ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВОЙ ПРОЕКТ

по дисциплине “Параллельные вычислительные технологии”

на тему

Разработка параллельной MPI-программы решения СЛАУ методом Якоби

Выполнил студент Бирюков Никита Андреевич
Ф.И.О.

Группы ИС-241

Работу принял _____ профессор д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2024

Содержание

ВВЕДЕНИЕ 3
1 Математическое описание решения СЛАУ методом Якоби 4
1.1 Описание метода Якоби 4
1.2 Сходимость метода Якоби и критерий окончания итераций 4
2 Реализация программы решения СЛАУ методом Якоби 6
2.1 Инициализация данных 6
2.2 Проверка матрицы на применимость метода Якоби 6
2.3 Ход вычислений 7
2.4 Сравнение результатов с GNU Scientific Library 8
3 Реализация MPI-программы решения СЛАУ методом Якоби 10
3.1 Схема распараллеливания 10
3.2 Инициализация данных 11
2.3 Проверка матрицы на применимость метода Якоби 11
2.4 Ход вычислений и коммуникации между процессами 12
4 Анализ масштабируемости MPI-программы 14
4.1 Характеристики вычислительной системы Oak 14
4.2 Результаты анализа масштабируемости программы 14
ЗАКЛЮЧЕНИЕ 17
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 18
ПРИЛОЖЕНИЕ 19

ВВЕДЕНИЕ

Основой целью данной работой является реализация и анализ масштабируемости параллельной MPI-программы для решения СЛАУ методом Якоби. Решение систем линейных алгебраических уравнений (СЛАУ) является одной из ключевых задач в области вычислительной математики, применяемой во многих сферах науки, поскольку многие физические и инженерные задачи сводятся к решению СЛАУ.

В данной работе будет рассмотрен один из множества итерационных методов решения СЛАУ – метод Якоби, который особенно эффективен для задач с диагонально преобладающими матрицами. Итерационные методы являются предпочтительными при работе с большими матрицами, где использование прямых методов вычислительно затратно.

В рамках исследования будут рассмотрены теоретические основы метода Якоби для решения СЛАУ, особенности реализация последовательной версии, сравнение полученного результата с результатом GNU Scientific Library (GSL), а также ключевые аспекты разработки и анализа параллельной версии программы с использованием MPI. Помимо оценки корректности и производительности, особое внимание будет уделено анализу масштабируемости программы на многопроцессорных системах.

1 Математическое описание решения СЛАУ методом Якоби

1.1 Описание метода Якоби

Метод Якоби – один из наиболее простых методов приведения системы матрицы к виду, удобному для итерации. Пусть требуется решить численно решить СЛАУ:

$$\begin{cases} a_{11}x_1 + \ldots + a_{1n}x_n = b_1 \\ \cdots \\ a_{n1}x_1 + \ldots + a_{nn}x_n = b_n \end{cases} \quad (1.1)$$

Предполагается, что $a_{ii} \neq 0, i \in \{1, \ldots, n\}$, иначе метод Якоби не применим, поскольку метод подразумевает деление на a_{ii}. Далее выражаем x_1 – через первое выражение, x_2 – через второе и т. д.:

$$\begin{cases} x_1 = \dfrac{1}{a_{11}}(b_1 - a_{12}x_2 - \cdots - a_{1n}x_n) \\ \cdots \\ x_n = \dfrac{1}{a_{n1}}(b_n - a_{n1}x_1 - \cdots - a_{n(n-1)}x_{n-1}) \end{cases} \quad (1.2)$$

Последовательность приближений вектора $x^{(k)}$, где k – номер итерации, строится следующим образом. Выбирается первое приближение вектора $x^{(0)}$. Далее на каждой итерации метода вычисляется новое приближение на основе формулы (1.2), где в правой части подставляются элементы вектора $x^{(k-1)}$:

$$\begin{cases} x_1^{(k+1)} = \frac{1}{a_{11}}\left(b_1 - a_{12}x_2^{(k)} - \cdots - a_{1n}x_n^{(k)}\right) \\ \cdots \\ x_n^{(k+1)} = \frac{1}{a_{n1}}\left(b_n - a_{11}x_1^{(k)} - \cdots - a_{n(n-1)}x_{(n-1)}^{(k)}\right) \end{cases} \quad (1.3)$$

1.2 Сходимость метода Якоби и критерий окончания итераций

Достаточным признаком сходимости метода Якоби является диагональное преобладание матрицы коэффициентов:

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \qquad i = 1,2, \ldots, n \quad (1.4)$$

Таким образом перед началом вычислений необходимо проверить матрицу коэффициентов на диагональное преобладание и неравенство нулю элементов на главной диагонали.

