

**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**

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

**Студента 2 курса, 21211 группы**

**Петрова Сергея Евгеньевича**

**Направление 09.03.01 – «Информатика и вычислительная техника»**

**Преподаватель:**

**Антон Юрьевич Кудинов**

**Новосибирск 2022**

## СОДЕРЖАНИЕ

<i>СОДЕРЖАНИЕ</i>	<i>2</i>
<i>ЦЕЛЬ</i>	<i>3</i>
<i>ЗАДАНИЕ</i>	<i>3</i>
<i>ВАРИАНТ ЗАДАНИЯ</i>	<i>4</i>
<i>Метод простой итерации</i>	<i>4</i>
<i>Исходные данные</i>	<i>4</i>
<i>ОПИСАНИЕ РАБОТЫ</i>	<i>5</i>
<i>Описание выполненной работы</i>	<i>5</i>
<i>Команды для компиляции и запуска</i>	<i>5</i>
<i>Результаты измерения</i>	<i>5</i>
<i>ЗАКЛЮЧЕНИЕ</i>	<i>7</i>
<i>ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)</i>	<i>8</i>
<i>mpi_slac_1.c</i>	<i>8</i>
<i>mpi_slac_2.c</i>	<i>11</i>
<i>run.sh</i>	<i>15</i>

## ЦЕЛЬ

*Ознакомиться с написанием параллельных программ при помощи MPI;*

## ЗАДАНИЕ

1. *Написать программу на языке C или C++, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида  $Ax = b$  в соответствии с выбранным вариантом. Здесь  $A$  – матрица размером  $N \times N$ ,  $x$  и  $b$  – векторы длины  $N$ . Тип элементов – *double*.*
  2. *Программу распараллелить с помощью MPI с разрезанием матрицы  $A$  по строкам на близкие по размеру, возможно не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Реализовать два варианта программы:*
    - a. *Векторы  $x$  и  $b$  дублируются в каждом MPI-процессе,*
    - b. *Векторы  $x$  и  $b$  разрезаются между MPI-процессами аналогично матрице  $A$ .*
- Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом).*
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. Исходные данные, параметры  $N$  и  $\varepsilon$  подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.*
  4. *Выполнить профилирование двух вариантов программы с помощью MPE при использовании 16 ядер.*
  5. *На основании полученных результатов сделать вывод о целесообразности использования первого или второго варианта программы.*

## ВАРИАНТ ЗАДАНИЯ

Пусть есть система из  $N$  линейных алгебраических уравнений в виде  $Ax = b$ , где  $A$  – матрица коэффициентов уравнений размером  $N \times N$ ,  $b$  – вектор правых частей размером  $N$ ,  $x$  – вектор решений размером  $N$ .

Решение системы уравнений итерационным методом состоит в выполнении следующих шагов:

1. Задается  $x^0$  – произвольное начальное приближение решения (вектор с произвольными начальными значениями);
2. Приближение многократно улучшается с использованием формулы вида  $x^{n+1} = f(x^n)$ , где функция  $f$  определяется используемым итерационным методом;
3. Процесс продолжается, пока не выполнится условие  $g(x^n) < \varepsilon$ , где функция  $g$  определяется используемым методом, а величина  $\varepsilon$  – требуемая точность;

### Метод простой итерации

В методе простой итерации преобразование решения на каждом шаге задается формулой:  $x^{n+1} = x^n - \tau(Ax^n - b)$ , где  $\tau$  – константа, параметр метода. Знак параметра  $\tau$  зависит от задачи. Если с некоторым знаком решение начинает расходиться, то следует сменить его на противоположный.

Критерий завершения счета  $\frac{\|Ax^n - b\|_2}{\|b\|_2} < \varepsilon$ , где  $\|u\|_2 = \sqrt{\sum_{i=1}^n u_i^2}$ .

