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 вычисления обратной матрицы методом Гаусса

Выполнил студент Григорьев Юрий Вадимович
Ф.И.О.

Группы ИС-142

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

Защищена _____ Оценка _____

Новосибирск – 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является разработка и реализация параллельной MPI-программы для вычисления обратной матрицы с использованием метода Гаусса. Обратной матрицей называют такую матрицу, при умножении которой на исходную в качестве результата получается единичная диагональная матрица. Обратные матрицы существуют только для квадратных и невырожденных (определители которых не равны нулю) матриц.

Метод Гаусса (Гаусса-Жордана) был выбран за его надёжность и эффективность в вычислении обратных матриц. Этот метод основан на последовательных преобразованиях строк исходной матрицы для приведения её к диагональному или единичному виду, что позволяет легко получить обратную матрицу.

В работе будут рассмотрены теоретические основы метода Гаусса для вычисления обратной матрицы, детали реализации данного метода в программном виде, а также подробное описание работы параллельной программы, включая инициализацию процессов, обмены информацией между процессами, вычислительный процесс и сохранение результата.

1 Вычисление обратной матрицы методом Гаусса

Метод Гаусса-Жордана для нахождения обратной матрицы основывается на элементарных преобразованиях строк матрицы. Цель - привести данную квадратную матрицу к единичной форме. Параллельно с этими преобразованиями, те же операции применяются к единичной матрице того же размера. В итоге, после приведения исходной матрицы к единичной, единичная матрица преобразуется в обратную к исходной.

Этапы Метода Гаусса-Жордана

1. **Формирование Расширенной Матрицы**

Исходная матрица A объединяется с единичной матрицей I того же размера. Это создает расширенную матрицу [A|I][A|I], где I - единичная матрица.

2. **Прямой Ход**

Строки расширенной матрицы преобразуются таким образом, чтобы привести левую часть (исходную матрицу A) к верхней треугольной форме. Это достигается путём элементарных преобразований строк: перестановка строк, умножение строки на ненулевой коэффициент и добавление к строке другой строки, умноженной на коэффициент.

3. **Обратный Ход**

После получения верхней треугольной матрицы начинается процесс обратного хода, целью которого является приведение левой части матрицы к диагональному виду, а затем к единичной матрице. Это достигается путем

дополнительных элементарных преобразований строк.

4. **Получение Обратной Матрицы**: По завершении этих преобразований, правая часть (матрица B) расширенной матрицы [I|B] представляет собой матрицу, обратную исходной.

В процессе преобразований крайне важно избегать деления на ноль, поэтому при выборе ведущего элемента в строке (для нормализации строки) следует выбирать максимальный по модулю элемент. Если в какой-то момент ведущий элемент (элемент на диагонали) оказывается равным нулю, это указывает на то, что матрица не обратима.

2 Реализация метода Гаусса

Пояснения по алгоритму работы MPI-программы (Приложение 2)

2.1 Инициализация и распределение данных

Функция `get_chunk` на основе общего количества процессов и номере текущего процесса определяет диапазон строк исходной матрицы (верхнюю и нижнюю границы), который будет обработан каждым процессом. Она обеспечивает равномерное распределение строк между процессами, учитывая, что общее количество строк может не делиться нацело на количество процессов.

Функция `get_proc` на основе заданного номера строки определяет, какой процесс отвечает за ее обработку.

Функция `get_input_matrix` генерирует исходную матрицу, которая будет преобразована. Функция `get_connected_matrix` создает сопутствующую единичную матрицу. В процессе вычислений эта матрица будет преобразована в обратную к исходной.

2.2 Ход вычислений и обмен информацией между процессами

Функция `inverse_matrix` обрабатывает один столбец (`cur_col`) матрицы A на каждом шаге алгоритма Гаусса. Целью этой функции является преобразование исходной матрицы таким образом, чтобы в текущем столбце все элементы, кроме главного (диагонального), стали нулями. Те же операции применяются и к единичной матрице X, которая в результате преобразуется в обратную матрицу к A.