Критерий окончания итераций при заданной точности ε имеет следующий вид:

$$\left\|x^{(k+1)} - x^{(k)}\right\| = \max\left(\left|x_i^{(k+1)} - x_i^{(k)}\right|\right), i = 1,2,\dots,n \quad (1.5)$$

$$\left\|x^{(k+1)} - x^{(k)}\right\| < \varepsilon \quad (1.6)$$

2 Реализация программы решения СЛАУ методом Якоби

2.1 Инициализация данных

Поскольку достаточным признаком сходимости метода Якоби является диагональное преобладание матрицы коэффициентов, то будем генерировать матрицу в соответствие с этим. Вектор свободных членов может быть любым.

Для инициализации матрицы коэффициентов и вектора свободных членов была реализована функция initialize, которая для каждой строки матрицы генерирует элементы в диапазоне от 1 до 100 и накапливает сумму этих элементов. Числа генерируются псевдослучайно и $seed$ для каждой строки заранее известен, что позволяет воспроизводить ход работы программы. Далее диагональному элементу прибавляется сумма всех элементов этой строки. Таким образом, проинициализированная матрица является диагонально преобладающей. Следовательно, можно использовать метод Якоби. Элементы вектора свободных членов также инициализируются в диапазоне от 1 до 100. Приведём код функции initialize:

```
22  void initialize(double* a, double* b, int n)
23  {
24      for (int i = 0; i < n; i++) {
25          srand(i * (n + 1));
26
27          double row_sum = 0;
28          for (int j = 0; j < n; j++) {
29              a[i * n + j] = rand() % 100 + 1;
30              row_sum += a[i * n + j];
31          }
32
33          a[i * n + i] += row_sum + (rand() % 50 + 10);
34          b[i] = rand() % 100 + 1;
35      }
36  }
```

2.2 Проверка матрицы на применимость метода Якоби

Так как в дальнейшем реализованный код может быть использован для настоящих вычислений, то была написана функция can_use_jacobi, которая проверяет матрицу коэффициентов на диагональное преобладание и неравенство

нулю элементов по главной диагонали. Если данная функция вернула true, то данную матрицу коэффициентов можно использовать для решения СЛАУ методом Якоби. Код функции can_use_jacobi:

```
 8  bool can_use_jacobi(const double* a, const int n)
 9  {
10      for (int i = 0; i < n; i++) {
11          if (a[i * n + i] == 0)
12              return false;
13
14          double row_sum = -fabs(a[i * n + i]);
15          for (int j = 0; j < n; j++) {
16              row_sum += fabs(a[i * n + j]);
17          }
18
19          if (row_sum >= fabs(a[i * n + i]))
20              return false;
21      }
22
23      return true;
24  }
```

2.3 Ход вычислений

После всех инициализации и всех проверок вызывается функция jacobi, которая решает СЛАУ методом Якоби. Внутри функции создаётся дополнительный массив для хранения временных значений вектора x. После чего запускается итерационный процесс. На каждой итерации в соответствие с формулой (1.3) вычисляется новое приближение, которое хранится во временном массиве temp:

```
37  for (int i = 0; i < n; i++) {
38      temp[i] = b[i];
39      for (int j = 0; j < n; j++) {
40          temp[i] -= (i != j) ? a[i * n + j] * x[j] : 0;
41      }
42      temp[i] /= a[i * n + i];
43  }
```

После каждой итерации по формуле (1.5) вычисляется разница между текущим и предыдущим решением и формируется текущее приближение вектора x:

```
45  delta = fabs(temp[0] - x[0]);
46  x[0] = temp[0];
47  for (int j = 1; j < n; j++) {
48      delta = fmax(delta, fabs(temp[j] - x[j]));
49      x[j] = temp[j];
50  }
```

Далее проверяется критерий окончания итерационного процесса (1.6):

```
36  while (delta > eps) {
…   …
51  }
```

После окончания итераций освобождается память, выделенная под временный массив.

