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

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

студента 2 курса, группы 21209

Панас Матвей Алексеевич

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Мичуров М.А.

Новосибирск 2023

ЦЕЛЬ

Познакомиться с интерфейсом MPI на примере выполнения простого расчётного задания. Выполнить профилирование написанных MPI-программ

ЗАДАНИЕ

1. Написать программу на языке C/C++, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b методом простых итераций.

2. Программу распараллелить с помощью MPI. Реализовать два варианта программы: вариант 1: векторы x и b дублируются в каждом MPI-процессе; вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A.

3. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1,2, 4, 8, 16. Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

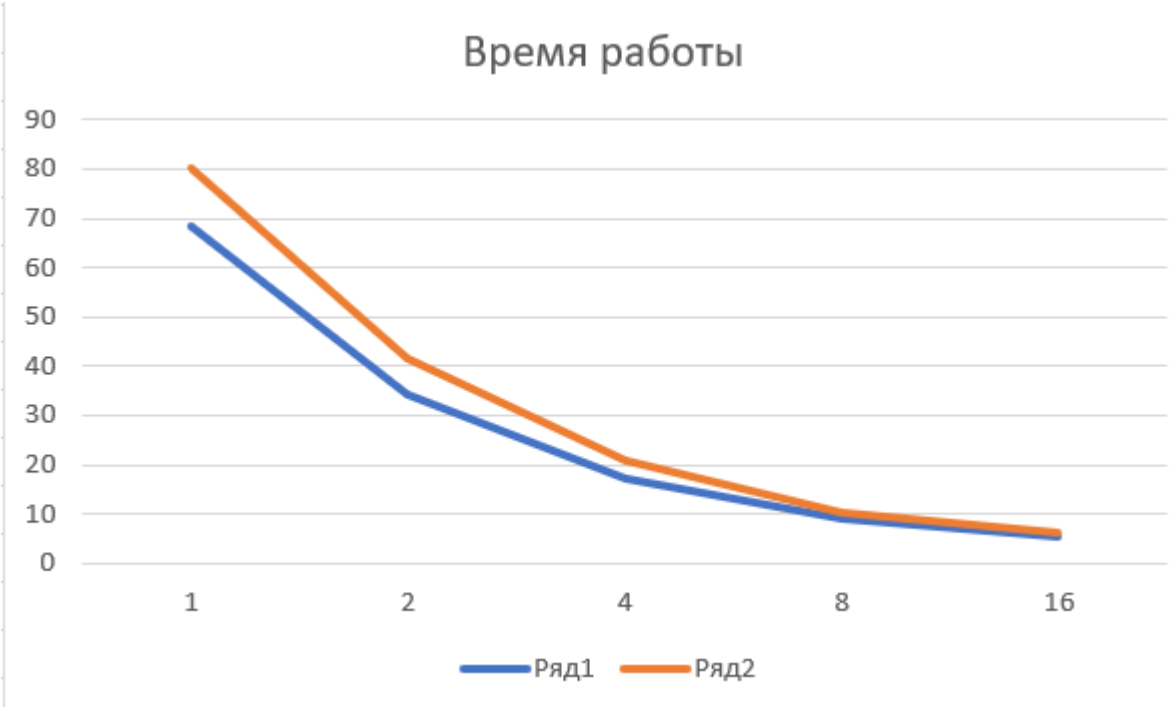
4. Выполнить профилирование двух вариантов программы при использовании 16-и ядер