Шаги Функции `inverse_matrix`:

1. Поиск Локального Максимума в Столбце

Для текущего столбца `cur_col` каждый процесс ищет локальный максимальный элемент (по модулю) в своей части матрицы. Локальный максимум и его индекс сохраняются в `local_max` и `local_index`.

2. Определение глобального максимума

Используя операцию `MPI_Allreduce`, локальные максимумы со всех процессов собираются и сравниваются для нахождения глобального максимума. Глобальный максимум (`global_data.value`) и его индекс (`global_data.index`) используются для определения строки с главным элементом в дальнейших вычислениях.

3. Проверка на необратимость

Если глобальный максимум (`global_data.value`) равен нулю, функция `inverse_matrix` возвращает `false`, что означает, что матрица вырождена, а значит не имеет обратной. Это поведение обрабатывается в функции `main` с возвратом кода ошибки и выводом сообщения об ошибке пользователю.

4. Обмен данными и нормализация строк

При наличии глобального максимума создается массив `s1`, куда процессы `diag_p` и `main_p` (процесс с диагональным элементом текущего столбца и процесс с главным элементом) копируют соответствующие строки из A и X. Применяется `MPI_Allreduce` для объединения данных из `s1` в `s2` на всех процессах. Он обеспечивает, что все процессы имеют одинаковую информацию

о нормализованной строке и строке с главным элементом. Осуществляется нормализация строки с главным элементом так, чтобы главный элемент в ней (ныне диагональный) стал равен 1.

5. Перестановка и вычитание строк

Если ранг текущего процесса совпадает с `main_p` или `diag_p`, то происходит обновление соответствующих строк в A и X. Все процессы вычитают нормализованную строку из остальных своих строк, чтобы обнулить элементы в текущем столбце, кроме диагонального.

2.3 Завершение вычислений и сбор результатов

По завершении преобразований каждый процесс содержит часть обратной матрицы. Для получения полной обратной матрицы, происходит сбор в процессе 0 функцией MPI_Allgatherv, в которую подаются заранее подготовленные каждым процессом массивы `recvcounts` и `displs` с количеством отправляемых элементов и их смещением в полной матрице соответственно.

Выполняется завершение отсчета времени выполнения программы, пользователю выводится информация о работе. Освобождается память выделенных массивов, финализируется работа MPI функцией `MPI_Finalize`.

3 Масштабируемость реализованной программы

3.1 Результаты запусков на кластере Oak

Таблица 1 – Результаты экспериментов

Количество процессов	Время работы программы для размера матрицы N x N	
	N = 1500	N = 5000
1 (послед.)	76,4306	1878,501
2 (2 x 1)	38,275	958,965
4 (2 x 2)	19,1628	471,292
6 (2 x 3)	12,9734	315,931
8 (2 x 4)	9,91196	238,076
10 (2 x 5)	8,53609	196,28
12 (2 x 6)	7,23194	160,478
14 (2 x 7)	6,63812	138,709
16 (2 x 8)	6,19978	125,993

Таблица 2 – Ускорение

Количество процессов	Ускорение работы программы для размера матрицы N x N относительно последовательной программы (Приложение 1)	
	N = 1000	N = 10000
2 (2 x 1)	1,99688047	1,958883797
4 (2 x 2)	3,988488112	3,985853781
6 (2 x 3)	5,891331494	5,945921736
8 (2 x 4)	7,710947179	7,89034174
10 (2 x 5)	8,953818434	9,570516609
12 (2 x 6)	10,56847817	11,70566059

14 (2 x 7)	11,51389249	13,54274777
16 (2 x 8)	12,32795357	14,9095664

3.2 Построение результирующего графика

С помощью Python и библиотеки matplotlib была создана программа (Приложение 3) для построения графика по данным из Таблицы 2.