2.4 Сравнение результатов с GNU Scientific Library

Для проверки корректности программы использовалась GNU Scientific Library (GSL). После получения вектора решения от GSL и вектора решения от функции jacobi поэлементно сравниваются вектора. Если разница между элементами векторов превышает заданный ε, то функция jacobi некорректна.

При всех запусках ε был равным 10^{-6}. Код для формирования решения от GSL:

```
41  int s;
42  gsl_matrix_view gsl_a = gsl_matrix_view_array(a, n, n);
43  gsl_vector_view gsl_b = gsl_vector_view_array(b, n);
44  gsl_vector* gsl_x = gsl_vector_alloc(n);
45
46  gsl_permutation* p = gsl_permutation_alloc(n);
47  gsl_linalg_LU_decomp(&gsl_a.matrix, p, &s);
48  gsl_linalg_LU_solve(&gsl_a.matrix, p, &gsl_b.vector, gsl_x);
```

Сравнение результатов от GSL и от функции jacobi:

```
57  for (int i = 0; i < n; i++) {
58      if (fabs(x[i] - gsl_vector_get(gsl_x, i)) > eps) {
59          fprintf(stderr,
60                  "Invalid result: elem %d: %f %f\n",
61                  i,
62                  x[i],
63                  gsl_vector_get(gsl_x, i));
64          break;
65      }
66  }
```

Запустим программу с $n = 6$ и сравним решения:

```
tobuso:~/cw-pct (main)$ ./bin/jacobi 6
JACOBI X[6]: 0.124490 0.142868 -0.086220 0.125462 0.195133 0.044121
GSL    X[6]: 0.124490 0.142869 -0.086220 0.125462 0.195133 0.044122
```

Как можно заметить разница между решениями не превышает заданный $\varepsilon = 10^{-6}$, следовательно написанная программа корректна.

3 Реализация MPI-программы решения СЛАУ методом Якоби

3.1 Схема распараллеливания

Для реализации параллельной программы была выбрана следующая схема: каждый процесс хранит только часть строк матрицы коэффициентов и часть вектора свободных членов. Вектор решения полностью хранится в каждом процессе, так как для вычисления каждого элемента нового приближения он нужен целиком.

Диапазон строк, обрабатываемых процессом, определяется функцией get_chunk, которая рассчитывает границы по рангу процесса и общему числу процессов. Каждому процессу достаётся по $n/commsize$ строк, где n – размерность матрицы, $commsize$ – количество процессов. Если остаток после деления не равен нулю, то первым $n \% commsize$ процессами достаётся на одну строку больше. Приведём код функции get_chunk:

```
 7    void get_chunk(int a, int b, int commsize, int rank, int* lb, int* ub)
 8    {
 9        int n = b - a + 1;
10        int q = n / commsize;
11        if (n % commsize)
12            q++;
13        int r = commsize * q - n;
14
15        int chunk = q;
16        if (rank >= commsize - r)
17            chunk = q - 1;
18
19        *lb = a;
20        if (rank > 0) {
21            if (rank <= commsize - r) {
22                *lb += q * rank;
23            } else {
24                int sum_of_big_chunks = q * (commsize - r);
25                int sum_of_small_chunks = (q - 1) * (rank - (commsize - r));
26                *lb += sum_of_big_chunks + sum_of_small_chunks;
27            }
28        }
29        *ub = *lb + chunk - 1;
30    }
```

3.2 Инициализация данных

Перед инициализацией данных процессам необходимо определить размерность матрицы, с которой придется работать. Нулевой процесс из аргументов командой строки берёт размерность матрицы и рассылает значение всем процессам с помощью операции MPI_Bcast:

```
40   if (rank == 0) {
41       n = (argc > 1) ? atoll(argv[1]) : 100;
42   }
43   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Инициализация данных в параллельной версии программы аналогична последовательной версии. Матрица коэффициентов генерируется диагонально преобладающей. Единственное отличие в том, что нужно учитывать положение элементов на главной диагонали, поскольку матрица хранится в распределённом виде. Диагональные элементы в распределенном виде хранятся со смещением, равным индексу первой обрабатываемой строки (lb) процесса в исходной матрице. Работа с элементами на главной диагонали выглядит следующим образом:

```
24  a[i * n + i + lb] += row_sum + (rand() % 50 + 10);
```

2.3 Проверка матрицы на применимость метода Якоби

Аналогично инициализации данных единственное отличие от последовательной версии будет только в определение элементов на главной диагонали. Если в одном из процессов обнаруживается, что элемент на главной диагонали не преобладает, то вызывается MPI_Abort:

```
55  if (!can_use_jacobi_mpi(a, n, lb, nrows)) {
56      fprintf(stderr, "matrix A can't be used for Jacobi method\n");
57      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
58  }
```

Изменённые части функции can_use_jacobi_mpi с учётом схемы распараллеливания:

```
36  if (a[i * n + i + lb] == 0)
37      return false;
…       …
44  if (row_sum >= fabs(a[i * n + i + lb]))
45      return false;
```

2.4 Ход вычислений и коммуникации между процессами

Перед началом итерационного процесса необходимо создать и проинициализировать массивы recvcounts и displs, которые в дальнейшем будут использоваться для коллективной операции MPI_Iallgatherv. Массивы recvcounts и displs определяют, какие части вектора x собираются от каждого процесса:

```
72  int offset = 0;
73  for (int i = 0; i < commsize; i++) {
74      int lb, ub;
75      get_chunk(0, n - 1, commsize, i, &lb, &ub);
76
77      recvcounts[i] = ub - lb + 1;
78      displs[i] = offset;
79      offset += recvcounts[i];
80  }
```

Для коммуникации между процессами были выбраны неблокирующие операции, поэтому необходимо использовать массив из двух элементов типа MPI_Request для отслеживания состояния операций:

```
82  MPI_Request reqs[2];
```

Каждый процесс вычисляет свою часть вектора x. Ход вычисления элементов вектора x практически не изменился. Только появилась необходимость учитывать положение элементов на главной диагонали:

```
84  for (int i = 0; i < nrows; i++) {
85      temp[i] = b[i];
86      for (int j = 0; j < n; j++) {
87          temp[i] -= (i + lb != j) ? a[i * n + j] * x[j] : 0;
88      }
89      temp[i] /= a[i * n + i + lb];
90  }
```

После хода итерации каждый процесс вычисляет локальную разницу между текущим и предыдущим решением. Далее, используя неблокирующую операцию MPI_Iallreduce, вычисляется максимальная разница, которая будет всем разослана:

```
 92  delta_local = fabs(temp[0] - x[lb]);
 93  for (int j = 1; j < nrows; j++)
 94      delta_local = fmax(delta_local, fabs(temp[j] - x[j + lb]));
 95  MPI_Iallreduce(
 96          &delta_local,
 97          &delta,
 98          1,
 99          MPI_DOUBLE,
100          MPI_MAX,
101          MPI_COMM_WORLD,
102          &reqs[0]);
```

Далее во всех процессах собирается вектор x с помощью неблокирующей коллективной операции MPI_Allgatherv:

```
103  MPI_Iallgatherv(
104          temp,
105          nrows,
106          MPI_DOUBLE,
107          x,
108          recvcounts,
109          displs,
110          MPI_DOUBLE,
111          MPI_COMM_WORLD,
112          &reqs[1]);
```

Поскольку использовались неблокирующие операции, воспользуемся операцией MPI_Waitall, чтобы дождаться завершения обмена между процессами:

```
113  MPI_Waitall(2, reqs, MPI_STATUS_IGNORE);
```

После хода итерационного процесса, как и в последовательной версии, проверяем критерий окончания итераций. После завершения итерационного процесса освобождаем использованную память.

4 Анализ масштабируемости MPI-программы

4.1 Характеристики вычислительной системы Oak

Все вычисления проводились на кафедральном вычислительном кластере Oak. На момент выполнения данной работы на кластере функционировало 4 узла на архитектуре x86_64: 2 x Intel Xeon Quad E5620, RAM 24 GB. Коммуникационная сеть: InfiniBand QDR (HCA Mellanox MT26428, switch Mellanox InfiniScale IV IS5030 QDR 36-Port), управляющая сеть: Gigabit Ethernet.

4.2 Результаты анализа масштабируемости программы

Для анализа масштабируемости программы было проведено 2 эксперимента: при $n = 2500$ и $n = 5000$. Это позволит проверить зависимость ускорения от объема входных данных.

В таблице 4.1 показаны результаты экспериментов.

Таблица 4.1 – Результаты экспериментов

Количество процессов	Время работы MPI-программы при различных n, сек	
	$n = 2500$	$n = 5000$
1	68,59	485,34
2 (2x1)	35,09	244,26
4 (2x2)	18,29	124,97
8 (2x4)	10,36	67,12
16 (2x8)	6,11	38,51
32 (2x16)	3,54	21,17

В таблице 4.2 показано ускорение параллельной программы при разном числе процессов и разных n.

Таблица 4.2 – Ускорение относительно последовательной версии

Количество процессов	Коэффициент ускорения при различных n	
	$n = 2500$	$n = 5000$
2 (2x1)	1,95	1,99
4 (2x2)	3,75	3,90
8 (2x4)	6,61	7,29
16 (2x8)	11,21	12,82
32 (2x16)	19,33	23,84

На рис. 4.1 продемонстрирована зависимость ускорения параллельной программы при разном числе процессов и разных n.

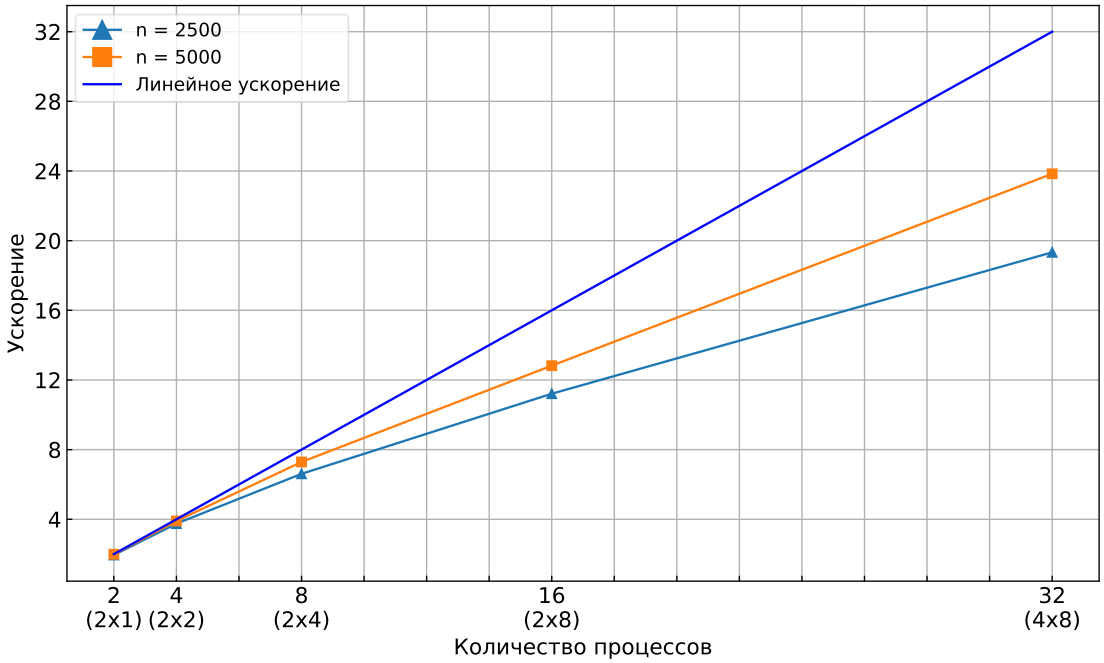

Рисунок 4.1 Зависимость ускорения от числа процессов

Из графика и таблиц видно, что программа достаточно хорошо масштабируется – коэффициент ускорения растёт с увеличением числа процессов. Также можно заметить, что при увеличении размера входных данных коэффициент ускорения возрастает. Однако стоит отметить, что из-за

особенностей метода Якоби идеального линейного ускорения не добиться, поскольку на каждом шаге итераций необходимо синхронизировать вектор решения между всеми процессами, что является трудоемкой коммуникационной операцией.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был исследован и реализован метод Якоби для решения СЛАУ. На основании проведённого анализа на вычислительной системе Oak можно сделать вывод, что разработанная параллельная MPI-программа для решения СЛАУ методом Якоби демонстрирует хорошую масштабируемость. Анализ результатов показал, что с увеличением числа процессов и размера входных данных коэффициент ускорения возрастает. Однако из-за необходимости частой синхронизации вектора решения между процессами идеального линейного ускорения не удалось достичь.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Теория и практика параллельных вычислений / Гергель В.П. – М.: Национальный Открытый Университет “ИНТУИТ”, 2016. – 501 с.

ПРИЛОЖЕНИЕ

1 Исходный код jacobi.h