### Исходные данные

Элементы главной диагонали матрицы  $A$  равны 2, остальные равны 1;

Все элементы вектора  $b$  равны  $N+1$ . В этом случае решением системы будет вектор, элементы которого равны 1.0;

Начальные значения элементов вектора  $x$  можно взять равными 0;

## ОПИСАНИЕ РАБОТЫ

### Описание выполненной работы

1. Написал 2 версии параллельной программы, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида  $Ax = b$  в соответствии с методом простой итерации;
2. Запустил 2 версии параллельной программы с одинаковыми параметрами ( $N = 5000$ ,  $\varepsilon = 10^{-7}$ ,  $\tau = 10^{-5}$ ) на кластере НГУ при использовании 1, 2, 4, 8, 16 ядер;
3. Построил графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер;
4. Выполнил профилирование двух вариантов программы с помощью MPE при использовании 16 ядер;

### Команды для компиляции и запуска

*Команда для компиляции MPI программы*

```
hpcuser265@clu:~> mpicc  
--std=c99  
-o mpi_slae_<version>  
mpi_slae_<version>.c
```

*Команда для компиляции MPI программы с использованием MPE*

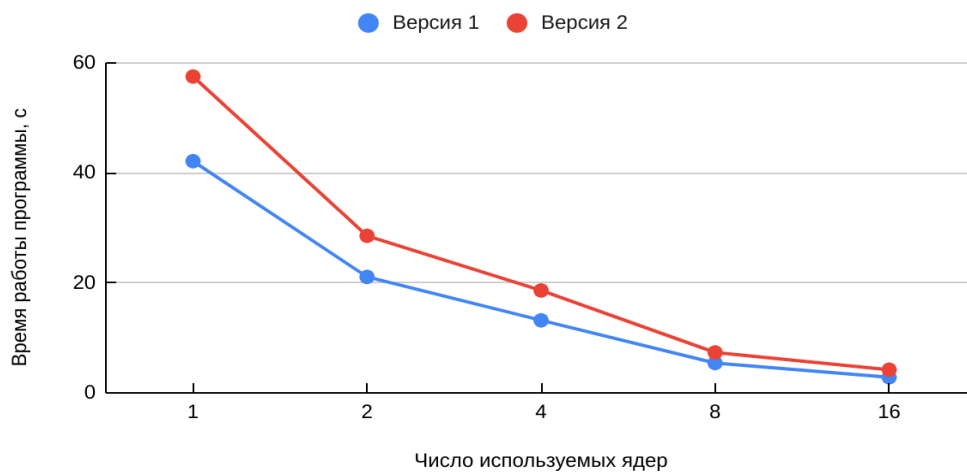
```
hpcuser265@clu:~> mpicc  
-mpilog  
--std=c99  
-o mpe_slae_<version>  
mpi_slae_<version>.c
```

*Команда для постановки задачи, описанной в скрипте run.sh, в очередь на кластере*

