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 Пеалкиви Даниил Яковлевич
Ф.И.О.

Группы ИС-242

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

Защищена _____ Оценка _____

Новосибирск – 2024

СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Метод Якоби представляет собой классический итерационный метод для решения систем линейных алгебраических уравнений (СЛАУ). Этот метод основан на разложении матрицы системы на ее диагональную, нижнюю треугольную и верхнюю треугольную составляющие. Решение находится посредством последовательных приближений, начиная с некоторого начального вектора. Хотя метод Якоби прост в реализации и обладает хорошей устойчивостью, он может оказаться медленным при решении крупных задач, особенно на одноядерных процессорах.

Для повышения эффективности решения таких задач метод Якоби был адаптирован для параллельных вычислений с использованием технологии MPI. Применение MPI позволяет распределить вычисления между несколькими процессами, выполняющимися на разных вычислительных узлах или ядрах, что существенно ускоряет процесс решения СЛАУ.

## 1. Условия эксперимента

### 1.1 Кластер

- В данном эксперименте используется серверный кластер Oak (oak.cpct.sibsutis.ru), используемый для хранения и обработки данных. Кластер оснащен четырьмя узлами, каждый из которых имеет 24-ядерный процессор Intel Xeon 8664 с частотой 2.2 ГГц, 24 ГБ RAM и сетевую карту QDR InfiniBand с пропускной способностью 40 Гбит/с. Каждая узловая система содержит коммутационную сеть Infiniband Mellanox Switch и сетевой адаптер Mellanox ConnectX-2, обеспечивающий поддержку сетей Ethernet и QDR (High Speed Ethernet, 1000BASE-T) для связи между узлами.

### 1.2 Описание условий эксперимента

- Тип эксперимента: Параллельное решение системы линейных алгебраических уравнений (СЛАУ) методом Якоби в распределенной среде.
- Число процессов: Для эксперимента было выбрано различное количество процессов (от 1 до 32), чтобы оценить влияние увеличения числа процессов на ускорение вычислений.
- Входные данные: Система линейных алгебраических уравнений с размерностями 15.000 и 28.000 переменных, что достаточно для демонстрации эффективности параллельных вычислений методом Якоби.
- Цель эксперимента: Оценить время выполнения параллельной реализации метода Якоби на различных конфигурациях количества процессов и проанализировать ускорение и эффективность использования ресурсов многопроцессорной системы.

## 2. Метод Якоби

2.1 Описание метода

Метод Якоби — это итерационный метод для решения системы линейных алгебраических уравнений вида:

$$Ax = b,$$

где $A$ — квадратная матрица порядка $n$, $b$ — известный вектор-столбец длины $n$, а $x$ — искомый вектор решений. Суть метода заключается в следующем: мы начинаем с некоторого начального приближения $x^{(0)}$ и последовательно уточняем его, пока не достигнем требуемой точности.

Формула для вычисления следующего приближения выглядит следующим образом:

$$x^{(k+1)} = D^{-1}(b + (L + U)x^{(k)}),$$

где:

- $D$ — диагональная часть матрицы $A$,
- $L$ — нижняя треугольная часть матрицы $A$ без диагонали,
- $U$ — верхняя треугольная часть матрицы $A$ без диагонали,
- $x^{(k)}$ — текущее приближение вектора решений.

**Основные шаги:**

1. **Инициализация:**
   - Задать начальное приближение $x^{(0)}$. Часто в качестве начального приближения выбирают нулевой вектор.
   - Установить критерий останова $\epsilon$ (погрешность).

2. **Основной цикл:**

- Вычислить новое приближение $x^{(k+1)}$ по формуле:

$$x^{(k+1)} = D^{-1}(b + (L + U)x^{(k)})$$

- Проверить условие сходимости:

$$||x^{(k+1)} - x^{(k)}|| \leq \epsilon$$

3. **Обновление:**

- Обновить текущее приближение: $x^{(k)} := x^{(k+1)}$ .
- Перейти к следующему шагу основного цикла.

Таким образом, метод Якоби постепенно уточняет решение, пока не будет достигнут необходимый уровень точности.

2.2 Параллельный метод Якоби

Параллельный метод Якоби использует технологию MPI для распределения вычислений между несколькими процессами. Основной принцип заключается в том, что матрица $A$ и вектор $b$ делятся на блоки, которые распределяются между процессами. Каждый процесс отвечает за обработку своего блока данных и обновление соответствующей части вектора решений $x$.

**Основные шаги:**

1. **Инициализация:**

- Процесс с рангом 0 читает исходные данные (матрицу $A$ и вектор $b$) и распределяет их между всеми процессами.
- Остальные процессы получают свои части матрицы и вектора.

