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 определителя методом Гаусса

Выполнил студент Никулин Максим Кириллович
Ф.И.О.

Группы ИС-242

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

Защищена _____ Оценка _____

Новосибирск – 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработать параллельную MPI-программу для вычисления определителя матрицы методом Гаусса. Программа должна эффективно использовать ресурсы многопроцессорной или кластерной системы, обеспечивая распределение вычислений между процессами и минимизируя затраты на обмен данными.

ПОСЛЕДОВАТЕЛЬНАЯ ВЕРСИЯ АЛГОРИТМА

1.1. Теоретическая часть

Метод Гаусса (Gaussian elimination, row reduction) – метод последовательного исключения переменных.

Основные этапы метода:

· Прямой ход — элементы матрицы последовательно модифицируются и приводятся к треугольной форме путем элементарных преобразований.

· Обратный ход (опционально, если требуется решить СЛАУ) — вычисление неизвестных с использованием треугольной матрицы.

В рассматриваемой реализации используется метод прямого хода для вычисления определителя матрицы. После приведения к треугольному виду определитель равен произведению элементов на главной диагонали.

Теоретическая сложность:

Прямой ход: $O(n^3)$, так как на каждой итерации k преобразуется оставшиеся элементы подматрицы размером $(n - k)$ $(n - k)$.

1.2. Описание алгоритма

1. Инициализация матрицы:

· Матрица коэффициентов A размера $n \times n$ заполняется случайными числами от 1 до 1000.

· Матрица хранится в памяти как одномерный массив, для доступа используется адресация $a\,[\, i \times n + j\,]$, где i – номер строки, j – номер столбца.

2. Прямой ход:

· Выбирается опорный элемент $a\,[\, k \times n + k\,]$.

· Вычисляется множитель lik = $\frac{a[i \times n + k]}{a[k \times n + k]}$

· Все элементы строки i корректируются по формуле

$$a[i \times n + j] = a[i \times n + j] - lik \cdot a[k \times n + j]$$

· После завершения цикла, элементы ниже диагонали обнуляются.

3. Вычисление определителя

· Матрица преобразована к треугольному виду, остается посчитать произведение диагональных элементов.

ПАРАЛЛЕЛЬНАЯ ВЕРСИЯ АЛГОРИТМА

2.1. Принцип работы алгоритма

Метод Гаусса применяется для решения систем линейных уравнений путем приведения матрицы к треугольному виду (прямой ход) с последующим обратным ходом для вычисления неизвестных. В данной реализации рассматривается только прямой ход для вычисления определителя матрицы. Алгоритм был адаптирован для работы в параллельной среде с использованием MPI.

Этапы параллельной версии алгоритма:

1. Разбиение матрицы между процессами:

· Матрица делится по строкам на фрагменты, которые обрабатываются процессами. Каждый процесс получает определенное количество строк, вычисляемое функцией get_chunk ().

2. Инициализация данных

· Каждый процесс генерирует строки своей подматрицы независимо. Для этого используется генератор случайных чисел, с учетом индекса строки для воспроизводимости.

3. Прямой ход:

· Если строка находится в памяти текущего процесса, она нормализуется и рассылается другим процессам с помощью MPI_Bcast. В противном случае процесс принимает нормализованную строку от владельца.

· Каждый процесс обрабатывает только свои строки, используя принятые данные для вычисления множителя scaling и корректировки элементов строк, чтобы обнулить значения в текущем столбце.

4. Вычисление определителя:

· Каждый процесс вычисляет произведение диагональных элементов своей подматрицы.

· Затем каждый вычисленный определитель передается корневому процессу через операцию MPI_Reduce и перемножаются между собой.

2.2. Преимущества параллельной версии

1. Сокращение времени выполнения:

Распараллеливание позволяет разделить работу между процессами, что ускоряет выполнение алгоритма по сравнению с последовательной реализацией.

2. Эффективное использование ресурсов:

Алгоритм хорошо масштабируется на системах с большим числом вычислительных узлов, если матрица достаточно большая, чтобы оправдать накладные расходы на коммуникацию.

3. Универсальность:

Реализация основана на стандартной библиотеке MPI, что делает алгоритм совместимым с различными параллельными вычислительными системами.

РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

3.1. Тестирование параллельной программы

В ходе тестирования параллельной версии алгоритма Гаусса были проведены эксперименты с различными размерами матриц и числами процессов. Основной целью тестов было измерение времени выполнения программы, и оценка ускорения параллельной реализации.

Для проведения замеров производительности параллельной программы использовался вычислительный Oak. Основные характеристики кластера:

· Процессоры: 4 узла x86-64: 2 ×Intel Xeon Quad E5620, RAM 24 GB.

