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 реализации алгоритма Флойда

Выполнил студент Рудцких Владислав Евгеньевич
Ф.И.О.

Группы ИВ-222

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

Защищена _____ Оценка _____

Новосибирск 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной курсовой работе рассматривается задача разработки параллельной программы для нахождения кратчайших путей в графе методом Флойда с использованием библиотеки MPI [1]. Алгоритм Флойда, также известный как алгоритм Флойда-Уоршелла, позволяет находить кратчайшие расстояния между всеми парами вершин в графе, что делает его полезным для решения различных задач в области теории графов и оптимизации. Основное внимание в работе уделено реализации алгоритма Флойда и его параллельной версии.

- Цель работы:
- Изучить алгоритм Флойда.
- Реализовать алгоритм Флойда на языке Си
- Реализовать параллельную версию алгоритма Флойда с использованием библиотеки MPI
- Сравнить обе версии алгоритма и провести анализ

1. ОПИСАНИЕ АЛГОРИТМА

1.1 Общая характеристика метода

Алгоритм Флойда, также известный как алгоритм Флойда-Уоршелла, предназначен для нахождения кратчайших путей между всеми парами вершин в графе. Он работает как для ориентированных, так и для неориентированных графов и может обрабатывать графы с отрицательными весами, при условии, что в графе нет отрицательных циклов.

Основные шаги алгоритма:

Инициализация:

Создается матрица расстояний D, где D[i][j] представляет собой вес ребра между вершинами i и j. Если между ними нет ребра, то значение устанавливается в бесконечность (или очень большое число). Для каждой вершины i расстояние до самой себя D[i][i] устанавливается в 0.

Основной цикл:

Алгоритм проходит через каждую вершину k в графе и обновляет матрицу расстояний. Для каждой пары вершин (i, j) проверяется, можно ли улучшить текущее расстояние D[i][j], используя вершину k в качестве промежуточной

1.2 Последовательная версия алгоритма

Первым этапом является инициализация данных и выделение памяти под начальную матрицу. Матрица графов заполняется из файла.

После выполнения всех инициализаций выполняется сам алгоритм. За каждую вершину программа выполняет следующий алгоритм на всех элементах матрицы:

D[i][j] = min(D[i][j], D[k][j] + D[i][k])

Где k - текущая итерация цикла.

Результирующая матрица должна стать матрицей кратчайших путей между элементами графа.

1.3 Параллельная версия алгоритма

Инициализация MPI:

Программа начинается с инициализации MPI с помощью MPI_Init, что позволяет процессам взаимодействовать друг с другом.

Определение ранга и размера коммуникатора:

MPI_Comm_size и MPI_Comm_rank используются для получения общего количества процессов (size) и ранга текущего процесса (rank).

Чтение матрицы смежности:

Если текущий процесс является корневым (rank == ROOT_PROC), он читает матрицу смежности из файла "matrix". Эта матрица представляет граф, где элементы матрицы указывают на веса ребер между вершинами. Если нет ребра, то вес может быть равен INF (бесконечность).

Рассылка информации о количестве вершин:

Корневой процесс рассылает количество вершин (numberOfVert) всем другим процессам с помощью MPI_Bcast.

Распределение работы между процессами:

Корневой процесс вычисляет, сколько строк матрицы будет обрабатываться каждым из рабочих процессов. Это делается с помощью массива rowCounts, который хранит количество строк для каждого процесса.

```
54      int workSize = size - 1;
55      int *rowCounts = (int *)malloc(sizeof(int) * workSize);
56
57      for (int i = 0; i < workSize - 1; i++) {
58        rowCounts[i] = numberOfVert / workSize;
59      }
60      if (numberOfVert % workSize == 0) {
61        rowCounts[workSize - 1] = numberOfVert / workSize;
62      } else {
63        rowCounts[workSize - 1] =
64            numberOfVert - ((numberOfVert / workSize) * (workSize - 1));
65      }
```

Рисунок 1.1 – Реализация распределения строк между процессами

Основной цикл алгоритма:

Основной цикл выполняется для каждой вершины графа. В каждой итерации:

Корневой процесс отправляет строки матрицы другим процессам с помощью MPI_Isend.

