**Лабораторная работа № 4 Астаппев Олег**

**Тема: Параллельные алгоритмы обработки графов**

Таблица 1. Результаты вычислительных экспериментов для метода Гаусса-Зейделя.

|  |  |  |  |
| --- | --- | --- | --- |
| Номер теста | Размер матрицы | Количество итераций | Время работы (сек.) |
| 1 | 10 | 6 | 0,000000 |
| 2 | 100 | 5 | 0,001000 |
| 3 | 1000 | 5 | 0,159000 |
| 4 | 2000 | 5 | 0,642000 |
| 5 | 3000 | 5 | 1,380000 |
| 6 | 4000 | 5 | 2,362000 |

Таблица 2. Сравнение экспериментального и теоретического времени метода Гаусса-Зейделя.

|  |  |  |  |
| --- | --- | --- | --- |
| Время выполнения базовой вычислительной операции τ (сек.) | | | |
| Номер теста | Размер матрицы | Время работы (сек.) | Теоретическое время (сек.) |
| 1 | 10 | 0,000000 | 0,125000 |
| 2 | 100 | 0,001000 | 0,498805 |
| 3 | 1000 | 0,159000 | 0,988050 |
| 4 | 2000 | 0,642000 | 1,195220 |
| 5 | 3000 | 1,380000 | 2,689240 |
| 6 | 4000 | 2,362000 | 4,780880 |

Таблица 3**.** Результаты вычислительных экспериментов для параллельного алгоритма Гаусса-Зейделя.

|  |  |  |  |
| --- | --- | --- | --- |
| Размер сетки | Последовательный алгоритм | Параллельный алгоритм | |
| 2 процессора | |
| Время | Ускорение |
| 10 | 0.000000 | 0.000000 | - |
| 100 | 0.001000 | 0,001639 | 0.610 |
| 1000 | 0.159000 | 0,162245 | 0.980 |
| 2000 | 0.642000 | 0,617308 | 1.040 |
| 3000 | 1.380000 | 1,131148 | 1.220 |
| 4000 | 2.362000 | 1,831008 | 1.290 |

Таблица 4.Ускорение вычислений, получаемое для параллельного алгоритма Гаусса-Зейделя.

|  |  |  |
| --- | --- | --- |
| Размер матрицы | Время выполнения параллельного алгоритма | |
| 2 процессора | |
| Модель | Эксперимент |
| 10 | 0.000000 | 0.000000 |
| 100 | 0,001326 | 0,001639 |
| 1000 | 0,135106 | 0,162245 |
| 2000 | 0,524263 | 0,617308 |
| 3000 | 0,874666 | 1,131148 |
| 4000 | 1,580278 | 1,831008 |

**SerialGaussSeidel Project**

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>

#include <math.h>

// Function for formatted matrix output

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

int i, j; // Loop variables

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

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

printf("%7.4f ", pMatrix[i\*ColCount+j]);

printf("\n");

}

}

// Function for the Gauss-Seidel algoritm

void ResultCalculation(double\* pMatrix, int Size, double &Eps,

int &Iterations) {

int i, j; // Loop variables

double dm, dmax,temp;

Iterations = 0;

do {

dmax = 0;

for (i = 1; i < Size - 1; i++)

for(j = 1; j < Size - 1; j++) {

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

pMatrix[Size \* i + j] = 0.25 \* (pMatrix[Size \* i + j + 1] +

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

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

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

dm = fabs(pMatrix[Size \* i + j] - temp);

if (dmax < dm) dmax = dm;

}

Iterations++;

//if (Iterations==1) PrintMatrix (pMatrix, Size, Size);

}

while (dmax > Eps);

}

// Function for computational process termination

void ProcessTermination(double\* pMatrix) {

delete [] pMatrix;

}

// Function for simple setting the grid node values

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

int i, j; // Loop variables

double h = 1.0 / (Size - 1);