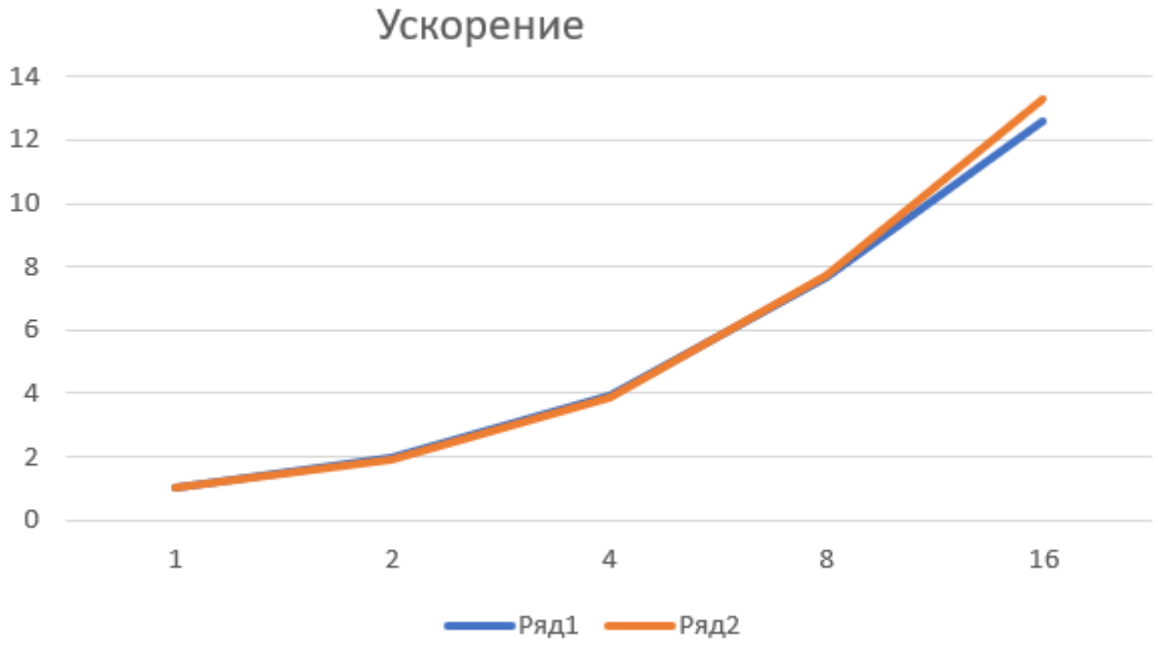
5. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы.