2. **Основной цикл:**

- Каждый процесс вычисляет свое частичное обновление вектора $X$ по формуле:

$$x_{local}^{(k+1)} = D_{local}^{-1}(b_{local} + (L_{local} + U_{local})x^{(k)})$$

- Все процессы обмениваются своими частями обновленного вектора $X$ друг с другом, чтобы собрать полный вектор $x^{(k+1)}$.

**3. Проверка сходимости:**

- Один из процессов (обычно процесс с рангом 0) собирает полные векторы $x^{(k)}$ и $x^{(k+1)}$ и проверяет условие сходимости:

$$||x^{(k+1)} - x^{(k)}|| \leq \epsilon$$

## 3. Результаты эксперимента

### 3.1 Время выполнения программы

Эксперименты проводились на системах линейных алгебраических уравнений с размерностями 15.000 и 28.000 переменных при различном числе процессов (узлы • ядра), что позволило оценить эффективность параллельной реализации метода Якоби.

Параметры эксперимента:

- Размеры систем: $n$ = 15.000 и $n$ = 28.000 переменных.
- Число процессов: 1, 8, 16, 32.
- Время выполнения программы при $n$ = 15.000 на 1 процессе: 1.47 с.
- Время выполнения программы при $n$ = 28.000 на 1 процессе: 5.18 с.

Ниже представлено время выполнения программы для разных конфигураций:

Таблица 3.1 – Результаты экспериментов с размерностью 15.000 переменных

| Узлы • Ядра | Общее время работы (с.) | Ускорение |
|---|---|---|
| 2 • 4 | 0.23 | 6.39 |
| 4 • 4 | 0.13 | 11.30 |
| 4 • 8 | 0.17 | 8.64 |

Таблица 3.1.1 – Результаты экспериментов с размерностью 28.000 переменных

| Узлы • Ядра | Общее время работы (с.) | Ускорение |
|---|---|---|
| 2 • 4 | 0.67 | 7.73 |
| 4 • 4 | 0.38 | 13.63 |
| 4 • 8 | 0.21 | 24.66 |