// Setting the grid node values

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

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

if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))

pMatrix[i\*Size+j] = 100;

else

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

}

}

// Function for memory allocation and setting the initial values

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

// Setting the size of initial matrix

do {

printf("\nEnter the grid size of the initial objects: ");

scanf("%d", &Size);

printf("\nChosen grid size = %d", Size);

if (Size <= 2)

printf("\nSize of grid must be greater than 2!\n");

}

while (Size <= 2);

// Setting the required accuracy

do {

printf("\nEnter the required accuracy: ");

scanf("%lf", &Eps);

printf("\nChosen accuracy = %lf", Eps);

if (Eps <= 2)

printf("\nAccuracy must be greater than 0!\n");

}

while (Eps <= 0);

// Memory allocation

pMatrix = new double [Size\*Size];

// Setting the grid node values

DummyDataInitialization(pMatrix, Size);

}

// Function for simple definition of matrix and vector elements

void main() {

double\* pMatrix; // The matrix of the grid nodes

int Size; // The matrix size

double Eps; // The required accuracy

int Iterations; // The iteration number

double start, finish, duration = 0.0;

printf("Serial Gauss - Seidel algorithm\n");

ProcessInitialization(pMatrix, Size, Eps);

start = clock();

ResultCalculation(pMatrix, Size, Eps, Iterations);

finish = clock();

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

// Printing the results

printf("\n Number of iterations: %d\n", Iterations);

// Printing the time spent by the Gauss-Seidel method

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

/\*

// Matrix output

printf ("Initial Matrix: \n");

PrintMatrix (pMatrix, Size, Size);

// The Gauss-Seidel method

ResultCalculation(pMatrix, Size, Eps, Iterations);

// Printing the result

printf("\n Number of iterations: %d\n",Iterations);

printf ("\n Result matrix: \n");

PrintMatrix (pMatrix, Size, Size);

\*/

getch();

// Computational process termination

ProcessTermination(pMatrix);

}

**ParallelGaussSeidel Project**

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>

#include <math.h>

#include <mpi.h>

static int ProcNum = 0; // Number of available processes

static int ProcRank = -1; // Rank of current process

// Function for distribution of the initial objects between the processes

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

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;

// Alloc memory for temporary objects

pSendInd = new int [ProcNum];

pSendNum = new int [ProcNum];

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

RowNum = (Size-2)/ProcNum+2;

pSendNum[0] = RowNum\*Size;

pSendInd[0] = 0;

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

RestRows = RestRows - RowNum + 1;

RowNum = (RestRows-2)/(ProcNum-i)+2;

pSendNum[i] = (RowNum)\*Size;

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

}

// Scatter the rows

MPI\_Scatterv(pMatrix , pSendNum, pSendInd, MPI\_DOUBLE, pProcRows,

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

delete []pSendInd;

delete []pSendNum;

}

// Function for computational process termination

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

if (ProcRank == 0)

delete [] pMatrix;

delete [] pProcRows;

}

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

int i,j; //Loop variables

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

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

printf("%7.4f ", pMatrix[i\*ColCount+j]);

printf("\n");

}

}

//Function for the execution of the Gauss-Seidel method iteration

double IterationCalculation(double\* pProcRows, int Size, int RowNum) {

int i, j; // Loop variables

double dm, dmax,temp;

dmax = 0;

for (i = 1; i < RowNum-1; i++)

for(j = 1; j < Size-1; j++) {

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

pProcRows[Size \* i + j] = 0.25 \* (pProcRows[Size \* i + j + 1] +

pProcRows[Size \* i + j - 1] +

pProcRows[Size \* (i + 1) + j] +

pProcRows[Size \* (i - 1) + j]);

dm = fabs(pProcRows[Size \* i + j] - temp);

if (dmax < dm) dmax = dm;

}

return dmax;

}

// Function for formatted matrix output

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

if (ProcRank == 0) {

printf("Initial Matrix: \n");

PrintMatrix(pMatrix, Size, Size);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

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

if (ProcRank == i) {

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

// fprintf(" Matrix Stripe:\n");

PrintMatrix(pProcRows, RowNum, Size);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

}

}

// Function for simple setting the grid node values

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

int i, j; // Loop variables

double h = 1.0 / (Size - 1);