· Сеть: InfiniBand QDR (HCA Mellanox MT26428, switch Mellanox InfiniScale IV IS5030 QDR 36-Port), управляющая сеть: Gigabit Ethernet.

· MPI: реализация библиотеки OpenMPI версии 4.1.4.

· Компилятор: GCC версии 9.4.0 с оптимизациями флагов −O2

Параметры эксперимента:

· Размеры матриц n = 3000, n = 5000.

· Число процессов 8, 16, 24, 32.

· Кластер с 4 вычислительными узлами, на каждом из которых запускалось по 2, 4, 6, 8 MPI-процесса.

· Ускорение считается по формуле:

$$S = \frac{T_{serial}}{T}$$

Таблица 3.1 – Результаты экспериментов.

Размер матрицы (n)	Число процессов (p)	Время последовательной программы (с.)	Время параллельной программы (с.)	Ускорение

3000	8	70.8	9.2	7.7
	16		4.9	14.5
	24		3.4	20.7
	32		2.7	26.6
5000	8	328.5	42.2	7.8
	16		21.6	15.2
	24		14.6	22.4
	32		11.8	27.96

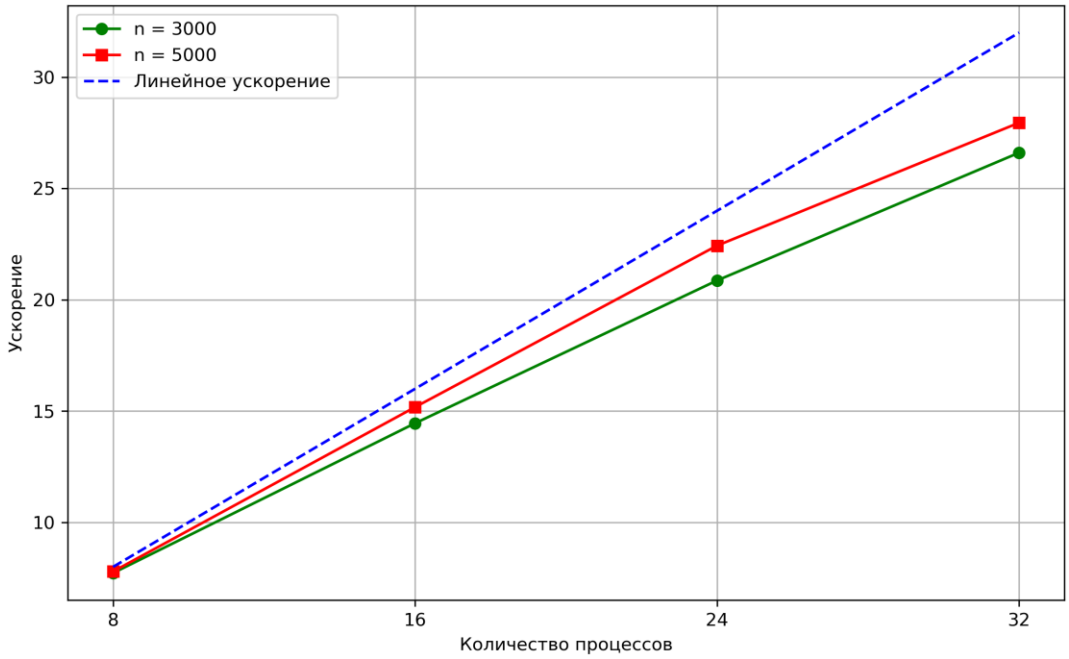

Рис. 3.1 – График масштабируемости.

3.2. Влияние числа процессов на производительность программы.

В ходе экспериментов была проанализирована производительность параллельной программы при различных числах процессов и размерах матриц. Основное внимание уделялось тому, как увеличение числа процессов влияет на ускорение и общую эффективность вычислений.

Наблюдения:

1. Скорость выполнения: Увеличение числа процессов приводит к значительному сокращению времени выполнения программы.

2. Ускорение: Ускорение программы возрастает с увеличением числа процессов, но не линейно. Это связано с ростом накладных расходов на коммуникацию между узлами.

3. Эффективность: Эффективность вычислений уменьшается с ростом числа процессов. При $n = 3000$ эффективность падает с 0.96 при 8 процессах до 0.83 при 32 процессах, что обусловлено увеличением доли времени, затрачиваемого на обмен данными между узлами. Однако для больших матриц ($n = 5000$) эффективность остается выше, поскольку вычислительная нагрузка перевешивает накладные расходы.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был разработан и исследован параллельный алгоритм для вычисления определителя матрицы методом Гаусса. Программа реализована с использованием библиотеки MPI и предназначена для эффективного использования ресурсов многопроцессорной или кластерной системы.

