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](#_Toc185382362)

[1 Математическое описание решения СЛАУ методом Якоби 4](#_Toc185382363)

[1.1 Описание метода Якоби 4](#_Toc185382364)

[1.2 Сходимость метода Якоби и критерий окончания итераций 4](#_Toc185382365)

[2 Реализация программы решения СЛАУ методом Якоби 6](#_Toc185382366)

[2.1 Инициализация данных 6](#_Toc185382367)

[2.2 Проверка матрицы на применимость метода Якоби 6](#_Toc185382368)

[2.3 Ход вычислений 7](#_Toc185382369)

[2.4 Сравнение результатов с GNU Scientific Library 8](#_Toc185382370)

[3 Реализация MPI-программы решения СЛАУ методом Якоби 10](#_Toc185382371)

[3.1 Схема распараллеливания 10](#_Toc185382372)

[3.2 Инициализация данных 11](#_Toc185382373)

[2.3 Проверка матрицы на применимость метода Якоби 11](#_Toc185382374)

[2.4 Ход вычислений и коммуникации между процессами 12](#_Toc185382375)

[4 Анализ масштабируемости MPI-программы 14](#_Toc185382376)

[4.1 Характеристики вычислительной системы Oak 14](#_Toc185382377)

[4.2 Результаты анализа масштабируемости программы 14](#_Toc185382378)

[ЗАКЛЮЧЕНИЕ 17](#_Toc185382379)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 18](#_Toc185382380)

[ПРИЛОЖЕНИЕ 19](#_Toc185382381)

# 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:

Предполагается, что , иначе метод Якоби не применим, поскольку метод подразумевает деление на . Далее выражаем – через первое выражение, – через второе и т. д.:

Последовательность приближений вектора , где – номер итерации, строится следующим образом. Выбирается первое приближение вектора . Далее на каждой итерации метода вычисляется новое приближение на основе формулы (1.2), где в правой части подставляются элементы вектора :

## 1.2 Сходимость метода Якоби и критерий окончания итераций

Достаточным признаком сходимости метода Якоби является диагональное преобладание матрицы коэффициентов:

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 имеет следующий вид:

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

## 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  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **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;  } |

## 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 . После чего запускается итерационный процесс. На каждой итерации в соответствие с формулой (1.3) вычисляется новое приближение, которое хранится во временном массиве temp:

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

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

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

Далее проверяется критерий окончания итерационного процесса (1.6):

|  |  |
| --- | --- |
| 36  …  51 | **while** (delta > eps) {  …  } |

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 , то функция jacobi некорректна.

При всех запусках . Код для формирования решения от GSL:

|  |  |
| --- | --- |
| 41  42  43  44  45  46  47  48 | **int** s;  gsl\_matrix\_view gsl\_a = gsl\_matrix\_view\_array(a, n, n);  gsl\_vector\_view gsl\_b = gsl\_vector\_view\_array(b, n);  gsl\_vector\* gsl\_x = gsl\_vector\_alloc(n);  gsl\_permutation\* p = gsl\_permutation\_alloc(n);  gsl\_linalg\_LU\_decomp(&gsl\_a.matrix, p, &s);  gsl\_linalg\_LU\_solve(&gsl\_a.matrix, p, &gsl\_b.vector, gsl\_x); |

Сравнение результатов от GSL и от функции jacobi:

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

Запустим программу с 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 , 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 строк, где – размерность матрицы, – количество процессов. Если остаток после деления не равен нулю, то первым процессами достаётся на одну строку больше. Приведём код функции get\_chunk:

|  |  |
| --- | --- |
| 7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | **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**;  } |

## 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  41  42  43 | **if** (rank == **0**) {  n = (argc > **1**) ? atoll(argv[**1**]) : **100**;  }  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 () 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  56  57  58 | **if** (!can\_use\_jacobi\_mpi(a, n, lb, nrows)) {  fprintf(stderr, "matrix A can't be used for Jacobi method**\n**");  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  } |

