row][j] = temp;
                        }
                        det *= -1.0;
                }

                for (i = k + 1; i < N; i++) {
                        double factor = A[i][k] / A[k][k];
                        for (j = k; j < N; j++) {
                                A[i][j] -= factor * A[k][j];
                        }
                }
```

```
        }

        for (i = 0; i < N; i++) {
                det *= A[i][i];
        }

        printf("Determinant: %lf\n", det);

        return 0;
}
```

1.3 Процедура, вычисляющая определитель матрицы параллельно

В листинге 2 представлена программа на ЯП C с использованием средств библиотеки MPI, реализующая вычисление определителя вещественной матрицы параллельно.

Листинг 2 — Параллельное вычисление определителя матрицы

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define N 100

static void print_matrix(double *matrix, size_t len)
{
        printf("\n");
        for (int row_idx = 0; row_idx < len; ++row_idx) {
                for (int col_idx = 0; col_idx < len;
                   ++col_idx) {
                        printf("%lf ", matrix[row_idx * len +
                           col_idx]);
                }
                printf("\n");
        }
}

static void print_array(double *arr, int size)
{
        for (int i = 0; i < size; i++) {
```

```
                printf("%lf ", arr[i]);
        }

        printf("\n");
}

static void init_matrix(double *matrix)
{
        for (size_t row_idx = 0; row_idx < N; ++row_idx)
        for (size_t col_idx = 0; col_idx < N; ++col_idx)
        matrix[row_idx * N + col_idx] = (rand() % 10) + 1;
}

#define SWAP(t, a, b) do { t c = a; a = b; b = c; } while (0)

static void swap(double *matrix, size_t count, size_t row1,
   size_t row2)
{
        for (size_t i = 0; i < count; i++)
        SWAP(double, matrix[i * N + row1], matrix[i * N +
           row2]);
}

static double gaussian(double *matrix, double *send_buffer,
   int num_cols, int rank, int size)
{
        double m_determinant = 0;
        int cur_control = 0;
        size_t swaps = 0;
        size_t cur_row = 0;
        size_t cur_index = 0;
        size_t row_swap;
        double det_val = 1;

        for (size_t i = 0; i < N; i++) {
                if (cur_control == rank) {
                        row_swap = cur_row;
                        double max = matrix[cur_index * N +
                           cur_row];
```

```
            for (size_t j = cur_row + 1; j < N;
               j++) {
                  if (matrix[cur_index * N + j]
                     > max) {
                        row_swap = j;
                        max =
                           matrix[cur_index *
                           N + j];
                  }
            }
      }

      MPI_Bcast(&row_swap, sizeof(size_t),
         MPI_BYTE, cur_control, MPI_COMM_WORLD);
      if (row_swap != cur_row) {
            swap(matrix, num_cols, cur_row,
               row_swap);
            swaps++;
      }

      if (cur_control == rank)
      for (size_t j = cur_row; j < N; j++)
      send_buffer[j] = matrix[cur_index * N + j] /
         matrix[cur_index * N  + cur_row];

      MPI_Bcast(send_buffer, N, MPI_DOUBLE,
         cur_control, MPI_COMM_WORLD);
      for (size_t j = 0; j < N; j++)
      for (size_t k = cur_row + 1; k < N; k++)
      matrix[j * N + k] -= matrix[j * N + cur_row]
         * send_buffer[k];

      if (cur_control == rank) {
            det_val = det_val * matrix[cur_index
               * N + cur_row];
            cur_index++;
      }

      cur_control++;
      if (cur_control == size)
```

```
                cur_control = 0;

                cur_row++;
        }

        MPI_Reduce(&det_val, &m_determinant, 1, MPI_DOUBLE,
           MPI_PROD, 0, MPI_COMM_WORLD);

        if (swaps % 2)
        m_determinant = -m_determinant;

        return m_determinant;
}

static void sort_by_process(double *list2, double *list1,
   size_t size)
{
        size_t index = 0;

        for (size_t i = 0; i < size; i++) {
                for (size_t j = i; j < N; j += size) {
                        list1[index] = list2[j];
                        index++;
                }
        }
}

int main(int argc, char *argv[])
{
        int rank, size;
        size_t num_rows, num_cols;
        double start_time, end_time;
        double det;
        double *matrix, *matrix_cpy, *ptr;
        double *send_buffer, *recv_buffer;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
if (!rank)
num_rows = N;

MPI_Bcast(&num_rows, sizeof(size_t), MPI_BYTE, 0,
   MPI_COMM_WORLD);

num_cols = num_rows / size;

matrix = malloc(sizeof(double) * N * N);
matrix_cpy = malloc(sizeof(double) * N * N);
ptr = matrix_cpy;

if (!rank) {
        init_matrix(matrix);
        memcpy(matrix_cpy, matrix, sizeof(double) * N
           * N);
}

send_buffer = malloc(sizeof(double) * N);
recv_buffer = malloc(sizeof(double) * num_cols);

for (size_t i = 0; i < N; i++) {
        if (!rank)
        sort_by_process(matrix_cpy, send_buffer,
           size);

        MPI_Scatter(send_buffer, num_cols,
           MPI_DOUBLE, recv_buffer, num_cols,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);

        if (!rank)
        matrix_cpy += N;

        for (size_t j = 0; j < num_cols; j++)
        matrix[j * N + i] = recv_buffer[j];
}

free(recv_buffer);

MPI_Barrier(MPI_COMM_WORLD);
```

```
        start_time = MPI_Wtime();

        det = gaussian(matrix, send_buffer, num_cols, rank,
           size);

        MPI_Barrier(MPI_COMM_WORLD);
        end_time = MPI_Wtime();

        if (!rank) {
                printf("Determinant: %lf\n", det);
                printf("MPI Time: %lf\n", end_time -
                   start_time);
        }

        free(matrix);
        free(ptr);
        free(send_buffer);

        MPI_Finalize();
        return 0;
}
```