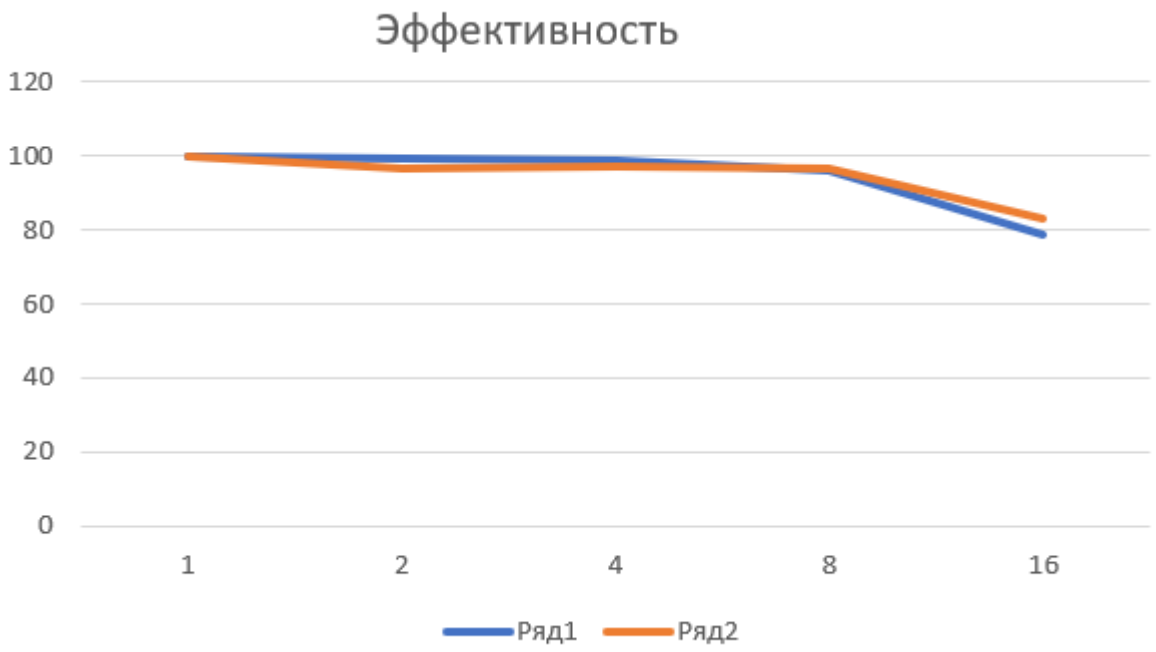
ОПИСАНИЕ РАБОТЫ

Была написана программа, реализующая метод простых итераций для решения системы линейных алгебраических уравнений (код в приложении 1). Затем она была распараллелена с использованием библиотеки MPI двумя способами: в первом матрица A разрезана между процессами по строкам, векторы x и b общие для всех процессов; во втором матрица A разрезана по строкам, векторы x и b разрезаны соответственно ей между всеми процессами. Замерено время
работы последовательной программы и двух параллельных при использовании 2, 4, 8, 16 процессорных ядер.
Вычислены ускорение и эффективность параллельных программ относительно последовательной. Получены следующие результаты:

Кол-во ядер	1	2	4	8	16
Время 1, с	68,42	34,3	17,3	8,92	5,43
Ускорение 1, раз	1	1,99	3,95	7,67	12,6
Эффективность 1, %	100	99,5	98,7	95,8	78,7
Время 2, с	80,3	41,45	20,62	10,35	6,03
Ускорение 2, раз	1	1,93	3,89	7,75	13,31
Эффективность 2, %	100	96,5	97,2	96,8	83,1


Время работы
90
80
70
60
50
40
30
20
10
0
1
2
4
8
16
Ряд1
Ряд2

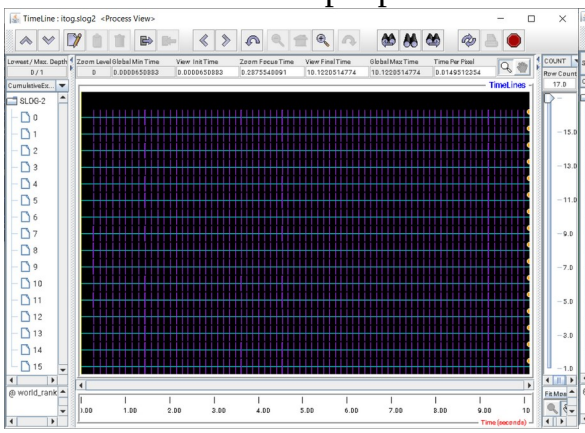

Ускорение
14
12
10
8
6
4
2
0
1
2
4
8
16
Ряд1
Ряд2

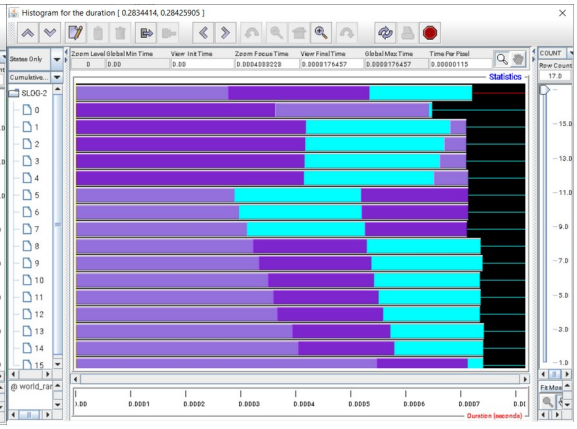

Эффективность
120
100
80
60
40
20
0
1
2
4
8
16
Ряд1
Ряд2

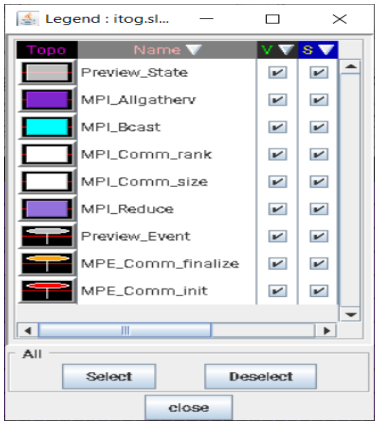
Как видно из графиков, эффективность программы при увеличении числа ядер уменьшается.