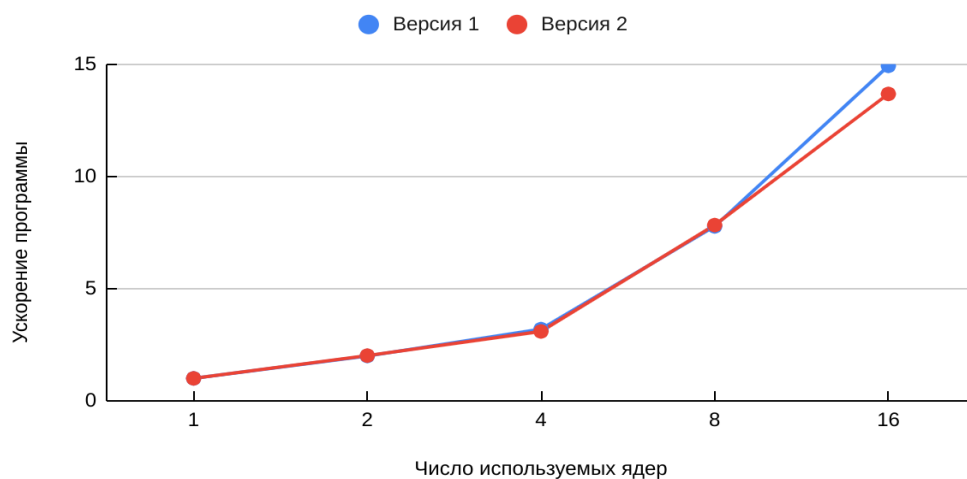
```
hpcuser265@clu:~> qsub run.sh
```

## Результаты измерения

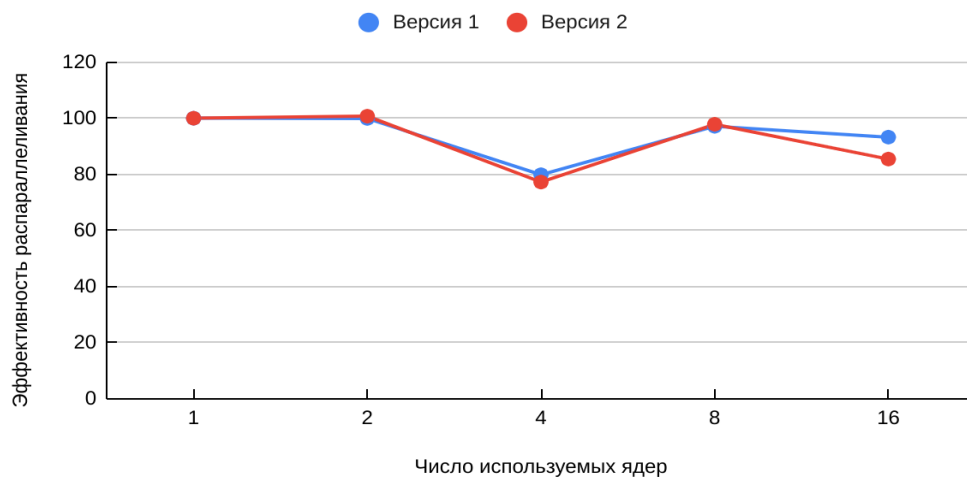
*Зависимость времени работы программы от числа ядер*



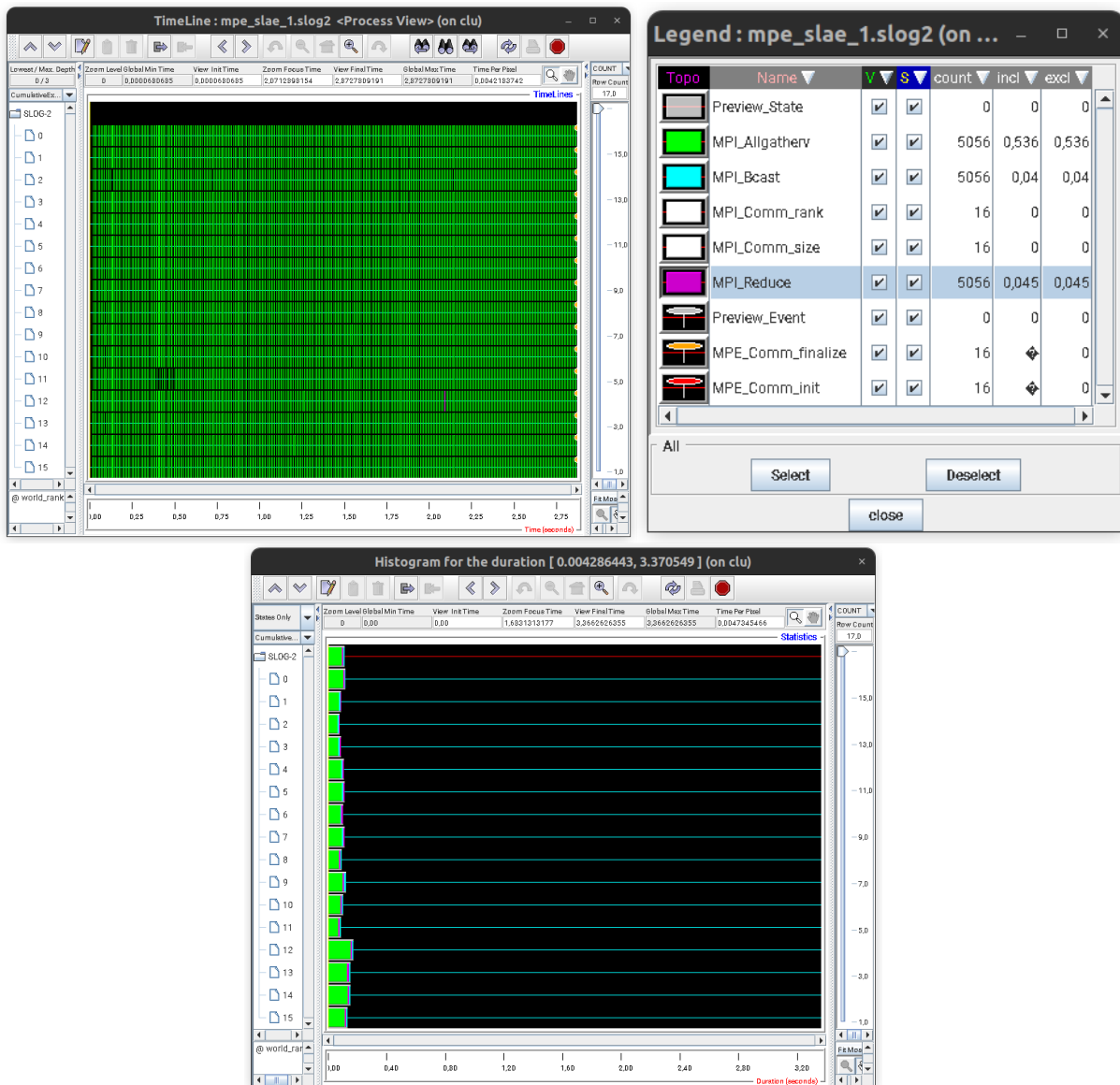
*Зависимость ускорения программы от числа ядер*