1.4 Оценка эффективности параллельной реализации алгоритма

В целях оценки эффективности параллельной реализации алгоритма будут рассмотрены показатели ускорения, получаемого при использовании параллельного алгоритма для некоторого количества процессоров по сравнению с последовательным вариантом выполнения на различных размера матрицы.

Время исполнения было замерено с использованием внутренних средств библиотеки MPI.

Оценка эффективности и ускорения проводится для выполняемого на 1, 2, 4, 5, 8 и 10 процессорах при размерности квадратной матрицы 800 элементов и 1600.

1.4.1 Оценка ускорения.

На графике 1 представлена зависимость значения ускорения от количества используемых процессоров при размерах матрицы 800.

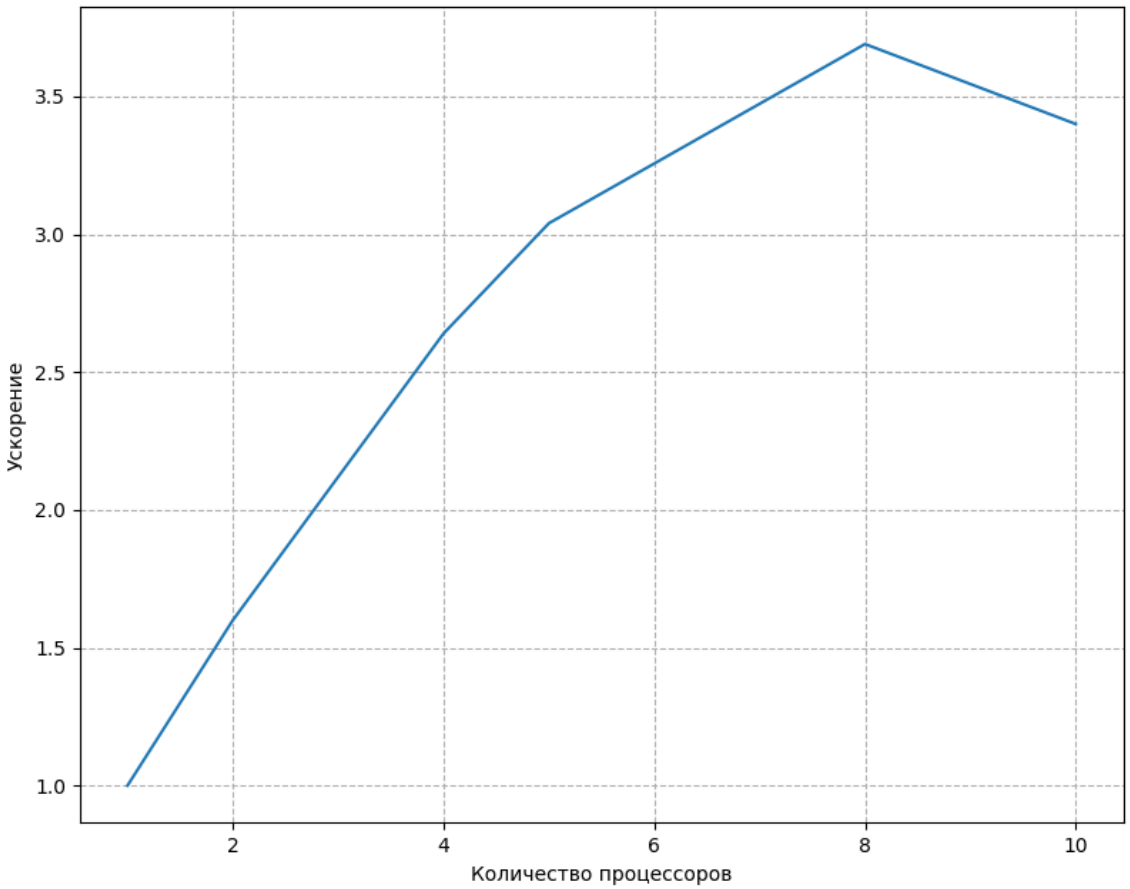

Рисунок 1 — Зависимость значения ускорения от количества используемых процессоров (размер матрицы 800)

Из представленного графика видно, что пик ускорения достигается при количестве процессоров, равному 8.

На графике 2 представлена зависимость значения ускорения от количества используемых процессоров при размерах матрицы 1600.

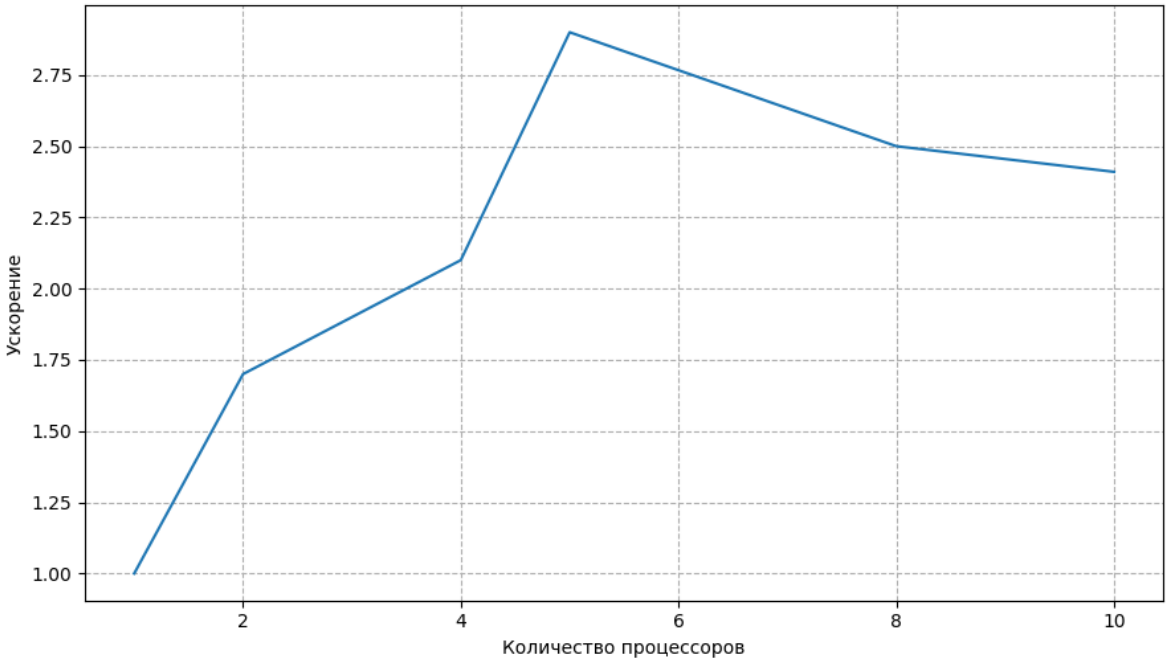

Рисунок 2 — Зависимость значения ускорения от количества используемых процессоров (размер матрицы 1600)

Из представленного графика видно, что пик ускорения достигается при количестве процессоров, равному 5.