**Лабораторная работа № 3 Астаппева Олега**

**Тема: Параллельные алгоритмы обработки графов**

Таблица 1. Результаты вычислительных экспериментов для алгоритма Флойда.

|  |  |  |
| --- | --- | --- |
| Номер теста | Количество вершин в графе | Время работы алгоритма (сек.) |
| 1 | 10 | 0.000000 |
| 2 | 500 | 4.048000 |
| 3 | 600 | 6.924000 |
| 4 | 700 | 10.938000 |
| 5 | 800 | 16.448000 |
| 6 | 900 | 23.420000 |
| 7 | 1000 | 32.195000 |

Таблица 2. Сравнение экспериментального и теоретического времени работы алгоритма Флойда.

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения одной операции τ (сек.) | | | |
| Номер теста | Количество вершин в графе | Время работы (сек.) | Теоретическое время (сек.) |
| 1 | 10 | 0.000000 | 0.000032 |
| 2 | 500 | 4.048000 | 4.015625 |
| 3 | 600 | 6.924000 | 6.939000 |
| 4 | 700 | 10.938000 | 11.018875 |
| 5 | 800 | 16.448000 | 16.448000 |
| 6 | 900 | 23.420000 | 23.419125 |
| 7 | 1000 | 32.195000 | 32.125000 |

Таблица 3**.** Результаты вычислительных экспериментов для параллельного алгоритма Флойда.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер теста | Количество вершин | Последовательный алгоритм Флойда | Параллельный алгоритм Флойда | | |
| 2 | 4 | 8 |
| 1 | 10 | 0.000000 | 0.000296 | 0.000776 | 0.001876 |
| 2 | 500 | 4.048000 | 2.178141 | 1.192616 | 0.855874 |
| 3 | 600 | 6.924000 | 3.437286 | 2.028100 | 1.458038 |
| 4 | 700 | 10.938000 | 5.481218 | 3.256542 | 2.333569 |
| 5 | 800 | 16.448000 | 8.177224 | 4.816982 | 3.422679 |
| 6 | 900 | 23.420000 | 11.623474 | 6.917077 | 5.154873 |
| 7 | 1000 | 32.195000 | 15.914496 | 9.504784 | 6.690693 |

Таблица 4.Ускорение вычислений, получаемое для параллельного алгоритма Флойда.

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Ускорение | | |
| 2 процесса | 4 процесса | 8 процессов |
| 1 | 0.000000 | 0.000000 | 0.000000 |
| 2 | 0.538078 | 0.294619 | 0.211431 |
| 3 | 0.496431 | 0.292909 | 0.210577 |
| 4 | 0.501117 | 0.297727 | 0.213345 |
| 5 | 0.497156 | 0.292861 | 0.208091 |
| 6 | 0.496305 | 0.295349 | 0.220106 |
| 7 | 0.494316 | 0.295225 | 0.207818 |

Таблица 5.Сравнение экспериментального и теоретического времени параллельного алгоритма Флойда.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Номер теста | Кол-во данных | Время выполнения параллельного алгоритма | | | | | |
| 2 процесса | | 4 процесса | | 8 процессов | |
| Модель | Эксперимент | Модель | Эксперимент | Модель | Эксперимент |
| 1 | 10 | 0.000016 | 0.000296 | 0.000008 | 0.000776 | 0.000003 | 0.001876 |
| 2 | 500 | 2.007812 | 2.178141 | 1.003906 | 1.192616 | 0.446180 | 0.855874 |
| 3 | 600 | 3.469500 | 3.437286 | 1.734750 | 2.028100 | 0.771000 | 1.458038 |
| 4 | 700 | 5.509437 | 5.481218 | 2.754718 | 3.256542 | 1.224319 | 2.333569 |
| 5 | 800 | 8.224000 | 8.177224 | 4.112000 | 4.816982 | 1.827555 | 3.422679 |
| 6 | 900 | 11.709562 | 11.623474 | 5.854781 | 6.917077 | 2.602125 | 5.154873 |
| 7 | 1000 | 16.062500 | 15.914496 | 8.031250 | 9.504784 | 3.569444 | 6.690693 |

Тест 1.