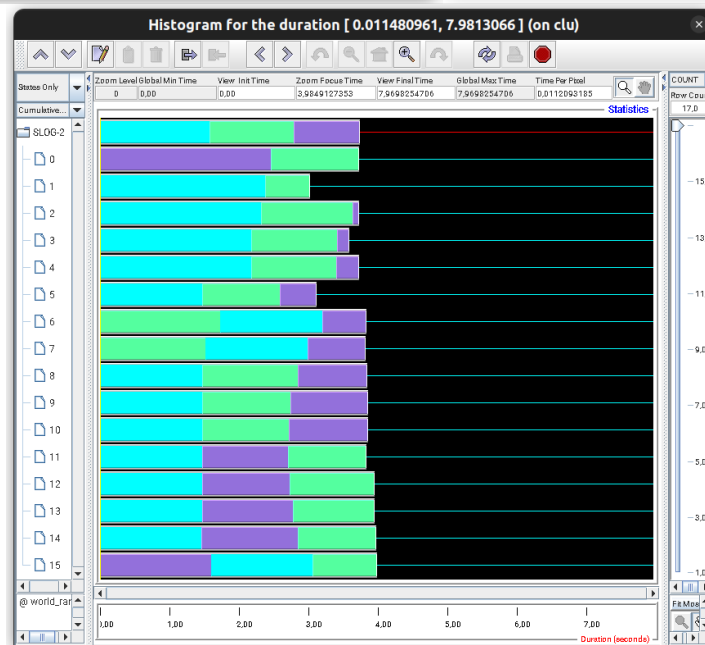
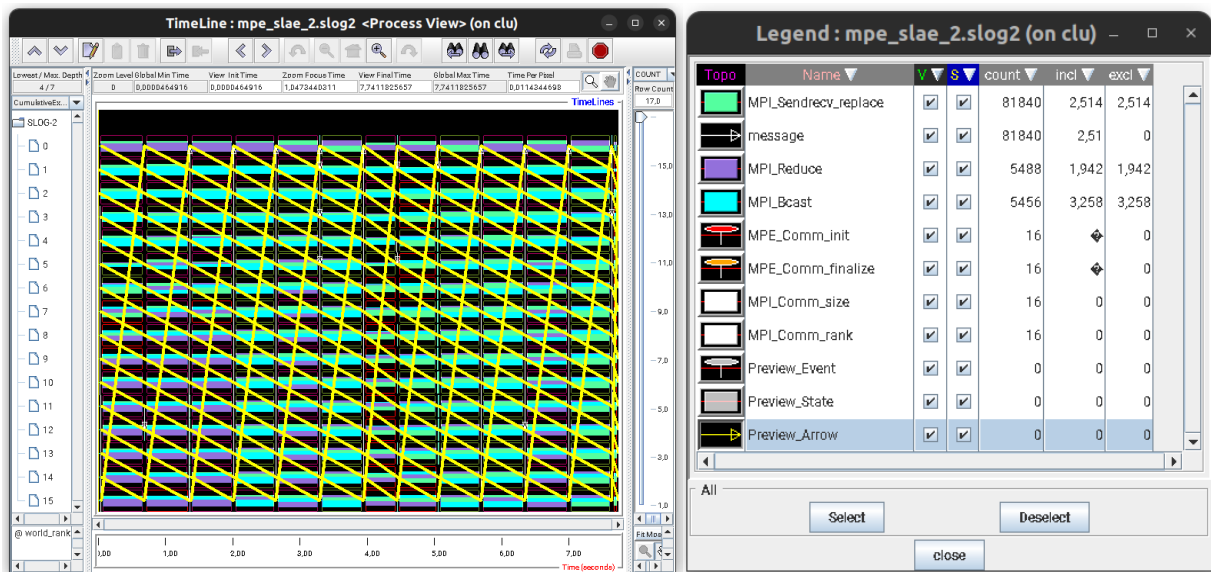
*Зависимость эффективности распараллеливания программы от числа ядер*