Рабочие процессы получают строки и обновляют их, используя алгоритм Флойда-Уоршелла для нахождения кратчайших путей. Каждый рабочий процесс обновляет свои строки, основываясь на полученной строке и текущем состоянии своих строк.

```
106     MPI_Recv(&rowCount, 1, MPI_INT, ROOT_PROC, 0, MPI_COMM_WORLD, &status
            );
107     cout << rank << " : " << rowCount << endl;
108     int *kRow = (int *)malloc(sizeof(int) * numberOfVert);
109     int *rows = (int *)malloc(sizeof(int) * (rowCount * numberOfVert));
110     for (int k = 0; k < numberOfVert; k++) {
111       MPI_Recv(kRow, numberOfVert, MPI_INT, ROOT_PROC, 0, MPI_COMM_WORLD,
112                &status);
113       for (int i = 0; i < rowCount; i++) {
114         MPI_Recv(&rows[i * numberOfVert], numberOfVert, MPI_INT, ROOT_PROC
                , 0,
115                  MPI_COMM_WORLD, &status);
116       }
117       for (int i = 0; i < rowCount; i++) {
118         for (int j = 0; j < numberOfVert; j++) {
119           rows[i * numberOfVert + j] = min(
120             rows[i * numberOfVert + j], rows[i * numberOfVert + k] +
                    kRow[j]);
121         }
122       }
123       for (int i = 0; i < rowCount; i++) {
124         MPI_Send(&rows[i * numberOfVert], numberOfVert, MPI_INT, ROOT_PROC
```

Рисунок

Рисунок 1.2 – Реализация алгоритма

Также все процессы имеют у себя k-ю строку в которой находятся зависимые данные.

После обновления строки рабочие процессы отправляют их обратно корневому процессу с помощью MPI_Send.

Вывод результатов:

После завершения всех итераций корневой процесс выводит финальную матрицу, которая содержит кратчайшие пути между всеми парами вершин, и время, затраченное на выполнение алгоритма.

Освобождение ресурсов:

В конце программы освобождаются выделенные ресурсы, и MPI завершается с помощью MPI_Finalize.

2. ЭКСПЕРИМЕНТАЛЬНЫЕ ИЗМЕРЕНИЯ

2.1 Организация эксперимента

Для проведения эксперимента использовался вычислительный кластер Oak [3]. Кластер состоит из четырех узлов x86-64: 2 x Intel Xeon Quad E5620, RAM 24GB. Коммуникационная сеть: InfiniBand QDR (HCA Mellanox MT26428, switch Mellanox InfiniScale IV IS5030 QDR 36-port), управляющая сеть: Gigabit Ethernet.

В качестве входных данных использовалась матрица графов для 10 и 5 вершин. Эксперимент проводился на 1, 2, 4, 8 процессах.

2.2 Результаты эксперимента

Таблица 2.1 – результаты эксперимента

Количество процессов	Время выполнения алгоритма для матрицы 10x10, с	Время выполнения алгоритма для матрицы 5x5, с
1	0.0423112	0.0371765
2	0.0262745	0.0236365
4	0.0207577	0.0191842
8	0.0192553	0.0193231