#include <cstdlib>

#include <cstdio>

#include <ctime>

#include <iostream>

#include <algorithm>

using namespace std;

const double InfinitiesPercent = 50.0;

const double RandomDataMultiplier = 10;

// Function for initializing the data by the random generator

void RandomDataInitialization(int \*pMatrix, int Size) {

srand((unsigned)time(0));

for (int i = 0; i < Size; i++)

for (int j = 0; j < Size; j++)

if (i != j) {

if ((rand() % 100) < InfinitiesPercent)

pMatrix[i \* Size + j] = -1;

else

pMatrix[i \* Size + j] = rand() + 1;

}

else

pMatrix[i \* Size + j] = 0;

}

// Function for formatted matrix output

void PrintMatrix(int \*pMatrix, int RowCount, int ColCount) {

for (int i = 0; i < RowCount; i++) {

for (int j = 0; j < ColCount; j++)

printf("%7d", pMatrix[i \* ColCount + j]);

printf("\n");

}

}

// Function for simple setting the initial data

void DummyDataInitialization(int \*pMatrix, int Size) {

for (int i = 0; i < Size; i++)

for (int j = i; j < Size; j++) {

if (i == j) pMatrix[i \* Size + j] = 0;

else

if (i == 0) pMatrix[i \* Size + j] = j;

else pMatrix[i \* Size + j] = -1;

pMatrix[j \* Size + i] = pMatrix[i \* Size + j];

}

}

// Function for allocating the memory and setting the initial values

void ProcessInitialization(int \*&pMatrix, int& Size) {

do {

printf("Enter the number of vertices: ");

cin >> Size;

if (Size <= 0)

printf("The number of vertices should be greater th zero\n");

} while (Size <= 0);

printf("Using graph with %d vertices\n", Size);

// Allocate memory for the adjacency matrix

pMatrix = new int[Size \* Size];

// Data initalization

//DummyDataInitialization(pMatrix, Size);

RandomDataInitialization(pMatrix, Size);

}

// Function for computational process termination

void ProcessTermination(int \*pMatrix) {

delete[]pMatrix;

}

int Min(int A, int B) {

int Result = (A < B) ? A : B;

if ((A < 0) && (B >= 0)) Result = B;

if ((B < 0) && (A >= 0)) Result = A;

if ((A < 0) && (B < 0)) Result = -1;

return Result;

}

// Serial Floyd algorithm

void SerialFloyd(int \*pMatrix, int Size) {

int t1, t2;

for (int k = 0; k < Size; k++)

for (int i = 0; i < Size; i++)

for (int j = 0; j < Size; j++)

if ((pMatrix[i \* Size + k] != -1) &&

(pMatrix[k \* Size + j] != -1)) {

t1 = pMatrix[i \* Size + j];

t2 = pMatrix[i \* Size + k] + pMatrix[k \* Size + j];

pMatrix[i \* Size + j] = Min(t1, t2);

}

}

int main()

{

int i = 1;

do {

int \*pMatrix; // Adjacency matrix

int Size; // Size of adjacency matrix

time\_t start, finish;

double duration = 0.0;

printf("Serial Floyd algorithm\n");

ProcessInitialization(pMatrix, Size);

start = clock();

// Parallel Floyd algorithm

SerialFloyd(pMatrix, Size);

finish = clock();

//printf("The matrix after Floyd algorithm\n");

//PrintMatrix(pMatrix, Size, Size);

duration = (finish - start) / double(CLOCKS\_PER\_SEC);

printf("Time of execution: %f\n", duration);

} while (i++ < 7);

cin.get();

return 0;

}

Тест 2.