## Визуализация трассировочного файла первой версии программы



## Визуализация трассировочного файла второй версии программы





## ЗАКЛЮЧЕНИЕ

*В ходе выполнения лабораторной работы:*

- *ознакомился с написанием параллельных программ при помощи MPI;*
- *написал две версии 2 версии параллельной программы, которая реализует итерационный алгоритм решения системы линейных алгебраических уравнений вида  $Ax = b$  в соответствии с методом простой итерации;*
- *построил графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер;*
- *выполнил профилирование программ с помощью MPE при использовании 16 ядер;*

*Анализируя результаты измерений, можно сделать вывод, что первая версия программы быстрее второй при одних и тех же параметрах, т.к. достаточно времени тратится на пересылку данных между процессами. С другой стороны вторая версия расходует меньше памяти, т.к. в каждом процессе, в отличие от первой версии, находятся уникальные части матрицы  $A$ , векторов  $x$  и  $b$ .*

## ПРИЛОЖЕНИЕ (ЛИСТИНГ ПРОГРАММЫ)

*mpi\_slac\_1.c*

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

#define N 5000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 100000

void set_matrix_part(int *line_counts, int *line_offsets,
                    int size, int process_count);
void generate_A_chunks(double *A_chunk, int line_count,
                     int line_size, int process_rank);
void generate_x(double *x, int size);
void generate_b(double *b, int size);
double calc_norm_square(const double *vector, int size);
void calc_Axb(const double* A_chunk, const double* x,
              const double* b, double* Axb_chunk,
              int chunk_size, int chunk_offset);
void calc_next_x(const double* Axb_chunk, const double* x,
                 double* x_chunk, double tau,
                 int chunk_size, int chunk_offset);

int main(int argc, char **argv)
{
    int process_rank;
    int process_count;
    int iter_count;
    double b_norm;
    double Axb_chunk_norm_square;
    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(MPI_COMM_WORLD, &process_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

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

```

A_chunk = malloc(sizeof(double) * line_counts[process_rank] * N);
x = malloc(sizeof(double) * N);
b = malloc(sizeof(double) * N);
generate_A_chunks(A_chunk, line_counts[process_rank],
                  N, line_offsets[process_rank]);
generate_x(x, N);
generate_b(b, N);

if (process_rank == 0)
    b_norm = sqrt(calc_norm_square(b, N));

Axb_chunk = malloc(sizeof(double) * line_counts[process_rank]);
x_chunk = malloc(sizeof(double) * 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,
             line_counts[process_rank],
             line_offsets[process_rank]);

    calc_next_x(Axb_chunk, x, x_chunk, TAU,
                line_counts[process_rank],
                line_offsets[process_rank]);
    MPI_Allgather(x_chunk, line_counts[process_rank],
                  MPI_DOUBLE, x, line_counts, line_offsets,
                  MPI_DOUBLE, MPI_COMM_WORLD);

    Axb_chunk_norm_square = calc_norm_square(Axb_chunk,
                                              line_counts[process_rank]);
    MPI_Reduce(&Axb_chunk_norm_square, &accuracy, 1,
               MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (process_rank == 0)
        accuracy = sqrt(accuracy) / b_norm;
    MPI_Bcast(&accuracy, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

finish_time = MPI_Wtime();

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

free(line_counts);
free(line_offsets);