Ниже приведены результаты профилирования:

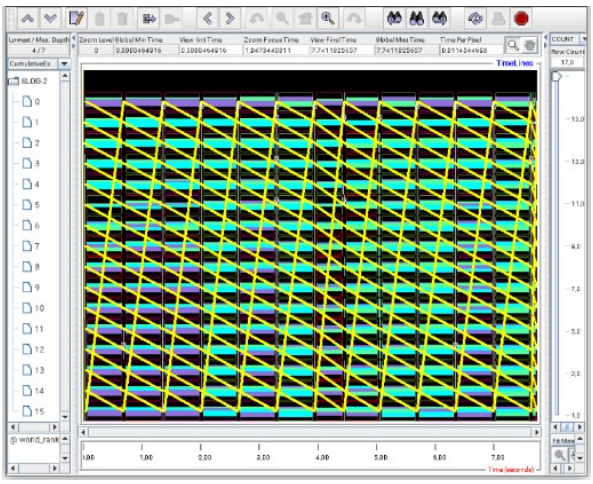
1 программа:

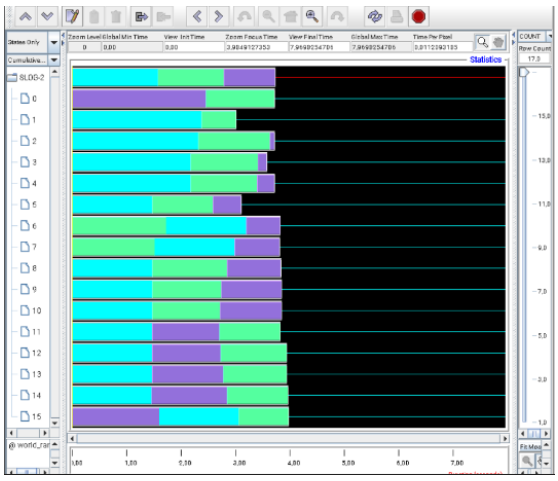


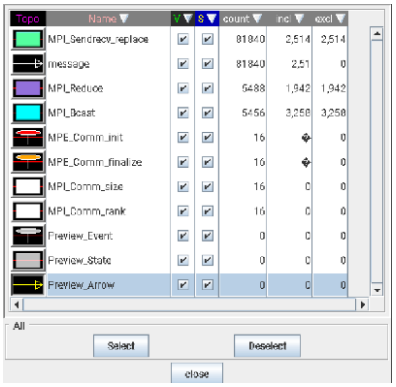




2 программа:







ЗАКЛЮЧЕНИЕ

В ходе выполнения работы обнаружено, что запуск распараллеленной программы на небольшом (2-4) числе ядер даёт пропорциональное этому числу ускорение работы программы. На большем числе ядер эффективность снижается, однако незначительно. Вариант параллельной программы с поделёнными между процессами векторами x и b (второй) оказался немного медленнее, так как в этом методе функция умножения части матрицы на вектор выполняется дольше из-за постоянной пересылки данных между процессами, хотя эффективность у него снижается медленнее.

Параллельная программа - I вариант

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

#define N 20000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 100000

void generate_A_chunks(double* A_chunk, int line_count, int line_size, int lineIndex) {
  for (int i = 0; i < line_count; i++) {
    for (int j = 0; j < line_size; ++j) {
      A_chunk[i * line_size + j] = 1;
    }

    A_chunk[i * line_size + lineIndex + i] = 2;
  }
}

void generate_x(double* x, int size) {
  for (int i = 0; i < size; i++) {
    x[i] = 0;
  }
}

void generate_b(double* b, int size) {
  for (int i = 0; i < size; i++) {
    b[i] = N + 1;
  }
}

void set_matrix_part(int* line_counts, int* line_offsets, int size, int process_count) {
  int offset = 0;
  for (int i = 0; i < process_count; ++i) {
    line_counts[i] = size / process_count;

    if (i < size % process_count) {
      ++line_counts[i];
    }

    line_offsets[i] = offset;
    offset += line_counts[i];
  }
}

double calc_norm_square(const double* vector, int size) {
  double norm_square = 0.0;
  for (int i = 0; i < size; ++i) {
    norm_square += vector[i] * vector[i];
  }