// Setting the grid node values

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

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

if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))

pMatrix[i\*Size+j] = 100;

else

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

}

}

// Function for memory allocation and setting the initial values

void ProcessInitialization (double\* &pMatrix, double\* &pProcRows,int &Size, int &RowNum, double &Eps) {

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

if (ProcRank == 0) {

do {

printf("\nEnter the grid size: ");

scanf("%d", &Size);

if (Size <= 2) {

printf("\n Size of grid must be greater than 2! \n");

}

if (Size < ProcNum) {

printf("Size of the objects must be greater than number of processes! \n ");

}

if ((Size-2)%ProcNum != 0) {

printf("Number of processes must be multiple of size of objects! \n");

}

}

while ( (Size <= 2) || (Size < ProcNum) || ((Size-2)%ProcNum != 0));

// Setting the required accuracy

do {

printf("\nEnter the required accuracy: ");

scanf("%lf", &Eps);

printf("\nChosen accuracy = %lf", Eps);

if (Eps <= 0)

printf("\nAccuracy must be greater than 0!\n");

}

while (Eps <= 0);

}

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

MPI\_Bcast(&Eps, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

// Define the number of matrix rows stored on each process

RestRows = Size;

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

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

RowNum = (RestRows-2)/(ProcNum-ProcRank)+2;

// Memory allocation

pProcRows = new double [RowNum\*Size];

// Define the values of initial objects’ elements

if (ProcRank == 0) {

// Initial matrix exists only on the pivot process

pMatrix = new double [Size\*Size];

// Values of elements are defined only on the pivot process

DummyDataInitialization(pMatrix, Size);

}

}

void ExchangeData(double\* pProcRows, int Size, int RowNum)

{

MPI\_Status status;

int NextProcNum = (ProcRank == ProcNum - 1)? MPI\_PROC\_NULL : ProcRank + 1;

int PrevProcNum = (ProcRank == 0)? MPI\_PROC\_NULL : ProcRank - 1;

// Send to NextProcNum and receive from PrevProcNum

MPI\_Sendrecv(pProcRows + Size \* (RowNum - 2),Size, MPI\_DOUBLE, NextProcNum, 4,

pProcRows, Size, MPI\_DOUBLE, PrevProcNum, 4, MPI\_COMM\_WORLD, &status);

// Send to PrevProcNum and receive from NextProcNum

MPI\_Sendrecv(pProcRows + Size, Size, MPI\_DOUBLE, PrevProcNum, 5,

pProcRows + (RowNum - 1) \* Size, Size,MPI\_DOUBLE, NextProcNum, 5,

MPI\_COMM\_WORLD, &status);

}

//Function for the parallel Gauss0Seidel method

void ParallelResultCalculation(double \*pProcRows, int Size, int RowNum,

double Eps, int &Iterations) {

double ProcDelta,Delta;

Iterations=0;

do {

Iterations++;

//Exchange the boundary rows of the process stripe

ExchangeData(pProcRows, Size,RowNum);

//The Gauss-Seidel method iteration

ProcDelta = IterationCalculation(pProcRows, Size, RowNum);

//Calculating the maximum value of the deviation

MPI\_Allreduce(&ProcDelta, &Delta, 1,MPI\_DOUBLE, MPI\_MAX, MPI\_COMM\_WORLD);

} while (Delta<Eps);

}

// Function for gathering the result vector

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

int i; // Loop variable

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;

//Alloc memory for temporary objects

pReceiveNum = new int [ProcNum];

pReceiveInd = new int [ProcNum];

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

pReceiveInd[0] = 0;

RowNum = (Size-2)/ProcNum+2;

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

pReceiveNum[0] = RowNum\*Size;

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

RestRows = RestRows - RowNum + 1;

RowNum = (RestRows-2)/(ProcNum-i)+2;

pReceiveNum[i] = RowNum\*Size;