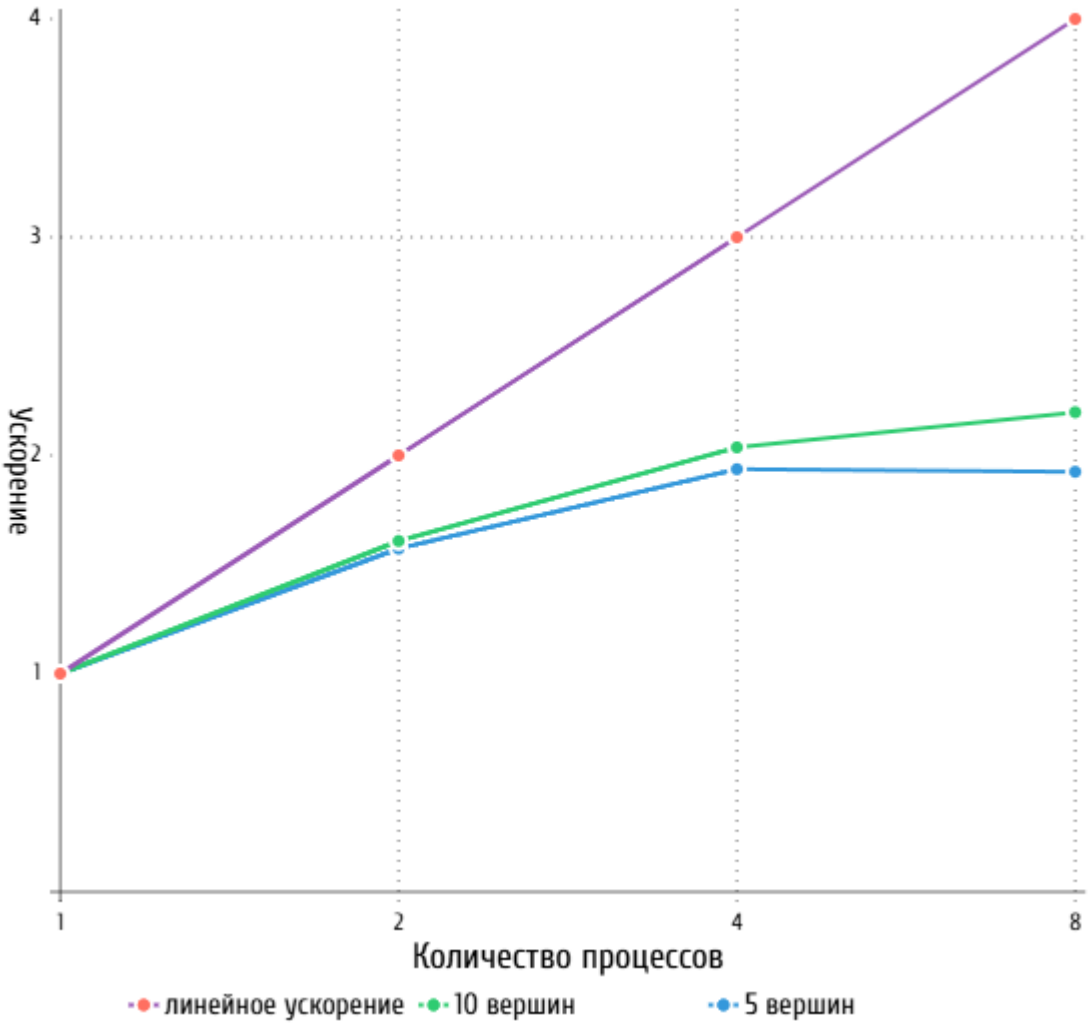

Рисунок 1 – График зависимости коэффициента ускорения от числа процессов

На рисунке 2.1 показано ускорение параллельной программы при разном числе процессов. Из графика видно, что для графов с большим количеством вершин эффективность параллелизации. Для графов с небольшим количеством вершин межпроцессное общение занимает значительную часть времени и не дает такого же прироста. Также стоит учитывать, что увеличение количества процессов на число большее, чем количество строк в матрицы не принесет прироста к производительности.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был разработан и исследован параллельный алгоритм Флойда для вычисления кратчайших путей в графе. Проведенное моделирование алгоритма показало, что при увеличении числа процессов наблюдается значительное ускорение выполнения программы.

Для больших графов алгоритм демонстрирует почти линейное ускорение при использовании до 4 процессов. Это свидетельствует о том, что параллельная реализация алгоритма эффективно использует доступные ресурсы, что позволяет значительно сократить время выполнения.

Однако с увеличением числа процессов выше 4 наблюдается снижение прироста производительности. Это связано с накладными расходами на коммуникацию между процессами, которые начинают доминировать над вычислительной нагрузкой. В результате, при использовании слишком большого количества процессов, эффективность алгоритма снижается, и прирост ускорения становится менее выраженным.

Для меньших графов масштабируемость алгоритма оказывается менее заметной. Это объясняется тем, что вычислительная нагрузка на процессах оказывается недостаточной по сравнению с затратами на обмен данными между ними. В таких случаях накладные расходы на управление процессами и передачу данных могут превышать выгоды от параллелизации.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

Алгоритм Флойда // https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

1. Функция mpi-bcast // https://learn.microsoft.com/ru-ru/message-passing-interface/mpi-bcast-function
2. Вычислительный кластер Oak // https://wiki.cpct.sibsutis.ru/