  return norm_square;
}

void calc_Axb(const double* A_chunk, const double* x, const double* b, double* Axb_chunk, int chunk_size, int chunk_offset) {
  for (int i = 0; i < chunk_size; ++i) {
    Axb_chunk[i] = -b[chunk_offset + i];
    for (int j = 0; j < N; ++j) Axb_chunk[i] += A_chunk[i * N + j] * x[j];
  }
}

void calc_next_x(const double* Axb_chunk, const double* x, double* x_chunk, double tau, int chunk_size, int chunk_offset) {
  for (int i = 0; i < chunk_size; ++i) x_chunk[i] = x[chunk_offset + i] - tau * Axb_chunk[i];
}

double* allocate_matrix(size_t n, int line) {
```

```c
double* allocate_matrix(size_t n, int line) {
  double* matrix = malloc(size: n * line * sizeof(double));
  check_print_error(matrix, message: "Failed to allocate memory to matrix\n");
  return matrix;
}

double* allocate_vector(size_t n) {
  double* vector = malloc(size: n * sizeof(double));
  check_print_error(a: vector, message: "Failed to allocate memory to vector\n");
  return vector;
}

int check_print_error(double* a, const char* message) {
  if (!a) {
    fprintf(stream: stderr, format: "%s", message);
    return 1;
  }
  return 0;
}
void print_vector(const double* vector, size_t n);

int main(int argc, char** argv) {
  int process_rank;
  int process_count;
  size_t iter_count;
  double b_norm;
  double accuracy = EPSILON + 1;
  double start_time;
  double finish_time;
  int* line_counts;
  int* line_offsets;
  double* A_chunk;
  double* x;
  double* b;
  double* Axb_chunk;
  double* x_chunk;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(comm: MPI_COMM_WORLD, size: &process_count);
  MPI_Comm_rank(comm: MPI_COMM_WORLD, &process_rank);

  line_counts = calloc(nmemb: process_count, size: sizeof(int));
  line_offsets = calloc(nmemb: process_count, size: sizeof(int));

  set_matrix_part(line_counts, line_offsets, size: N, process_count);

  A_chunk = allocate_matrix(n: N, line_counts[process_rank]);

  x = allocate_vector(n: N);
  b = allocate_vector(n: N);
  generate_A_chunks(A_chunk, line_counts[process_rank], line_size: N, lineIndex: line_offsets[process_rank]);
  generate_x(x, size: N);
  generate_b(b, size: N);

  b_norm = 0.0;
  if (process_rank == 0) {
    b_norm = sqrt(x: calc_norm_square(vector: b, size: N));
  }

  Axb_chunk = allocate_vector(line_counts[process_rank]);
  x_chunk = allocate_vector(line_counts[process_rank]);

  start_time = MPI_Wtime();

  for (iter_count = 0; accuracy > EPSILON && iter_count < MAX_ITERATION_COUNT; ++iter_count) {
```

```
    calc_Axb(A_chunk, x, b, Axb_chunk, chunk_size: line_counts[process_rank], chunk_offset: line_offsets[process_rank]);

    calc_next_x(Axb_chunk, x, x_chunk, tau: TAU, chunk_size: line_counts[process_rank], chunk_offset: line_offsets[process_rank]);
    MPI_Allgatherv(sendbuf: x_chunk, sendcount: line_counts[process_rank], sendtype: MPI_DOUBLE, recvbuf: x, recvcounts: line_counts, displs: line_offsets, recvtype: MPI_DOUBLE, comm: MPI_COMM_WORLD);

    double Axb_chunk_norm_square = calc_norm_square(vector: Axb_chunk, size: line_counts[process_rank]);
    MPI_Reduce(sendbuf: &Axb_chunk_norm_square, recvbuf: &accuracy, count: 1, datatype: MPI_DOUBLE, op: MPI_SUM, root: 0, comm: MPI_COMM_WORLD);

    if (process_rank == 0) {
      accuracy = sqrt(x: accuracy) / b_norm;
    }

    MPI_Bcast(buffer: &accuracy, count: 1, datatype: MPI_DOUBLE, root: 0, comm: MPI_COMM_WORLD);
  }

  finish_time = MPI_Wtime();

  if (process_rank == 0) {
    if (iter_count == MAX_ITERATION_COUNT)
      printf(format: "Too many iterations\n");
    else {
      // printf("Norm: %lf\n", sqrt(calc_norm_square(x, N)));
      // print_vector(x, N);
      printf(format: "Time: %lf sec\n", finish_time - start_time);
    }
  }

  free(ptr: line_counts);
  free(ptr: line_offsets);
  free(ptr: x);
  free(ptr: b);
  free(ptr: A_chunk);
  free(ptr: Axb_chunk);
  free(ptr: x_chunk);