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

}

//Gather the whole result vector on every processor

MPI\_Allgatherv(pProcRows, pReceiveNum[ProcRank], MPI\_DOUBLE, pMatrix,

pReceiveNum, pReceiveInd, MPI\_DOUBLE, MPI\_COMM\_WORLD);

//Free the memory

delete [] pReceiveNum;

delete [] pReceiveInd;

}

void SerialResultCalculation(double \*pSerialMatrix, int Size, double Eps, int &Iter){

int i, j; // Loop variables

double dm, dmax,temp;

Iter = 0;

do {

dmax = 0;

for (i = 1; i < Size - 1; i++)

for(j = 1; j < Size - 1; j++) {

temp = pSerialMatrix[Size \* i + j];

pSerialMatrix[Size \* i + j] = 0.25 \* (pSerialMatrix[Size \* i + j + 1] +

pSerialMatrix[Size \* i + j - 1] +

pSerialMatrix[Size \* (i + 1) + j] +

pSerialMatrix[Size \* (i - 1) + j]);

dm = fabs(pSerialMatrix[Size \* i + j] - temp);

if (dmax < dm) dmax = dm;

}

Iter++;

}

while (dmax > Eps);

}

// Function to copy the initial data

void CopyData(double \*pMatrix, int Size, double \*pSerialMatrix)

{

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

pSerialMatrix[i]=pMatrix[i];

//copy(pMatrix, pMatrix + Size, pSerialMatrix);

}

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

int equal = 0; // =1, if the matrices are not equal

int Iter;

if (ProcRank == 0) {

SerialResultCalculation(pSerialMatrix, Size, Eps, Iter);

for (int i=0; i<Size\*Size; i++) {

if (fabs(pSerialMatrix[i]-pMatrix[i]) >= Eps)

equal = 1;break;

}

if (equal == 1)

printf("The results of serial and parallel algorithms are NOT identical. Check your code.");

else

printf("The results of serial and parallel algorithms are identical.");

}

}

// Function for setting the grid node values by a random generator

void RandowmDataInitialization (double\* pMatrix, int Size) {

int i, j; // Loop variables

srand(unsigned(clock()));

// Setting the grid node values

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

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

if ((i==0) || (i== Size-1) || (j==0) || (j==Size-1))

pMatrix[i\*Size+j] = 100;

else

pMatrix[i\*Size+j] = rand()/double(1000);

}

}

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

double\* pMatrix; // The matrix of the grid nodes

double\* pProcRows; // Stripe of the matrix on current process

double\* pSerialMatrix;//The result of the serial method

int Size; // The matrix size

double Eps; // The required accuracy

int Iterations; // The iteration number

double currDelta, delta;

int RowNum;// Number of rows in matrix stripe

double start, finish, duration = 0.0;

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

MPI\_Init(&argc, &argv);

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

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

if(ProcRank == 0) {

printf("Parallel Gauss - Seidel algorithm \n");

fflush(stdout);

}

// Function for memory allocation and setting the initial values

ProcessInitialization (pMatrix,pProcRows,Size,RowNum,Eps);

// Creating the copy of the initial data

if (ProcRank == 0) {

pSerialMatrix = new double[Size\*Size];

CopyData(pMatrix, Size, pSerialMatrix);

}

start = MPI\_Wtime();

// Distributing the initial objects between the processes

DataDistribution(pMatrix, pProcRows, Size,RowNum);

//Paralle Gauss-Seidel method

ParallelResultCalculation(pProcRows, Size,RowNum,Eps, Iterations);

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

//Gathering the calculation results

ResultCollection(pMatrix, pProcRows,Size, RowNum);

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

finish = MPI\_Wtime();

duration = finish - start;

if (ProcRank == 0)

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

/\*

//Printing the result

printf("\n Iter %d \n", Iterations);

printf("\nResult matrix: \n");

if (ProcRank==0) {

//TestResult(pMatrix,Size,pMatrixCopy,Eps);

PrintMatrix(pMatrix,Size,Size);

}

\*/

if (ProcRank == 0) {

delete []pSerialMatrix;

}

// Process termination

ProcessTermination(pMatrix, pProcRows);

MPI\_Finalize();

}