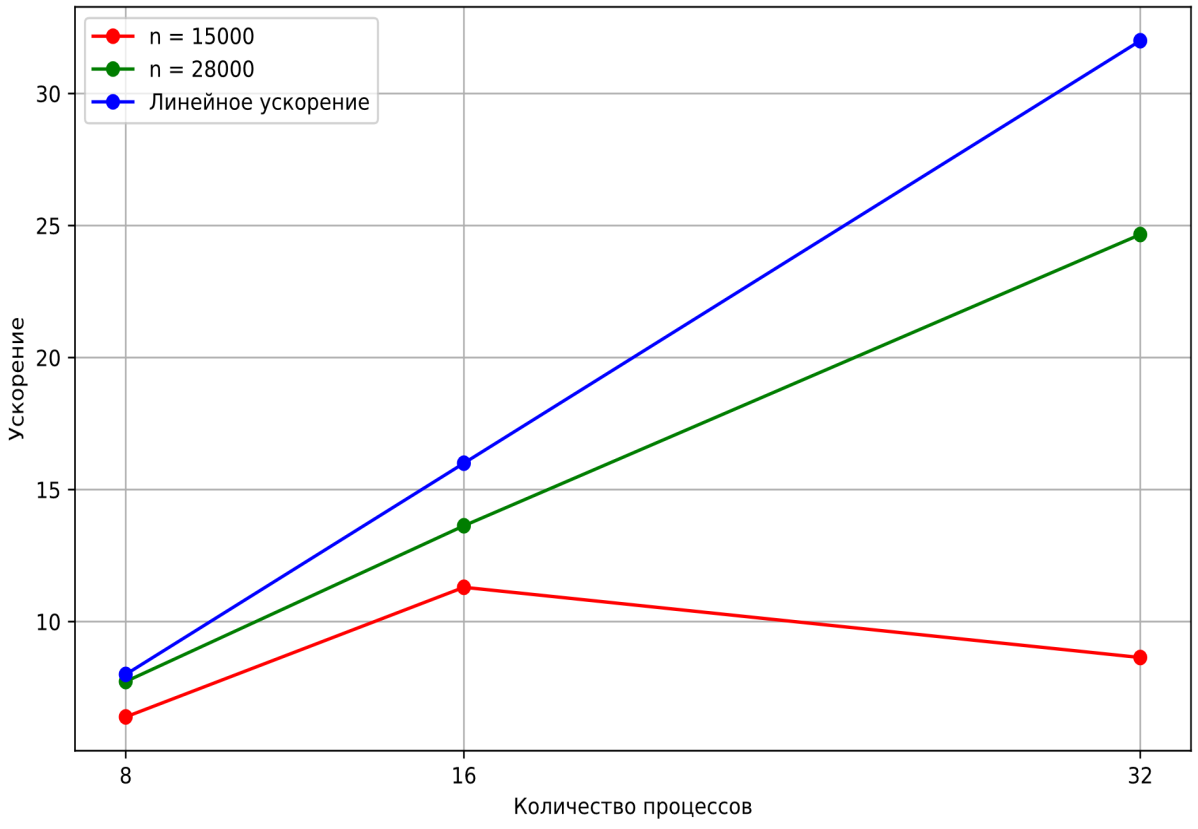


Рис. 3.1 – График масштабируемости.

**Ускорение** рассчитывалось по формуле:

$$S_p = \frac{T_1}{T_p},$$

где $T_1$ – время выполнения программы на одном процессе, $T_p$ – время выполнения на $p$ процессах.

3.2 Анализ результатов

1. **Ускорение:**

Анализ данных показывает, что с увеличением числа процессов ускорение возрастает, но не линейно.

Для задачи с размерностью $n = 15000$ прирост ускорения наблюдается до 16 процессов, но при использовании 32 процессов ускорение уменьшается. Это связано с увеличением накладных расходов на коммуникацию, которые начинают преобладать над вычислительной нагрузкой при относительно небольшом размере задачи.

Для задачи с размерностью $n = 28000$ ускорение значительно увеличивается с ростом числа процессов, достигая максимума на 32 процессах. Это указывает на то, что для больших задач использование большего количества процессов становится более эффективным, так как нагрузка на каждый процесс остается достаточно высокой, чтобы компенсировать накладные расходы.

2. **Общее время работы:**

Видно, что при увеличении числа процессов общее время выполнения программы уменьшается, но не строго пропорционально числу процессов. Это объясняется следующими факторами:

Накладные расходы: увеличение числа процессов приводит к росту накладных расходов на коммуникацию и синхронизацию между процессами. Для небольшой задачи ($n = 15000$) эти расходы оказывают более заметное влияние, что приводит к ухудшению производительности на большем числе процессов.

# ЗАКЛЮЧЕНИЕ

Проведенные эксперименты показали высокую эффективность параллельной реализации метода Якоби для решения систем линейных алгебраических уравнений большой размерности. Увеличение числа процессов привело к значительному сокращению времени выполнения программы, что делает использование параллельных вычислений оправданным и целесообразным при работе с крупными задачами.

Несмотря на наличие некоторых ограничений, связанных с накладными расходами на коммуникацию и синхронизацию процессов, параллельная версия метода Якоби продемонстрировала сублинейное ускорение, что свидетельствует об успешности предложенного подхода. Полученные результаты подтверждают важность применения технологий вроде MPI для решения сложных вычислительных задач в условиях современных высокопроизводительных вычислительных систем.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Голуб, Г.Х., Ван Лоун, Ч.Ф. Матричные вычисления. М.: Мир, 1989.
2. Фостер, И. Проектирование и построение параллельных программ: концепции и инструменты для параллельного программирования. М.: Вильямс, 2000.
3. Pacheco, P.S. An Introduction to Parallel Programming. Morgan Kaufmann Publishers Inc., 2011.
4. Quinn, M.J. Parallel Programming in C with MPI and OpenMP. McGraw-Hill Education, 2003.