```h
#pragma once
#include <stdbool.h>
#include <stdlib.h>

void jacobi(const double* a,
            const double* b,
            double* x,
            const int n,
            const double eps);
bool can_use_jacobi(const double* a, const int n);
```

2 Исходный код jacobi.c

```c
#include <assert.h>

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

#include <jacobi.h>

bool can_use_jacobi(const double* a, const int n)
{
    for (int i = 0; i < n; i++) {
        if (a[i * n + i] == 0)
            return false;

        double row_sum = -fabs(a[i * n + i]);
        for (int j = 0; j < n; j++) {
            row_sum += fabs(a[i * n + j]);
        }

        if (row_sum >= fabs(a[i * n + i]))
            return false;
    }

    return true;
}

void jacobi(const double* a,
            const double* b,
            double* x,
            const int n,
            const double eps)
{
    double delta = eps + 1;
    double* temp = malloc(sizeof(*temp) * n);
    assert(temp && "not enough memory");

```

```
36      while (delta > eps) {
37          for (int i = 0; i < n; i++) {
38              temp[i] = b[i];
39              for (int j = 0; j < n; j++) {
40                  temp[i] -= (i != j) ? a[i * n + j] * x[j] : 0;
41              }
42              temp[i] /= a[i * n + i];
43          }
44
45          delta = fabs(temp[0] - x[0]);
46          x[0] = temp[0];
47          for (int j = 1; j < n; j++) {
48              delta = fmax(delta, fabs(temp[j] - x[j]));
49              x[j] = temp[j];
50          }
51      }
52
53      free(temp);
54  }
```

3 Исходный код jacobi/main.c

```
 1  #include <assert.h>
 2  #include <inttypes.h>
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #include <sys/time.h>
 6
 7  #ifdef COMPARE_WITH_GSL
 8  #include <gsl/gsl_linalg.h>
 9  #endif
10
11  #include <jacobi.h>
12
13  #define PRINT_INFO 1
14
15  double wtime()
16  {
17      struct timeval t;
18      gettimeofday(&t, NULL);
19      return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
20  }
21
22  void initialize(double* a, double* b, int n)
23  {
24      for (int i = 0; i < n; i++) {
25          srand(i * (n + 1));
26
27          double row_sum = 0;
28          for (int j = 0; j < n; j++) {
29              a[i * n + j] = rand() % 100 + 1;
30              row_sum += a[i * n + j];
31          }
32
```

```
33          a[i * n + i] += row_sum + (rand() % 50 + 10);
34          b[i] = rand() % 100 + 1;
35      }
36  }
37
38  #ifdef COMPARE_WITH_GSL
39  void compare(double* a, double* b, double* x, int n, double eps)
40  {
41      int s;
42      gsl_matrix_view gsl_a = gsl_matrix_view_array(a, n, n);
43      gsl_vector_view gsl_b = gsl_vector_view_array(b, n);
44      gsl_vector* gsl_x = gsl_vector_alloc(n);
45
46      gsl_permutation* p = gsl_permutation_alloc(n);
47      gsl_linalg_LU_decomp(&gsl_a.matrix, p, &s);
48      gsl_linalg_LU_solve(&gsl_a.matrix, p, &gsl_b.vector, gsl_x);
49
50  #if PRINT_INFO
51      printf("GSL    X[%d]: ", n);
52      for (int i = 0; i < n; i++)
53          printf("%f ", gsl_vector_get(gsl_x, i));
54      printf("\n");
55  #endif
56
57      for (int i = 0; i < n; i++) {
58          if (fabs(x[i] - gsl_vector_get(gsl_x, i)) > eps) {
59              fprintf(stderr,
60                      "Invalid result: elem %d: %f %f\n",
61                      i,
62                      x[i],
63                      gsl_vector_get(gsl_x, i));
64              break;
65          }
66      }
67
68      gsl_permutation_free(p);
69      gsl_vector_free(gsl_x);
70  }
71  #endif
72
73  int main(int argc, char** argv)
74  {
75      if (argc != 2) {
76          fprintf(stderr, "Usage: jacobi <n>\n");
77          return EXIT_FAILURE;
78      }
79      double total_time = -wtime();
80
81      const double eps = 1e-6;
82      const int n = atoll(argv[1]);
83      double* a = malloc(sizeof(*a) * n * n);
84      double* b = malloc(sizeof(*b) * n);
85      double* x = calloc(sizeof(*x), n);
86      assert(a && b && x && "not enough memory");
87
```