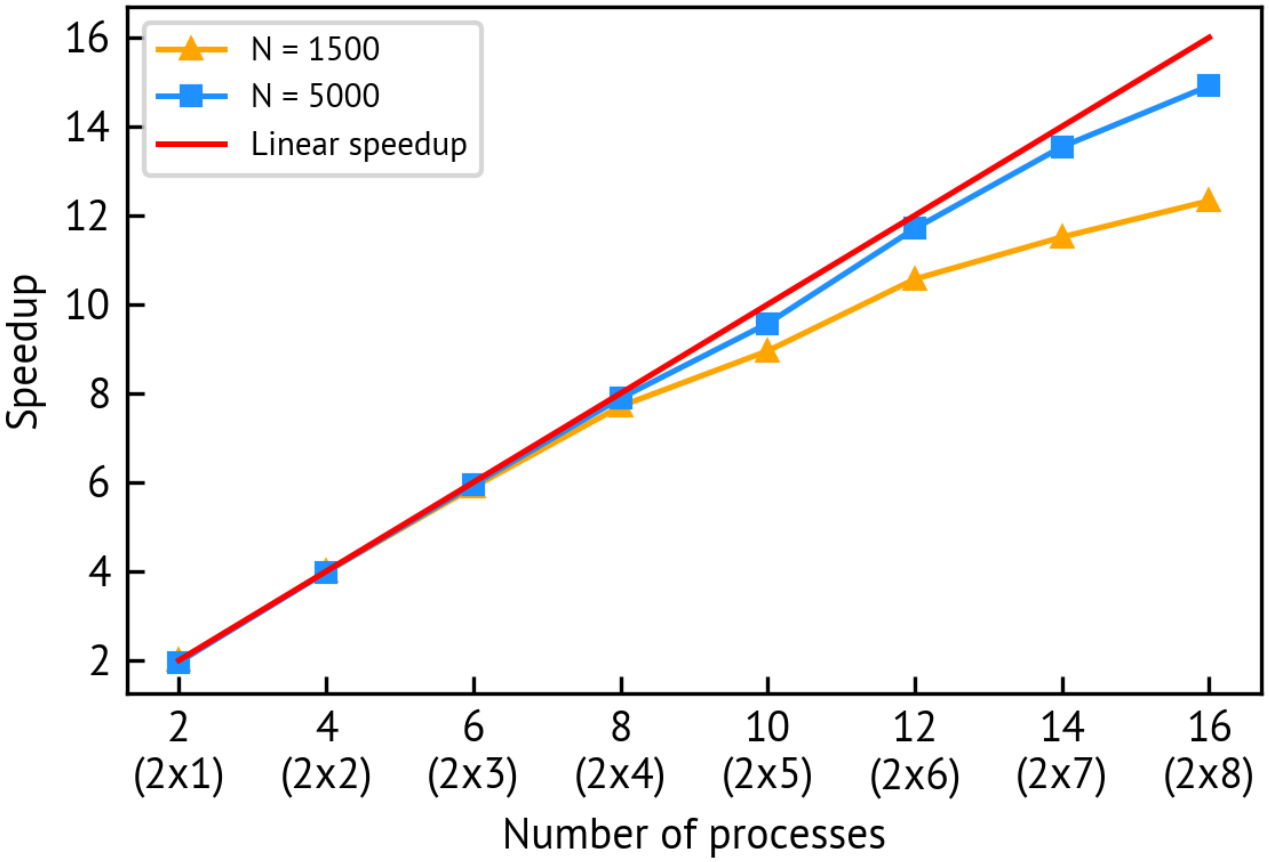

Рисунок 1 - График зависимости ускорения выполнения от количества процессов MPI-программы относительно последовательной версии

ЗАКЛЮЧЕНИЕ

В работе была успешно реализована параллельная программа для вычисления обратной матрицы с использованием метода Гаусса-Жордана и технологии MPI. Программа демонстрирует эффективное распределение вычислительных задач между процессами и хорошую масштабируемость при увеличении работающих процессов, что позволяет значительно ускорить процесс вычисления обратной матрицы.