Изменённые части функции can\_use\_jacobi\_mpi с учётом схемы распараллеливания:

|  |  |
| --- | --- |
| 36  37  …  44  45 | **if** (a[i \* n + i **+ lb**] == **0**)  **return** false;  …  **if** (row\_sum >= fabs(a[i \* n + i **+ lb**]))  **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 собираются от каждого процесса:

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

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 . Ход вычисления элементов вектора практически не изменился. Только появилась необходимость учитывать положение элементов на главной диагонали:

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

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  93  94  95  96  97  98  99  100  101  102 | delta\_local = fabs(temp[**0**] - x[lb]);  **for** (**int** j = **1**; j < nrows; j++)  delta\_local = fmax(delta\_local, fabs(temp[j] - x[j + lb]));  MPI\_Iallreduce(  &delta\_local,  &delta,  **1**,  MPI\_DOUBLE,  MPI\_MAX,  MPI\_COMM\_WORLD,  &reqs[**0**]); |

Далее во всех процессах собирается вектор с помощью неблокирующей коллективной операции MPI\_Allgatherv:

|  |  |
| --- | --- |
| 103  104  105  106  107  108  109  110  111  112 | MPI\_Iallgatherv(  temp,  nrows,  MPI\_DOUBLE,  x,  recvcounts,  displs,  MPI\_DOUBLE,  MPI\_COMM\_WORLD,  &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 и . 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 , сек | |
|  |  |
| 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 .

Таблица 4.2 – Ускорение относительно последовательной версии

|  |  |  |
| --- | --- | --- |
| Количество процессов | Коэффициент ускорения при различных | |
|  |  |
| 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 .