  MPI_Finalize();

  return 0;
}

void print_vector(const double* vector, size_t n) {
  for (size_t i = 0; i < n; ++i) {
    printf(format: "%lf ", vector[i]);
  }
  printf(format: "\n");
}
```

Параллельная программа - II вариант

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

#define N 20000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 100000

void generate_A_chunk(double* A_chunk, int line_count, int line_size, int lineIndex) {
  for (int i = 0; i < line_count; i++) {
    for (int j = 0; j < line_size; ++j) {
      A_chunk[i * line_size + j] = 1;
    }

    A_chunk[i * line_size + lineIndex + i] = 2;
  }
}

void generate_x_chunk(double* x_chunk, int size) {
  for (int i = 0; i < size; i++) {
    x_chunk[i] = 0;
  }
}

void generate_b_chunk(double* b_chunk, int size) {
  for (int i = 0; i < size; i++) {
    b_chunk[i] = N + 1;
  }
}

void set_matrix_part(int* line_counts, int* line_offsets, int size, int process_count) {
  int offset = 0;
  for (int i = 0; i < process_count; ++i) {
    line_counts[i] = size / process_count;

    if (i < size % process_count) {
      ++line_counts[i];
    }

    line_offsets[i] = offset;
    offset += line_counts[i];
  }
}

double calc_norm_square(double* vector, int size) {
  double norm_square = 0.0;
  for (int i = 0; i < size; ++i) {
    norm_square += vector[i] * vector[i];
  }

  return norm_square;
}

void copy_vector(double* dest, const double* src, int size) {
  for (int i = 0; i < size; i++) {
    dest[i] = src[i];
  }
}

void calc_Axb(const double* A_chunk, const double* x_chunk, const double* b_chunk, double* recv_x_chunk, double* Axb_chunk, int* line_counts, int* line_offsets,
              int process_rank, int process_count) {
  int src_rank = (process_rank + process_count - 1) % process_count;
  int dest_rank = (process_rank + 1) % process_count;
  int current_rank;

```

```
  copy_vector(dest: recv_x_chunk, src: x_chunk, size: line_counts[process_rank]);

  for (int i = 0; i < process_count; ++i) {
    current_rank = (process_rank + i) % process_count;
    for (int j = 0; j < line_counts[process_rank]; ++j) {
      if (i == 0) Axb_chunk[j] = -b_chunk[j];
      {
        for (int k = 0; k < line_counts[current_rank]; ++k) {
          Axb_chunk[j] += A_chunk[j * N + line_offsets[current_rank] + k] * recv_x_chunk[k];
        }
      }
    }

    if (i != process_count - 1) {
      MPI_Sendrecv_replace(buf: recv_x_chunk, line_counts[0], datatype: MPI_DOUBLE, dest_rank, sendtag: process_rank, source: src_rank, recvtag: src_rank, comm: MPI_COMM_WORLD, status: MPI_STATUS_IGNORE);
    }
  }
}

void calc_next_x(const double* Axb_chunk, double* x_chunk, double tau, int chunk_size) {
  for (int i = 0; i < chunk_size; ++i) x_chunk[i] -= tau * Axb_chunk[i];
}

int check_print_error(double* a, const char* message) {
  if (!a) {
    fprintf(stream: stderr, format: "%s", message);
    return 1;
  }
  return 0;
}

double* allocate_matrix(size_t n, int line) {
  double* matrix = malloc(size: n * line * sizeof(double));
  check_print_error(matrix, message: "Failed to allocate memory to matrix\n");
  return matrix;
}

double* allocate_vector(size_t n) {
  double* vector = malloc(size: n * sizeof(double));
  check_print_error(a: vector, message: "Failed to allocate memory to vector\n");
  return vector;
}

int main(int argc, char** argv) {
  int process_rank;
  int process_count;
  int iter_count;
  double b_chunk_norm;
  double b_norm;
  double x_chunk_norm;
  double x_norm;
  double Axb_chunk_norm_square;
  double accuracy = EPSILON + 1;
  double start_time;
  double finish_time;
  int* line_counts;
  int* line_offsets;
  double* x_chunk;
  double* b_chunk;
  double* A_chunk;
  double* Axb_chunk;
  double* recv_x_chunk;

  MPI_Init(&argc, &argv);