ПРИЛОЖЕНИЕ

Исходный код параллельного алгоритма

```
#include <iostream>

#include <algorithm>

#include <fstream>

#include <mpi.h>

#define INF 101

#define ROOT_PROC 0

using namespace std;

void printMatrix(int matrix, int numberOfVert) { for (int i = 0; i < numberOfVert; i++) { for (int j = 0; j < numberOfVert; j++) { cout << matrix[inumberOfVert + j] << " "; } cout << endl; } }

int main(int argc, char** argv) { srand(time(NULL)); MPI_Init(&argc, &argv);

int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);


int numberOfVert;
double time, timeResult;
MPI_Request request;
MPI_Status status;


if(rank == ROOT_PROC) {
    numberOfVert = 5;
    int *matrix;
    time = MPI_Wtime();
    ifstream file("matrix");
```

```
    file >> numberOfVert;
    matrix = (int *)malloc(sizeof(int) * (numberOfVert * numberOfVert));
    for (int i = 0; i < numberOfVert; i++) {
        for (int j = 0; j < numberOfVert; j++) {
            file >> matrix[i*numberOfVert + j];
        }
    }

    file.close();

    cout << "Root process. Number of vertex sent to other processes" << endl;
    MPI_Bcast(&numberOfVert, 1, MPI_INT, ROOT_PROC, MPI_COMM_WORLD);

    cout << "Root process. Print first matrix:" << endl;
    printMatrix(matrix, numberOfVert);
    int workSize = size - 1;
    int *rowCounts = (int*)malloc(sizeof(int) * workSize);

    for(int i = 0; i < workSize - 1; i++) {
        rowCounts[i] = numberOfVert / workSize;
    }
    if(numberOfVert % workSize == 0) {
        rowCounts[workSize-1] = numberOfVert / workSize;
    }
    else {
        rowCounts[workSize - 1] = numberOfVert - ((numberOfVert / workSize)
*  (workSize - 1));
    }

    cout << "Root process. Row counts sent to other processes" << endl;
    for(int i = 1; i < size; i++) {
        MPI_Send(&rowCounts[i-1], 1, MPI_INT, i, 0, MPI_COMM_WORLD);
    }

    cout << "Root process. Main loop start" << endl;
    for(int k = 0; k < numberOfVert; k++) {
```

```
        for(int p = 1; p < size; p++) {
            MPI_Isend(&matrix[k*numberOfVert], numberOfVert, MPI_INT, p, 0,
MPI_COMM_WORLD, &request);
        }
        int num = 0;
        for(int i = 0; i < workSize; i++) {
            for(int j = 0; j < rowCounts[i]; j++) {
                MPI_Isend(&matrix[num*numberOfVert], numberOfVert, MPI_INT,
i+1, 0, MPI_COMM_WORLD, &request);
                num++;
            }
        }
        num = 0;
        for(int i = 0; i < workSize; i++) {
            for(int j = 0; j < rowCounts[i]; j++) {
                MPI_Recv(&matrix[num*numberOfVert], numberOfVert, MPI_INT,
i+1, 0, MPI_COMM_WORLD, &status);
                num++;
            }
        }
    }

    cout << "Root process. Print final matrix:" << endl;
    printMatrix(matrix, numberOfVert);

    timeResult = MPI_Wtime() - time;
    cout << "Root process. Time spent: " << timeResult << endl;

    free(matrix);
}
else {
    MPI_Bcast(&numberOfVert, 1, MPI_INT, ROOT_PROC, MPI_COMM_WORLD);
    int rowCount;
    MPI_Recv(&rowCount, 1, MPI_INT, ROOT_PROC, 0, MPI_COMM_WORLD, &status);
    int *kRow = (int *)malloc(sizeof(int) * numberOfVert);
    int *rows = (int *)malloc(sizeof(int) * (rowCount * numberOfVert));
    for(int k = 0; k < numberOfVert; k++) {
```

```
        MPI_Recv(kRow, numberOfVert, MPI_INT, ROOT_PROC, 0, MPI_COMM_WORLD,
&status);
        for(int i = 0; i < rowCount; i++) {
            MPI_Recv(&rows[i*numberOfVert], numberOfVert, MPI_INT,
ROOT_PROC, 0, MPI_COMM_WORLD, &status);
        }
        for (int i = 0; i < rowCount; i++) {
            for (int j = 0; j < numberOfVert; j++) {
                rows[i*numberOfVert + j] = min(
                        rows[i*numberOfVert + j],
                        rows[i*numberOfVert + k] + kRow[j]
                );
            }
        }
        for(int i = 0; i < rowCount; i++) {
            MPI_Send(&rows[i*numberOfVert], numberOfVert, MPI_INT,
ROOT_PROC, 0, MPI_COMM_WORLD);
        }
    }
    free(kRow);
    free(rows);
}
MPI_Finalize();
return 0;


      }
```