#include <cstdlib>

#include <cstdio>

#include <ctime>

#include <algorithm>

#include <mpi.h>

#define FIRST\_VERSION 0

#include "ParallelFloyd.h"

#include "ParallelFloydTest.h"

using namespace std;

static int ProcRank; // Rank of current process

static int ProcNum; // Number of processes

const double InfinitiesPercent = 50.0;

const double RandomDataMultiplier = 10;

int Min(int A, int B) {

int Result = (A < B) ? A : B;

if((A < 0) && (B >= 0)) Result = B;

if((B < 0) && (A >= 0)) Result = A;

if((A < 0) && (B < 0)) Result = -1;

return Result;

}

int main(int argc, char\* argv[]) {

int \*pMatrix; // Adjacency matrix

int Size; // Size of adjacency matrix

int \*pProcRows; // Process rows

int RowNum; // Number of process rows

double start, finish;

double duration = 0.0;

int \*pSerialMatrix = 0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

if(ProcRank == 0)

printf("Parallel Floyd algorithm\n");

// Process initialization

ProcessInitialiazation(pMatrix, pProcRows, Size, RowNum);

if (ProcRank == 0) {

// Matrix copying

pSerialMatrix = new int[Size \* Size];

CopyMatrix(pMatrix, Size, pSerialMatrix);

}

start = MPI\_Wtime();

// Distributing the initial data between processes

DataDistribution(pMatrix, pProcRows, Size, RowNum);

// Testing the distribution

//TestDistribution(pMatrix, pProcRows, Size, RowNum);

// Parallel Floyd algorithm

ParallelFloyd(pProcRows, Size, RowNum);

//ParallelPrintMatrix(pProcRows, Size, RowNum);

// Process data collection

ResultCollection(pMatrix, pProcRows, Size, RowNum);

//if(ProcRank == 0)

// PrintMatrix(pMatrix, Size, Size);

finish = MPI\_Wtime();

//TestResult(pMatrix, pSerialMatrix, Size);

duration = finish - start;

if(ProcRank == 0)

printf("Time of execution: %f\n", duration);

if (ProcRank == 0)

delete []pSerialMatrix;

// Process termination

ProcessTermination(pMatrix, pProcRows);

MPI\_Finalize();

return 0;

}

// Function for allocating the memory and setting the initial values

void ProcessInitialiazation(int \*&pMatrix, int \*&pProcRows, int& Size, int& RowNum) {

setvbuf(stdout, 0, \_IONBF, 0);

if(ProcRank == 0) {

do {

printf("Enter the number of vertices: ");

scanf("%d", &Size);

if(Size < ProcNum)

printf("The number of vertices should be greater then number of processes\n");

} while(Size < ProcNum);

printf("Using graph with %d vertices\n", Size);

}

// Broadcast the number of vertices

MPI\_Bcast(&Size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Number of rows for each process

int RestRows = Size;

for(int i = 0; i < ProcRank; i++)

RestRows = RestRows - RestRows / (ProcNum - i);

RowNum = RestRows / (ProcNum - ProcRank);

// Allocate memory for the current process rows

pProcRows = new int[Size \* RowNum];

if(ProcRank == 0) {

// Allocate memory for the adjacency matrix

pMatrix = new int[Size \* Size];

// Data initalization

DummyDataInitialization(pMatrix, Size);

//RandomDataInitialization(pMatrix, Size);

}

}

// Function for computational process termination

void ProcessTermination(int \*pMatrix, int \*pProcRows) {

if(ProcRank == 0)

delete []pMatrix;

delete []pProcRows;

}

// Function for simple setting the initial data

void DummyDataInitialization(int \*pMatrix, int Size) {

for(int i = 0; i < Size; i++)

for(int j = i; j < Size; j++) {

if(i == j) pMatrix[i \* Size + j] = 0;

else

if(i == 0) pMatrix[i \* Size + j] = j;

else pMatrix[i \* Size + j] = -1;

pMatrix[j \* Size + i] = pMatrix[i \* Size + j];

}

}

// Function for setting the data by the random generator

void RandomDataInitialization(int \*pMatrix, int Size) {

srand( (unsigned)time(0) );

for(int i = 0; i < Size; i++)

for(int j = 0; j < Size; j++)

if(i != j) {

if((rand() % 100) < InfinitiesPercent)

pMatrix[i \* Size + j] = -1;

else

pMatrix[i \* Size + j] = rand() + 1;