Тестирование программы на различных размерах матриц показало значительное улучшение производительности по сравнению с последовательными методами. Это подчёркивает преимущества использования параллельных вычислений в задачах, требующих интенсивных вычислений и обработки больших данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем

2. Гергель В.П. Теория и практика параллельных вычислений

3. Karniadakis G., Kirby R. Parallel Scientific Computing in C++ and MPI

4. Эндрюс Г. Основы многопоточного, параллельного и распределенного программирования

ПРИЛОЖЕНИЕ

1 Исходный код последовательной программы

```
#include <cstdlib>
#include <iostream>
#include <sys/time.h>
#include <vector>

using namespace std;

int n;

double wtime() {
  struct timeval t;
  gettimeofday(&t, NULL);
  return (double)t.tv_sec + (double)t.tv_usec * 1E-6;
}

bool invertMatrix(double *matrix) {
  double *temp = (double*)malloc(n * n * 2 * sizeof(double));
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      temp[i * n + j] = matrix[i * n + j];
    }
    temp[i * n + i + n] = 1;
  }

  // direct way
  for (int i = 0; i < n; ++i) {
    // row normalization
    double diag = temp[i * n + i];
    if (diag == 0) return false; // no inverse matrix
    for (int j = 0; j < n * 2; ++j) {
      temp[i * n + j] /= diag;
    }
    // col zeroing
    for (int k = 0; k < n; ++k) {
      if (k == i) continue;
      double factor = temp[k * n + i];
      for (int j = 0; j < n * 2; ++j) {
        temp[k * n + j] -= factor * temp[i * n + j];
      }
    }
  }

  // getting result
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      matrix[i * n + j] = temp[i * n + j + n];
    }
  }
  return true;
}
```

```
double *get_matrix() {
  double *a = (double*)malloc(n * n * sizeof(double));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      a[i * n + j] = min(n - j, n - i);
    }
  }
  return a;
}

int main(int argc, char **argv) {
  double t = -wtime();
  if (argc > 1) {
    n = atoi(argv[1]);
  } else {
    n = 1680;
  }
  double *matrix = get_matrix();
  if (invertMatrix(matrix)) {
    t += wtime();
    cout << "n = " << n << ", t = " << t << " sec\n";
  } else {
    cout << "No inverse matrix\n";
  }
  free(matrix);
  return 0;
}
```