# ПРИЛОЖЕНИЕ

jac.c

```
1. #include <string.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <math.h>
5. #include <mpi.h>
6. #include <time.h>
7.
8. #define N 15000
9. //#define N 28000
10.   #define EPSILON 1e-6
11.
12.   // Функция для инициализации матрицы A и вектора b
13.   void initialize_matrix_and_vector(double* A, double* b) {
14.      for (int i = 0; i < N; ++i) {
15.          for (int j = 0; j < N; ++j) {
16.              A[i * N + j] = (i == j) ? 10.0 : (double)(rand() % 10) /
   10.0;
17.          }
18.          b[i] = (double)(rand() % 10) / 10.0;
19.      }
20.   }
21.
22.   // Функция для вычисления одного шага метода Якоби
23.   void jacobi_step(double* local_A, double* local_b, int local_n, int
   start_row, double* global_x, double* new_local_x) {
24.      for (int i = 0; i < local_n; ++i) {
25.          double sum = local_b[i];
26.          for (int j = 0; j < N; ++j) {
27.              if (start_row + i != j) {
28.                  sum -= local_A[i * N + j] * global_x[j];
29.              }
30.          }
31.          if (local_A[i * N + start_row + i] == 0) {
32.              printf("Division by zero on row %d\n", start_row + i);
33.              MPI_Abort(MPI_COMM_WORLD, 1);
34.          }
```

```
35.             new_local_x[i] = sum / local_A[i * N + start_row + i];
36.         }
37.     }
38.
39.     // Функция для проверки сходимости
40.     int check_convergence(double* old_global_x, double* new_global_x, int
    n) {
41.         double max_diff = 0.0;
42.         for (int i = 0; i < n; ++i) {
43.             double diff = fabs(new_global_x[i] - old_global_x[i]);
44.             if (diff > max_diff) {
45.                 max_diff = diff;
46.             }
47.         }
48.         return max_diff <= EPSILON;
49.     }
50.
51.     int main(int argc, char** argv) {
52.         int rank, size;
53.         MPI_Init(&argc, &argv);
54.         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
55.         MPI_Comm_size(MPI_COMM_WORLD, &size);
56.
57.         double *A = NULL, *b = NULL;
58.         double *local_A = NULL, *local_b = NULL;
59.         double *global_x = NULL, *new_local_x = NULL;
60.
61.         // Рассчитать количество строк у каждого процесса
62.         int rows_per_proc = N / size;
63.         int remainder = N % size;
64.         int local_n = rows_per_proc + (rank < remainder ? 1 : 0);
65.         int start_row = rank * rows_per_proc + (rank < remainder ? rank :
    remainder);
66.
67.         // Инициализация данных на процессе 0
68.         if (rank == 0) {
69.             A = (double*)malloc(N * N * sizeof(double));
70.             b = (double*)malloc(N * sizeof(double));
71.             initialize_matrix_and_vector(A, b);
```

```
72.      }
73.
74.      // Распределить данные между процессами
75.      local_A = (double*)malloc(local_n * N * sizeof(double));
76.      local_b = (double*)malloc(local_n * sizeof(double));
77.      global_x = (double*)malloc(N * sizeof(double));
78.      new_local_x = (double*)malloc(local_n * sizeof(double));
79.
80.      int *sendcounts = (int*)malloc(size * sizeof(int));
81.      int *displs = (int*)malloc(size * sizeof(int));
82.
83.      for (int i = 0; i < size; i++) {
84.          sendcounts[i] = (rows_per_proc + (i < remainder ? 1 : 0)) * N;
85.          displs[i] = i * rows_per_proc * N + (i < remainder ? i * N :
   remainder * N);
86.      }
87.
88.      MPI_Scatterv(A, sendcounts, displs, MPI_DOUBLE, local_A, local_n *
   N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
89.
90.      for (int i = 0; i < size; i++) {
91.          sendcounts[i] = rows_per_proc + (i < remainder ? 1 : 0);
92.          displs[i] = i * rows_per_proc + (i < remainder ? i : remainder);
93.      }
94.
95.      MPI_Scatterv(b, sendcounts, displs, MPI_DOUBLE, local_b, local_n,
   MPI_DOUBLE, 0, MPI_COMM_WORLD);
96.
97.      memset(global_x, 0, N * sizeof(double)); // Начальное приближение
98.
99.      double start_time = 0.0, end_time = 0.0;
100.     if (rank == 0) {
101.         start_time = MPI_Wtime();
102.     }
103.
104.     int converged = 0, iteration = 0;
105.
106.     while (!converged && iteration < 10000) {
```

```
107.         jacobi_step(local_A, local_b, local_n, start_row, global_x,
   new_local_x);
108.
109.         MPI_Allgatherv(new_local_x, local_n, MPI_DOUBLE, global_x,
   sendcounts, displs, MPI_DOUBLE, MPI_COMM_WORLD);
110.
111.         if (rank == 0) {
112.             converged = check_convergence(global_x, global_x, N);
113.         }
114.
115.         MPI_Bcast(&converged, 1, MPI_INT, 0, MPI_COMM_WORLD);
116.
117.         iteration++;
118.     }
119.
120.     if (rank == 0) {
121.         end_time = MPI_Wtime();
122.         printf("Время выполнения: %.6f секунд\n", end_time -
   start_time);
123.     }
124.
125.     free(A); free(b);
126.     free(local_A); free(local_b);
127.     free(global_x); free(new_local_x);
128.     free(sendcounts); free(displs);
129.
130.     MPI_Finalize();
131.     return 0;
132. }
```