```

```c
  MPI_Comm_size(comm: MPI_COMM_WORLD, size: &process_count);
  MPI_Comm_rank(comm: MPI_COMM_WORLD, &process_rank);

  line_counts = malloc(size: sizeof(int) * process_count);
  line_offsets = malloc(size: sizeof(int) * process_count);
  set_matrix_part(line_counts, line_offsets, size: N, process_count);

  x_chunk = allocate_vector(line_counts[process_rank]);
  b_chunk = allocate_vector(line_counts[process_rank]);
  A_chunk = allocate_vector(line_counts[process_rank] * N);

  generate_x_chunk(x_chunk, size: line_counts[process_rank]);
  generate_b_chunk(b_chunk, size: line_counts[process_rank]);
  generate_A_chunk(A_chunk, line_counts[process_rank], line_size: N, lineIndex: line_offsets[process_rank]);

  b_chunk_norm = calc_norm_square(vector: b_chunk, size: line_counts[process_rank]);
  MPI_Reduce(sendbuf: &b_chunk_norm, recvbuf: &b_norm, count: 1, datatype: MPI_DOUBLE, op: MPI_SUM, root: 0, comm: MPI_COMM_WORLD);

  if (process_rank == 0) {
    b_norm = sqrt(x: b_norm);
  }

  Axb_chunk = allocate_vector(line_counts[process_rank]);
  recv_x_chunk = allocate_vector(line_counts[0]);

  start_time = MPI_Wtime();

  for (iter_count = 0; accuracy > EPSILON && iter_count < MAX_ITERATION_COUNT; ++iter_count) {
    calc_Axb(A_chunk, x_chunk, b_chunk, recv_x_chunk, Axb_chunk, line_counts, line_offsets, process_rank, process_count);

    calc_next_x(Axb_chunk, x_chunk, tau: TAU, chunk_size: line_counts[process_rank]);

    Axb_chunk_norm_square = calc_norm_square(vector: Axb_chunk, size: line_counts[process_rank]);
    MPI_Reduce(sendbuf: &Axb_chunk_norm_square, recvbuf: &accuracy, count: 1, datatype: MPI_DOUBLE, op: MPI_SUM, root: 0, comm: MPI_COMM_WORLD);
    if (process_rank == 0) {
      accuracy = sqrt(x: accuracy) / b_norm;
    }
    MPI_Bcast(buffer: &accuracy, count: 1, datatype: MPI_DOUBLE, root: 0, comm: MPI_COMM_WORLD);
  }

  finish_time = MPI_Wtime();

  x_chunk_norm = calc_norm_square(vector: x_chunk, size: line_counts[process_rank]);
  MPI_Reduce(sendbuf: &x_chunk_norm, recvbuf: &x_norm, count: 1, datatype: MPI_DOUBLE, op: MPI_SUM, root: 0, comm: MPI_COMM_WORLD);

  if (process_rank == 0) {
    if (iter_count == MAX_ITERATION_COUNT) {
      fprintf(stream: stderr, format: "Too many iterations\n");
    } else {
      printf(format: "Norm: %lf\n", sqrt(x: x_norm));
      printf(format: "Time: %lf sec\n", finish_time - start_time);
    }
  }

  free(ptr: line_counts);
  free(ptr: line_offsets);
  free(ptr: x_chunk);
  free(ptr: b_chunk);
  free(ptr: A_chunk);
  free(ptr: Axb_chunk);

  MPI_Finalize();

  return 0;
}
```