```
88      initialize(a, b, n);
89      if (!can_use_jacobi(a, n)) {
90          fprintf(stderr, "matrix A can't be used for Jacobi method\n");
91          return EXIT_FAILURE;
92      }
93      jacobi(a, b, x, n, eps);
94
95  #if PRINT_INFO
96      printf("JACOBI X[%d]: ", n);
97      for (int i = 0; i < n; i++) {
98          printf("%lf ", x[i]);
99      }
100     printf("\n");
101 #endif
102
103 #if COMPARE_WITH_GSL
104     initialize(a, b, n);
105     compare(a, b, x, n, eps);
106 #endif
107     total_time += wtime();
108     printf("Jacobi (serial):\n[n=%d] time (sec): %.6lf\n",
109            n,
110            total_time);
111
112     free(a);
113     free(b);
114     free(x);
115     return 0;
116 }
```

4 Исходный код jacobi_mpi.h

```
1  #pragma once
2  #include <stdbool.h>
3  #include <stdlib.h>
4
5  void get_chunk(
6         int a,
7         int b,
8         int commsize,
9         int rank,
10        nt* lb,
11        int* ub);
12 void jacobi_mpi(
13        const double* a,
14        const double* b,
15        double* x,
16        const int n,
17        const double eps);
18 bool can_use_jacobi_mpi(
19        const double* a,
20        const int n,
21        const int lb,
22        const int nrows);
```

5 Исходный код jacobi_mpi.c

```
#include <assert.h>
#include <math.h>
#include <mpi.h>

#include <jacobi_mpi.h>

void get_chunk(int a, int b, int commsize, int rank, int* lb, int* ub)
{
    int n = b - a + 1;
    int q = n / commsize;
    if (n % commsize)
        q++;
    int r = commsize * q - n;

    int chunk = q;
    if (rank >= commsize - r)
        chunk = q - 1;

    *lb = a;
    if (rank > 0) {
        if (rank <= commsize - r) {
            *lb += q * rank;
        } else {
            int sum_of_big_chunks = q * (commsize - r);
            int sum_of_small_chunks = (q - 1) * (rank - (commsize - r));
            *lb += sum_of_big_chunks + sum_of_small_chunks;
        }
    }
    *ub = *lb + chunk - 1;
}

bool can_use_jacobi_mpi(
        const double* a, const int n, const int lb, const int nrows)
{
    for (int i = 0; i < nrows; i++) {
        if (a[i * n + i + lb] == 0)
            return false;

        double row_sum = -fabs(a[i * n + i + lb]);
        for (int j = 0; j < nrows; j++) {
            row_sum += fabs(a[i * n + j]);
        }