}

else

pMatrix[i \* Size + j] = 0;

}

// Data distribution among the processes

void DataDistribution(int \*pMatrix, int \*pProcRows, int Size, int RowNum) {

int \*pSendNum; // The number of elements sent to the process

int \*pSendInd; // The index of the first data element sent to the process

int RestRows = Size; // Number of rows, that haven’t been distributed yet

// Allocate memory for temporary objects

pSendInd = new int[ProcNum];

pSendNum = new int[ProcNum];

// Define the disposition of the matrix rows for current process

RowNum = Size / ProcNum;

pSendNum[0] = RowNum \* Size;

pSendInd[0] = 0;

for (int i = 1; i < ProcNum; i++) {

RestRows -= RowNum;

RowNum = RestRows / (ProcNum - i);

pSendNum[i] = RowNum \* Size;

pSendInd[i] = pSendInd[i - 1] + pSendNum[i - 1];

}

// Scatter the rows

MPI\_Scatterv(pMatrix, pSendNum, pSendInd, MPI\_INT,

pProcRows, pSendNum[ProcRank], MPI\_INT, 0, MPI\_COMM\_WORLD);

// Free allocated memory

delete []pSendNum;

delete []pSendInd;

}

// Function for process result collection

void ResultCollection(int \*pMatrix, int \*pProcRows, int Size, int RowNum) {

int \*pReceiveNum; // Number of elements, that current process sends

int \*pReceiveInd; /\* Index of the first element from current process

in result vector \*/

int RestRows = Size; // Number of rows, that haven’t been gathered yet

// Allocate memory for temporary objects

pReceiveNum = new int[ProcNum];

pReceiveInd = new int[ProcNum];

// Define the disposition of the result vector block of current process

RowNum = Size / ProcNum;

pReceiveInd[0] = 0;

pReceiveNum[0] = RowNum \* Size;

for(int i = 1; i < ProcNum; i++) {

RestRows -= RowNum;

RowNum = RestRows / (ProcNum - i);

pReceiveNum[i] = RowNum \* Size;

pReceiveInd[i] = pReceiveInd[i - 1] + pReceiveNum[i - 1];

}

// Gather the whole matrix on process with rank 0

MPI\_Gatherv(pProcRows, pReceiveNum[ProcRank], MPI\_INT,

pMatrix, pReceiveNum, pReceiveInd, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Free allocated memory

delete []pReceiveNum;

delete []pReceiveInd;

}

// Parallel Floyd algorithm

void ParallelFloyd(int \*pProcRows, int Size, int RowNum) {

int \*pRow = new int[Size];

int t1, t2;

for(int k = 0; k < Size; k++) {

// Distribute row among all processes

RowDistribution(pProcRows, Size, RowNum, k, pRow);

if(ProcRank == 0) {

printf("Row %d after distribution:", k);

fflush(stdout);

PrintMatrix(pRow, 1, Size);

}

// Update adjacency matrix elements

for(int i = 0; i < RowNum; i++)

for(int j = 0; j < Size; j++)

if( (pProcRows[i \* Size + k] != -1) &&

(pRow [j] != -1)) {

t1 = pProcRows[i \* Size + j];

t2 = pProcRows[i \* Size + k] + pRow[j];

pProcRows[i \* Size + j] = Min(t1, t2);

}

}

delete []pRow;

}

// Function for row broadcasting among all processes

void RowDistribution(int \*pProcRows, int Size, int RowNum, int k, int \*pRow) {

int ProcRowRank; // Process rank with the row k

int ProcRowNum; // Process row number

// Finding the process rank with the row k

int RestRows = Size;

int Ind = 0;

int Num = Size / ProcNum;

for(ProcRowRank = 1; ProcRowRank < ProcNum + 1; ProcRowRank ++) {

if(k < Ind + Num ) break;

RestRows -= Num;