Осуществлено моделирование работы разработанного алгоритма на различных конфигурациях кластера и размерах матриц. Показано, что программа обеспечивает распределение вычислений между процессами и демонстрирует значительное ускорение по сравнению с последовательной версией, особенно при увеличении размера матрицы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хван И.И., Мельников В.П. Параллельные алгоритмы и их реализация. – М.: МГТУ им. Н.Э. Баумана, 2015. – 432 с.

2. Парамонов П.В., Коротеев А.И. MPI и параллельные вычисления. – Новосибирск: Сибирское научное издательство, 2010. – 312 с.

3. Гроуп У. Использование и оптимизация MPI в высокопроизводительных системах // Журнал вычислительных систем. – 2018. – Т. 29, № 3. – С. 45–58.

4. Thakur R., Gropp W., Lusk E. An Abstract Device Interface for Implementing Portable Parallel I/O Interfaces // Proc. of the 6th Symposium on the Frontiers of Massively Parallel Computation. – Annapolis, USA, 1996. – P. 180–187.

5. Rabenseifner R., Hoefler T., Gropp W. Optimization of MPI Communication for Large-Scale Systems // Journal of High-Performance Computing Applications. – 2016. – Vol. 30, No. 4. – P. 394–408.

ПРИЛОЖЕНИЕ

parallel.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#include <mpi.h>

int get_chunk(int n, int commsize, int rank)
{
    int q = n / commsize;
    if (n % commsize)
        q++;
    int r = commsize * q - n;
    /* Compute chunk size for the process */
    int chunk = q;
    if (rank >= commsize - r)
        chunk = q - 1;
    return chunk;
}

int main(int argc, char *argv[])
{
    int n = 3000;
    int rank, commsize;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &commsize);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int nrows = get_chunk(n, commsize, rank);
    int *rows = malloc(sizeof(*rows) * nrows);
    double *a = malloc(sizeof(*a) * n * nrows);
    double *tmp = malloc(sizeof(*tmp) * n);

    for (int i = 0; i < nrows; i++)
    {
        rows[i] = rank + commsize * i;
        srand(rows[i] * (n + 1));
        for (int j = 0; j < n; j++)
            a[i * n + j] = rand() % 1000 + 1;
    }

    double t = MPI_Wtime();
    int row = 0;
    for (int i = 0; i < n - 1; i++)
    {
        // Исключаем x_i
        if (i == rows[row])
        {
            MPI_Bcast(&a[row * n], n, MPI_DOUBLE, rank, MPI_COMM_WORLD);
            for (int j = 0; j < n; j++)
                tmp[j] = a[row * n + j];
            row++;
        }
        else
        {
            MPI_Bcast(tmp, n, MPI_DOUBLE, i % commsize, MPI_COMM_WORLD);
        }

        for (int j = row; j < nrows; j++)
        {
            double scaling = a[j * n + i] / tmp[i];
```

```
            for (int k = i; k < n; k++)
                a[j * n + k] -= scaling * tmp[k];
        }
    }

    double local_det = 1.0;
    for (int i = 0; i < nrows; i++)
    {
        local_det *= a[rows[i]];
    }

    double global_det = 1.0;
    MPI_Reduce(&local_det, &global_det, 1, MPI_DOUBLE, MPI_PROD, 0,
MPI_COMM_WORLD);
    t = MPI_Wtime() - t;
    free(tmp);
    free(rows);
    free(a);

    if (rank == 0)
    {
        printf("Gaussian Elimination (MPI): n %d, time (sec) %.6f\n", n,
t);
        printf("Speedup :%lf \n", ((328.534671) / t));
    }

    MPI_Finalize();
    return 0;
  }
```

Serial.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>

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

int main()
{
    int n = 3000;
    double t = wtime();
    double *a = malloc(sizeof(*a) * n * n);

    for (int i = 0; i < n; i++)
    {
        srand(i * (n + 1));
        for (int j = 0; j < n; j++)
        {
            a[i * n + j] = rand() % 1000 + 1;
        }
    }

    double det = 1.0;

    for (int k = 0; k < n - 1; k++)
    {
        double pivot = a[k * n + k];
```

```
        for (int i = k + 1; i < n; i++)
        {
            double lik = a[i * n + k] / pivot;
            for (int j = k; j < n; j++)
            {
                a[i * n + j] -= lik * a[k * n + j];
            }
        }
    }

    for (int i = 0; i < n; i++)
    {
        det *= a[i * n + i];
    }

    t = wtime() - t;

    printf("%lf\n",det);
    printf("Gaussian Elimination (serial): n %d, time (sec) %.6f\n", n,
t);

    free(a);
    return 0;
}
```