```

```

    free(x);
    free(b);
    free(A_chunk);
    free(Axb_chunk);
    free(x_chunk);

    MPI_Finalize();

    return 0;
}

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];
    }

```

### *mpi\_slac\_2.c*

```

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

#define N 5000
#define EPSILON 1E-7
#define TAU 1E-5
#define MAX_ITERATION_COUNT 100000

void set_matrix_part(int* line_counts, int* line_offsets,
                    int size, int process_count);
void generate_A_chunk(double* A_chunk, int line_count,
                     int line_size, int process_rank);
void generate_x_chunk(double* x_chunk, int size);
void generate_b_chunk(double* b_chunk, int size);
double calc_norm_square(double* vector, int size);
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);
void calc_next_x(const double* Axb, double* x_chunk,
                 double tau, int chunk_size);
void copy_vector(double* dest, const double* src, int size);

```

```

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);

    MPI_Comm_size(MPI_COMM_WORLD, &process_count);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

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

    x_chunk = malloc(sizeof(double) * line_counts[process_rank]);
    b_chunk = malloc(sizeof(double) * line_counts[process_rank]);
    A_chunk = malloc(sizeof(double) * line_counts[process_rank] * N);
    generate_x_chunk(x_chunk, line_counts[process_rank]);
    generate_b_chunk(b_chunk, line_counts[process_rank]);
    generate_A_chunk(A_chunk, line_counts[process_rank],
                    N, line_offsets[process_rank]);

    b_chunk_norm = calc_norm_square(b_chunk,
                                    line_counts[process_rank]);
    MPI_Reduce(&b_chunk_norm, &b_norm, 1, MPI_DOUBLE,
              MPI_SUM, 0, MPI_COMM_WORLD);
    if (process_rank == 0)
        b_norm = sqrt(b_norm);

    Axb_chunk = malloc(sizeof(double) * line_counts[process_rank]);
    recv_x_chunk = malloc(sizeof(double) * 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,
                    line_counts[process_rank]);

    Axb_chunk_norm_square = calc_norm_square(Axb_chunk,
                                              line_counts[process_rank]);
    MPI_Reduce(&Axb_chunk_norm_square, &accuracy, 1,
               MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (process_rank == 0)
        accuracy = sqrt(accuracy) / b_norm;
    MPI_Bcast(&accuracy, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

finish_time = MPI_Wtime();

x_chunk_norm = calc_norm_square(x_chunk,
                                line_counts[process_rank]);
MPI_Reduce(&x_chunk_norm, &x_norm, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);

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

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

MPI_Finalize();

return 0;
}

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 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(recv_x_chunk, x_chunk, 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] += recv_x_chunk[k] *
                A_chunk[j * N + k + line_offsets[current_rank]];
    }

    if (i != process_count - 1)
        MPI_Sendrecv_replace(recv_x_chunk, line_counts[0],
                              MPI_DOUBLE, dest_rank,
                              process_rank, src_rank, src_rank,
                              MPI_COMM_WORLD, 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];
}

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

```

*run.sh*

```

#!/bin/bash

#PBS -l walltime=00:05:00
#PBS -l select=<nodes>:ncpus=<cpus>:mpiprocs=<processes>
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')
echo "Number of MPI process: $MPI_NP"

echo 'File $PBS_NODEFILE:'
cat $PBS_NODEFILE
echo

mpirun -hostfile $PBS_NODEFILE -np $MPI_NP <MPI_program>

```