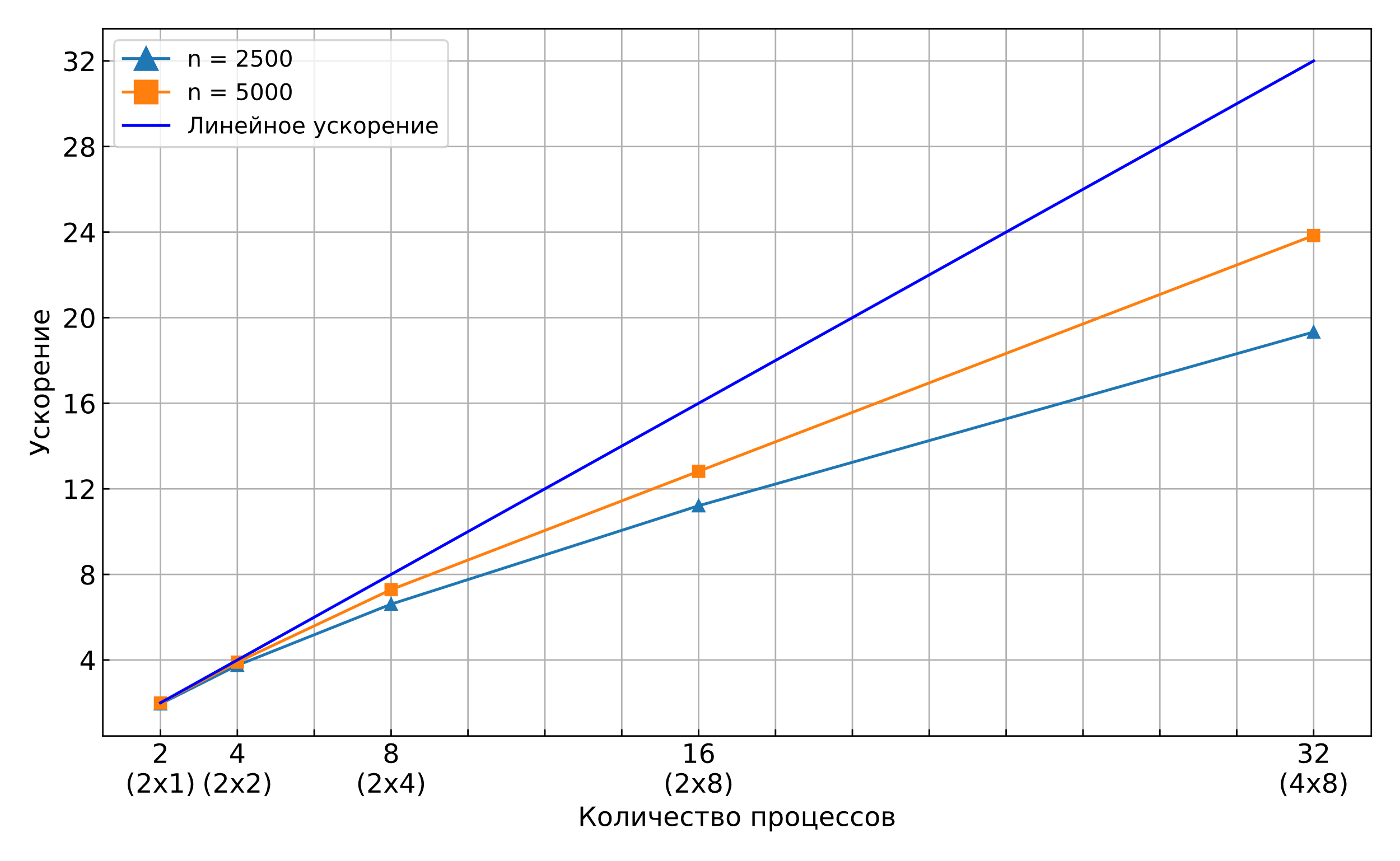


Рисунок 4.1 Зависимость ускорения от числа процессов

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 c.

# ПРИЛОЖЕНИЕ

1 Исходный код jacobi.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | #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");  **while** (delta > eps) {  **for** (**int** i = **0**; i < n; i++) {  temp[i] = b[i];  **for** (**int** j = **0**; j < n; j++) {  temp[i] -= (i != j) ? a[i \* n + j] \* x[j] : **0**;  }  temp[i] /= a[i \* n + i];  }  delta = fabs(temp[**0**] - x[**0**]);  x[**0**] = temp[**0**];  **for** (**int** j = **1**; j < n; j++) {  delta = fmax(delta, fabs(temp[j] - x[j]));  x[j] = temp[j];  }  }  free(temp);  } |