Ind += Num;

Num = RestRows / (ProcNum - ProcRowRank);

}

ProcRowRank = ProcRowRank - 1;

ProcRowNum = k - Ind;

if(ProcRowRank == ProcRank)

// Copy the row to pRow array

copy(&pProcRows[ProcRowNum\*Size],&pProcRows[(ProcRowNum+1)\*Size],pRow);

// Broadcast row to all processes

MPI\_Bcast(pRow, Size, MPI\_INT, ProcRowRank, MPI\_COMM\_WORLD);

}

// Function for formatted output of all stripes

void ParallelPrintMatrix(int \*pProcRows, int Size, int RowNum) {

for(int i = 0; i < ProcNum; i++) {

MPI\_Barrier(MPI\_COMM\_WORLD);

if (ProcRank == i) {

printf("ProcRank = %d\n", ProcRank);

fflush(stdout);

printf("Proc rows:\n");

fflush(stdout);

PrintMatrix(pProcRows, RowNum, Size);

fflush(stdout);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

}

}

// Function for testing the data distribution

void TestDistribution(int \*pMatrix, int \*pProcRows, int Size, int RowNum) {

MPI\_Barrier(MPI\_COMM\_WORLD);

if (ProcRank == 0) {

printf("Initial adjacency matrix:\n");

PrintMatrix(pMatrix, Size, Size);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

ParallelPrintMatrix(pProcRows, Size, RowNum);

}

// Testing the result of parallel Floyd algorithm

void TestResult(int \*pMatrix, int \*pSerialMatrix, int Size) {

MPI\_Barrier(MPI\_COMM\_WORLD);

if(ProcRank == 0) {

SerialFloyd(pSerialMatrix, Size);

if(!CompareMatrices(pMatrix, pSerialMatrix, Size)) {

printf("Results of serial and parallel algorithms are "

"NOT identical. Check your code\n");

}

else {

printf("Results of serial and parallel algorithms are "

"identical\n");

}

}

}

ParallelFloydTest.h

#ifndef PARALLELFLOYDTEST\_H\_

#define PARALLELFLOYDTEST\_H\_

// Function for copying the matrix

void CopyMatrix(int \*pMatrix, int Size, int \*pMatrixCopy);

// Function for comparing the matrices

bool CompareMatrices(int \*pMatrix1, int \*pMatrix2, int Size);

// Serial Floyd algorithm

void SerialFloyd(int \*pMatrix, int Size);

// Function for formatted matrix output

void PrintMatrix(int \*pMatrix, int RowCount, int ColCount);

#endif

ParallelFloyd.h

#ifndef PARALLELFLOYD\_H\_

#define PARALLELFLOYD\_H\_

int Min(int A, int B);

// Function for allocating the memory and setting the initial values

void ProcessInitialiazation(int \*&pMatrix, int \*&pProcRows, int& Size, int& RowNum);

// Function for computational process termination

void ProcessTermination(int \*pMatrix, int \*pProcRows);

// Function for simple setting the initial data

void DummyDataInitialization(int \*pMatrix, int Size);

// Function for setting the data by the random generator

void RandomDataInitialization(int \*pMatrix, int Size);

// Data distribution among the processes

void DataDistribution(int \*pMatrix, int \*pProcRows, int Size, int RowNum);

// Function for process result collection

void ResultCollection(int \*pMatrix, int \*pProcRows, int Size, int RowNum);

// Parallel Floyd algorithm

void ParallelFloyd(int \*pProcRows, int RowSize, int RowNum);

// Function for row broadcasting among all processes

void RowDistribution(int \*pProcRows, int Size, int RowNum, int k, int \*pRow);

// Function for formatted output of all stripes

void ParallelPrintMatrix(int \*pProcRows, int Size, int RowNum);

// Function for testing the data distribution

void TestDistribution(int \*pMatrix, int \*pProcRows, int Size, int RowNum);

// Testing the result of parallel Floyd algorithm

void TestResult(int \*pMatrix, int \*pSerialMatrix, int Size);

#endif