        if (row_sum >= fabs(a[i * n + i + lb]))
            return false;
    }

    return true;
}

void jacobi_mpi(
        const double* a,
        const double* b,
```

```
54          double* x,
55          const int n,
56          const double eps)
57  {
58      double delta = eps + 1, delta_local;
59      int lb, ub, commsize, rank;
60
61      MPI_Comm_size(MPI_COMM_WORLD, &commsize);
62      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
63
64      get_chunk(0, n - 1, commsize, rank, &lb, &ub);
65      int nrows = ub - lb + 1;
66
67      double* temp = malloc(sizeof(*temp) * nrows);
68      int* displs = malloc(sizeof(*displs) * commsize);
69      int* recvcounts = malloc(sizeof(*recvcounts) * commsize);
70      assert(temp && displs && recvcounts && "not enough memory");
71
72      int offset = 0;
73      for (int i = 0; i < commsize; i++) {
74          int lb, ub;
75          get_chunk(0, n - 1, commsize, i, &lb, &ub);
76
77          recvcounts[i] = ub - lb + 1;
78          displs[i] = offset;
79          offset += recvcounts[i];
80      }
81
82      MPI_Request reqs[2];
83      while (delta > eps) {
84          for (int i = 0; i < nrows; i++) {
85              temp[i] = b[i];
86              for (int j = 0; j < n; j++) {
87                  temp[i] -= (i + lb != j) ? a[i * n + j] * x[j] : 0;
88              }
89              temp[i] /= a[i * n + i + lb];
90          }
91
92          delta_local = fabs(temp[0] - x[lb]);
93          for (int j = 1; j < nrows; j++)
94              delta_local = fmax(delta_local, fabs(temp[j] - x[j + lb]));
95          MPI_Iallreduce(
96                  &delta_local,
97                  &delta,
98                  1,
99                  MPI_DOUBLE,
100                 MPI_MAX,
101                 MPI_COMM_WORLD,
102                 &reqs[0]);
103         MPI_Iallgatherv(
104                 temp,
105                 nrows,
106                 MPI_DOUBLE,
107                 x,
108                 recvcounts,
```

```
109                 displs,
110                 MPI_DOUBLE,
111                 MPI_COMM_WORLD,
112                 &reqs[1]);
113         MPI_Waitall(2, reqs, MPI_STATUS_IGNORE);
114     }
115
116     free(recvcounts);
117     free(displs);
118     free(temp);
119 }
```

6 Исходный код jacobi_mpi/main.c

```
1  #include <assert.h>
2  #include <mpi.h>
3  #include <stdio.h>
4
5  #include <jacobi_mpi.h>
6
7  #define PRINT_INFO 1
8
9  void initialize(double* a, double* b, int n, int commsize, int rank)
10 {
11     int lb, ub;
12     get_chunk(0, n - 1, commsize, rank, &lb, &ub);
13     int nrows = ub - lb + 1;
14
15     for (int i = 0; i < nrows; i++) {
16         srand((lb + i) * (n + 1));
17
18         double row_sum = 0;
19         for (int j = 0; j < n; j++) {
20             a[i * n + j] = rand() % 100 + 1;
21             row_sum += a[i * n + j];
22         }
23
24         a[i * n + i + lb] += row_sum + (rand() % 50 + 10);
25         b[i] = rand() % 100 + 1;
26     }
27 }
28
29 int main(int argc, char* argv[])
30 {
31     const double eps = 1e-6;
32     int commsize, rank;
33     double total_time = -MPI_Wtime();
34
35     MPI_Init(&argc, &argv);
36     MPI_Comm_size(MPI_COMM_WORLD, &commsize);
37     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
38
39     int n;
40     if (rank == 0) {
```

```
41          n = (argc > 1) ? atoll(argv[1]) : 100;
42      }
43      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
44
45      int lb, ub;
46      get_chunk(0, n - 1, commsize, rank, &lb, &ub);
47      int nrows = ub - lb + 1;
48
49      double* a = malloc(sizeof(*a) * n * nrows);
50      double* b = malloc(sizeof(*b) * nrows);
51      double* x = calloc(sizeof(*x), n);
52      assert(a && b && x && "not enough memory");
53
54      initialize(a, b, n, commsize, rank);
55      if (!can_use_jacobi_mpi(a, n, lb, nrows)) {
56          fprintf(stderr, "matrix A can't be used for Jacobi method\n");
57          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
58      }
59      jacobi_mpi(a, b, x, n, eps);
60
61      double total_max;
62      total_time += MPI_Wtime();
63      MPI_Reduce(
64              &total_time,
65              &total_max,
66              1,
67              MPI_DOUBLE,
68              MPI_MAX,
69              0,
70              MPI_COMM_WORLD);
71
72  #if PRINT_INFO
73      if (rank == 0) {
74          printf("JACOBI X[%d]: ", n);
75          for (int i = 0; i < n; i++) {
76              printf("%lf ", x[i]);
77          }
78          printf("\n");
79      }
80  #endif
81
82      if (rank == 0) {
83          printf("Jacobi (%d procs):\n", commsize);
84          printf("[n=%d] time (sec) %.6lf\n", n, total_max);
85      }
86
87      free(a);
88      free(b);
89      free(x);
90      MPI_Finalize();
91      return 0;
92  }
```