3 Исходный код jacobi/main.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116 | #include <assert.h>  #include <inttypes.h>  #include <stdio.h>  #include <stdlib.h>  #include <sys/time.h>  #ifdef COMPARE\_WITH\_GSL  #include <gsl/gsl\_linalg.h>  #endif  #include <jacobi.h>  #define PRINT\_INFO 1  **double** **wtime**()  {  **struct** timeval t;  gettimeofday(&t, NULL);  **return** (**double**)t.tv\_sec + (**double**)t.tv\_usec \* **1E-6**;  }  **void** **initialize**(**double**\* a, **double**\* b, **int** n)  {  **for** (**int** i = **0**; i < n; i++) {  srand(i \* (n + **1**));  **double** row\_sum = **0**;  **for** (**int** j = **0**; j < n; j++) {  a[i \* n + j] = rand() % **100** + **1**;  row\_sum += a[i \* n + j];  }  a[i \* n + i] += row\_sum + (rand() % **50** + **10**);  b[i] = rand() % **100** + **1**;  }  }  #ifdef COMPARE\_WITH\_GSL  **void** **compare**(**double**\* a, **double**\* b, **double**\* x, **int** n, **double** eps)  {  **int** s;  gsl\_matrix\_view gsl\_a = gsl\_matrix\_view\_array(a, n, n);  gsl\_vector\_view gsl\_b = gsl\_vector\_view\_array(b, n);  gsl\_vector\* gsl\_x = gsl\_vector\_alloc(n);  gsl\_permutation\* p = gsl\_permutation\_alloc(n);  gsl\_linalg\_LU\_decomp(&gsl\_a.matrix, p, &s);  gsl\_linalg\_LU\_solve(&gsl\_a.matrix, p, &gsl\_b.vector, gsl\_x);  #if PRINT\_INFO  printf("GSL X[%d]: ", n);  **for** (**int** i = **0**; i < n; i++)  printf("%f ", gsl\_vector\_get(gsl\_x, i));  printf("**\n**");  #endif  **for** (**int** i = **0**; i < n; i++) {  **if** (fabs(x[i] - gsl\_vector\_get(gsl\_x, i)) > eps) {  fprintf(stderr,  "Invalid result: elem %d: %f %f**\n**",  i,  x[i],  gsl\_vector\_get(gsl\_x, i));  **break**;  }  }  gsl\_permutation\_free(p);  gsl\_vector\_free(gsl\_x);  }  #endif  **int** **main**(**int** argc, **char**\*\* argv)  {  **if** (argc != **2**) {  fprintf(stderr, "Usage: jacobi <n>**\n**");  **return** EXIT\_FAILURE;  }  **double** total\_time = -wtime();  **const** **double** eps = **1e-6**;  **const** **int** n = atoll(argv[**1**]);  **double**\* a = malloc(**sizeof**(\*a) \* n \* n);  **double**\* b = malloc(**sizeof**(\*b) \* n);  **double**\* x = calloc(**sizeof**(\*x), n);  assert(a && b && x && "not enough memory");  initialize(a, b, n);  **if** (!can\_use\_jacobi(a, n)) {  fprintf(stderr, "matrix A can't be used for Jacobi method**\n**");  **return** EXIT\_FAILURE;  }  jacobi(a, b, x, n, eps);  #if PRINT\_INFO  printf("JACOBI X[%d]: ", n);  **for** (**int** i = **0**; i < n; i++) {  printf("%lf ", x[i]);  }  printf("**\n**");  #endif  #if COMPARE\_WITH\_GSL  initialize(a, b, n);  compare(a, b, x, n, eps);  #endif  total\_time += wtime();  printf("Jacobi (serial):**\n**[n=%d] time (sec): %.6lf**\n**",  n,  total\_time);  free(a);  free(b);  free(x);  **return** **0**;  } |

4 Исходный код jacobi\_mpi.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | #pragma once  #include <stdbool.h>  #include <stdlib.h>  **void** **get\_chunk**(  **int** a,  **int** b,  **int** commsize,  **int** rank,  nt\* lb,  **int**\* ub);  **void** **jacobi\_mpi**(  **const** **double**\* a,  **const** **double**\* b,  **double**\* x,  **const** **int** n,  **const** **double** eps);  **bool** **can\_use\_jacobi\_mpi**(  **const** **double**\* a,  **const** **int** n,  **const** **int** lb,  **const** **int** nrows); |