2 Исходный код MPI-программы

```
#include <mpi.h>

#include <cmath>
#include <cstdlib>
#include <iomanip>
#include <iostream>

// Global variables
int rank, commsize, lb, ub, nrows, n;

void get_chunk(int *l, int *u) {
  int rows_per_process = n / commsize;
  int remaining_rows = n % commsize;
  if (rank < remaining_rows) {
    *l = rank * (rows_per_process + 1);
    *u = *l + rows_per_process;
  } else {
    *l = remaining_rows * (rows_per_process + 1) + (rank - remaining_rows) *
rows_per_process;
    *u = *l + rows_per_process - 1;
  }
}

int get_proc(int idx) {
```

```
  int rows_per_process = n / commsize;
  int remaining_rows = n % commsize;
  int threshold = remaining_rows * (rows_per_process + 1);
  if (idx < threshold) {
    // Index falls in the range of processes with an extra row
    return idx / (rows_per_process + 1);
  } else {
    // Index falls in the range of processes without an extra row
    return remaining_rows + (idx - threshold) / rows_per_process;
  }
}

double *get_input_matrix() {
  double *matrix = (double*)malloc(nrows * n * sizeof(double));
  for (int i = 0; i < nrows; i++) {
    for (int j = 0; j < n; j++) {
      matrix[i * n + j] = std::min(n - j, n - i - rank * nrows);
    }
  }
  return matrix;
}

double *get_connected_matrix() {
  double *x = (double*)malloc(nrows * n * sizeof(double));
  for (int i = 0; i < nrows; i++) {
    for (int j = 0; j < n; j++) {
      if (i + rank * nrows == j) {
        x[i * n + j] = 1.0;
      } else {
        x[i * n + j] = 0.0;
      }
    }
  }
  return x;
}

// Function for zeroing cur_col in matrix A & leaving diagonal with 1
// All operations are duplicated to matrix X
bool inverse_matrix(double *a, double *x, int cur_col) {
  // Getting local maximum in cur_col
  double local_max = 0.0;
  int local_index = -1;
  for (int i = 0; i < nrows; i++) {
    int global_index = i + lb;
    if (global_index >= cur_col) {
      double value = std::fabs(a[i * n + cur_col]);
      if (value > local_max) {
        local_max = value;
        local_index = global_index;
      }
    }
  }

  // Struct for sending (local) & receiving (global) maximums from processes
  struct {
    double value;
    int index;
```

```
  } local_data = {local_max, local_index}, global_data;

  // Getting global maximum
  MPI_Allreduce(&local_data, &global_data, 1, MPI_DOUBLE_INT, MPI_MAXLOC,
MPI_COMM_WORLD);

  // If global cur_col maximum is 0, matrix is singular and non-invertible
  if (global_data.value == 0.0) {
    return false;
  }

  // Calculating processes with diagonalised col & main element
  int diag_p = get_proc(cur_col);
  int main_p = get_proc(global_data.index);

  // Create & fill array for transferring data
  double *s1 = (double*)malloc(n * 4 * sizeof(double));
  for (int i = 0; i < n * 4; i++) {
    s1[i] = 0.0;
  }

  // Process diag_p puts needed rows to s1
  if (rank == diag_p) {
    for (int i = n; i < n * 2; ++i) {
      s1[i] = a[(cur_col - rank * nrows) * n + i - n];
      s1[i + n * 2] = x[(cur_col - rank * nrows) * n + i - n];
    }
  }

  // Process main_p puts needed rows to s1
  if (rank == main_p) {
    for (int i = 0; i < n; ++i) {
      s1[i] = a[(global_data.index - rank * nrows) * n + i];
      s1[i + n * 2] = x[(global_data.index - rank * nrows) * n + i];
    }
  }

  // Reduce s1 arrays to s2
  double *s2 = (double*)malloc(n * 4 * sizeof(double));
  MPI_Allreduce(s1, s2, n * 4, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  // All processes normalise row with main elem
  double c = s2[cur_col];
  for (int i = 0; i < n; i++) {
    s2[i] = s2[i] / c;
    s2[i + n * 2] = s2[i + n * 2] / c;
  }

  // Processes with main and diagonal elements swap rows
  if (rank == main_p) {
    for (int i = n; i < n * 2; i++) {
      a[(global_data.index - rank * nrows) * n + i - n] = s2[i];
      x[(global_data.index - rank * nrows) * n + i - n] = s2[i + n * 2];
    }
  }

  if (rank == diag_p) {
```

```
      for (int i = 0; i < n; i++) {
        a[(cur_col - rank * nrows) * n + i] = s2[i];
        x[(cur_col - rank * nrows) * n + i] = s2[i + n * 2];
      }
    }

    // All processes subtract diagonal row from their rows, zeroing cur_col
    // (except diagonal)
    for (int i = 0; i < nrows; i++) {
      if (i + rank * nrows != cur_col) {
        c = a[i * n + cur_col];
        for (int j = 0; j < n; j++) {
          a[i * n + j] = a[i * n + j] - s2[j] * c;
          x[i * n + j] = x[i * n + j] - s2[j + n * 2] * c;
        }
      }
    }

    // Free up resources used for transferring data
    free(s1);
    free(s2);

    return true;
}

int main(int argc, char **argv) {
    // Start measuring time
    double t = -MPI_Wtime();

    // Initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    n = 6;
    if (argc > 1) {
      n = std::atoi(argv[1]);
    }

    // Get lower & upper bounds for this process, calculate number of local rows
    get_chunk(&lb, &ub);
    nrows = ub - lb + 1;

    // Get matrices A and I (X after performing inversion operations)
    double *a = get_input_matrix();
    double *x = get_connected_matrix();

    // Perform operations with i-th col
    for (int i = 0; i < n; ++i) {
      if (!inverse_matrix(a, x, i)) {
        std::cerr << "No inverse matrix\n";
        return 1;
      }
    }

    // Fill receive counts and displacements for all procs
    double *recvbuf = nullptr;
```

```
  int *recvcounts = (int*)malloc(commsize * sizeof(int));
  int *displs = (int*)malloc(commsize * sizeof(int));
  if (rank == 0) {
    recvbuf = (double*)malloc(n * n * sizeof(double));
    for (int i = 0; i < commsize; ++i) {
      int l, u;
      get_chunk(&l, &u);
      recvcounts[i] = (u - l + 1) * n;
      displs[i] = l * n;
    }
  }

  // Gathering all parts of inverse matrix on proc 0
  MPI_Gatherv(a, nrows * n, MPI_DOUBLE, recvbuf, recvcounts, displs, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

  // End measuring time
  t += MPI_Wtime();

  // Here receive buffer on proc 0 contains entire inverse matrix

  if (rank == 0) {
    std::cout << commsize << " procs, n = " << n << ", t = " << t << " sec\n";
  }

  // Free up allocated memory, finalize
  free(a);
  free(x);
  free(recvcounts);
  free(displs);
  if (rank == 0) {
    free(recvbuf);
  }
  MPI_Finalize();
  return 0;
}
```