5 Исходный код jacobi\_mpi.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119 | #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,  **double**\* x,  **const** **int** n,  **const** **double** eps)  {  **double** delta = eps + **1**, delta\_local;  **int** lb, ub, commsize, rank;  MPI\_Comm\_size(MPI\_COMM\_WORLD, &commsize);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  get\_chunk(**0**, n - **1**, commsize, rank, &lb, &ub);  **int** nrows = ub - lb + **1**;  **double**\* temp = malloc(**sizeof**(\*temp) \* nrows);  **int**\* displs = malloc(**sizeof**(\*displs) \* commsize);  **int**\* recvcounts = malloc(**sizeof**(\*recvcounts) \* commsize);  assert(temp && displs && recvcounts && "not enough memory");  **int** offset = **0**;  **for** (**int** i = **0**; i < commsize; i++) {  **int** lb, ub;  get\_chunk(**0**, n - **1**, commsize, i, &lb, &ub);  recvcounts[i] = ub - lb + **1**;  displs[i] = offset;  offset += recvcounts[i];  }  MPI\_Request reqs[**2**];  **while** (delta > eps) {  **for** (**int** i = **0**; i < nrows; i++) {  temp[i] = b[i];  **for** (**int** j = **0**; j < n; j++) {  temp[i] -= (i + lb != j) ? a[i \* n + j] \* x[j] : **0**;  }  temp[i] /= a[i \* n + i + lb];  }  delta\_local = fabs(temp[**0**] - x[lb]);  **for** (**int** j = **1**; j < nrows; j++)  delta\_local = fmax(delta\_local, fabs(temp[j] - x[j + lb]));  MPI\_Iallreduce(  &delta\_local,  &delta,  **1**,  MPI\_DOUBLE,  MPI\_MAX,  MPI\_COMM\_WORLD,  &reqs[**0**]);  MPI\_Iallgatherv(  temp,  nrows,  MPI\_DOUBLE,  x,  recvcounts,  displs,  MPI\_DOUBLE,  MPI\_COMM\_WORLD,  &reqs[**1**]);  MPI\_Waitall(**2**, reqs, MPI\_STATUS\_IGNORE);  }  free(recvcounts);  free(displs);  free(temp);  } |

6 Исходный код jacobi\_mpi/main.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92 | #include <assert.h>  #include <mpi.h>  #include <stdio.h>  #include <jacobi\_mpi.h>  #define PRINT\_INFO 1  **void** **initialize**(**double**\* a, **double**\* b, **int** n, **int** commsize, **int** rank)  {  **int** lb, ub;  get\_chunk(**0**, n - **1**, commsize, rank, &lb, &ub);  **int** nrows = ub - lb + **1**;  **for** (**int** i = **0**; i < nrows; i++) {  srand((lb + i) \* (n + **1**));  **double** row\_sum = **0**;  **for** (**int** j = **0**; j < n; j++) {  a[i \* n + j] = rand() % **100** + **1**;  row\_sum += a[i \* n + j];  }  a[i \* n + i + lb] += row\_sum + (rand() % **50** + **10**);  b[i] = rand() % **100** + **1**;  }  }  **int** **main**(**int** argc, **char**\* argv[])  {  **const** **double** eps = **1e-6**;  **int** commsize, rank;  **double** total\_time = -MPI\_Wtime();  MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &commsize);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  **int** n;  **if** (rank == **0**) {  n = (argc > **1**) ? atoll(argv[**1**]) : **100**;  }  MPI\_Bcast(&n, **1**, MPI\_INT, **0**, MPI\_COMM\_WORLD);  **int** lb, ub;  get\_chunk(**0**, n - **1**, commsize, rank, &lb, &ub);  **int** nrows = ub - lb + **1**;  **double**\* a = malloc(**sizeof**(\*a) \* n \* nrows);  **double**\* b = malloc(**sizeof**(\*b) \* nrows);  **double**\* x = calloc(**sizeof**(\*x), n);  assert(a && b && x && "not enough memory");  initialize(a, b, n, commsize, rank);  **if** (!can\_use\_jacobi\_mpi(a, n, lb, nrows)) {  fprintf(stderr, "matrix A can't be used for Jacobi method**\n**");  MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);  }  jacobi\_mpi(a, b, x, n, eps);  **double** total\_max;  total\_time += MPI\_Wtime();  MPI\_Reduce(  &total\_time,  &total\_max,  **1**,  MPI\_DOUBLE,  MPI\_MAX,  **0**,  MPI\_COMM\_WORLD);  #if PRINT\_INFO  **if** (rank == **0**) {  printf("JACOBI X[%d]: ", n);  **for** (**int** i = **0**; i < n; i++) {  printf("%lf ", x[i]);  }  printf("**\n**");  }  #endif  **if** (rank == **0**) {  printf("Jacobi (%d procs):**\n**", commsize);  printf("[n=%d] time (sec) %.6lf**\n**", n, total\_max);  }  free(a);  free(b);  free(x);  MPI\_Finalize();  **return** **0**;  } |