3 Исходный код Python-программы для построения графика

```
#!/usr/bin/env python3
import matplotlib.pyplot as plt
import numpy as np
def draw(filenames, labels, dest_filename):
    plt.rcParams["legend.markerscale"] = 1.0
    plt.rcParams['font.family'] = 'sans-serif'
    plt.rcParams['font.sans-serif'] = ['PT Sans']
    plt.rcParams['font.size'] = '9'
    plt.rcParams["legend.loc"] = "upper left"
    plt.rcParams["legend.fontsize"] = '7'
    cm = 1 / 2.54 # centimeters in inches
    fig = plt.figure(figsize = (10 * cm, 7 * cm))
    ax = fig.add_subplot(111)
    ax.set_title("")
    ax.set(xlabel = "Number of processes", ylabel = "Speedup")
```

```
    ax.label_outer()
    ax.xaxis.set_ticks(np.arange(0, 17, 2))
    ax.xaxis.set_tick_params(direction='in', which='both')
    ax.yaxis.set_tick_params(direction='in', which='both')
    for (fname, datalabel) in zip(filenames, labels):
        data = np.loadtxt(fname)
        x = data[:, 0]
        y = data[:, 1]
        if datalabel == "N = 1500":
            marker = '-^'
            color = "orange"
        elif datalabel == "N = 5000":
            marker = '-s'
            color = "dodgerblue"
        else:
            marker = '-'
            color = "red"
        ax.plot(x, y, marker, c = color, markersize = 4.0, linewidth = 1.2, label =
datalabel)
    labels = [item.get_text() for item in ax.get_xticklabels()]
    labels[1] = '2\n(2x1)'
    labels[2] = '4\n(2x2)'
    labels[3] = '6\n(2x3)'
    labels[4] = '8\n(2x4)'
    labels[5] = '10\n(2x5)'
    labels[6] = '12\n(2x6)'
    labels[7] = '14\n(2x7)'
    labels[8] = '16\n(2x8)'
    ax.set_xticklabels(labels)
    plt.tight_layout()
    ax.legend()
    fig.savefig(dest_filename, dpi = 300)

if __name__ == "__main__":
    draw(["gauss_1500.dat", "gauss_5000.dat", "linear.dat"], ["N = 1500", "N = 5000",
"Linear speedup"], "chart.png")
```