Work 13 Кириленко Константин 351



#include <iostream>

#include <mpi.h>

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

#include <time.h>

#include <math.h>

int\* pPivotPos; // The number of pivot rows selected at the iterations

int\* pPivotIter; // The iterations, at which the rows were pivots

using namespace std;

typedef struct {

int PivotRow;

double MaxValue;

} TThreadPivotRow;

// Finding the pivot row

int ParallelFindPivotRow(double\*\* pMatrix, int Size, int Iter, int NProc, int ProcId) {

int PivotRow = -1; // The index of the pivot row

double MaxValue = 0; // The value of the pivot element

int i; // Loop variable

// Choose the row, that stores the maximum element

//#pragma omp parallel

TThreadPivotRow ThreadPivotRow;

ThreadPivotRow.MaxValue = 0;

ThreadPivotRow.PivotRow = -1;

int\* b = new int[NProc];

double\* m = new double[NProc];

int n1 = Size / NProc;

int n2 = (ProcId + 1) \* n1;

if (NProc == ProcId + 1) {

n2 = Size;

}

int st = ProcId \* n1;

for (int i = st; i < n2; i++) {

if ((pPivotIter[i] == -1) &&

(fabs(pMatrix[i][Iter]) > ThreadPivotRow.MaxValue)) {

ThreadPivotRow.PivotRow = i;

ThreadPivotRow.MaxValue = fabs(pMatrix[i][Iter]);

}

}

b[ProcId] = ThreadPivotRow.PivotRow;

m[ProcId] = ThreadPivotRow.MaxValue;

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

MPI\_Bcast(&(b[i]), 1, MPI\_INT, i, MPI\_COMM\_WORLD);

MPI\_Bcast(&(m[i]), 1, MPI\_DOUBLE, i, MPI\_COMM\_WORLD);

if (m[i] > MaxValue) {

PivotRow = b[i];

MaxValue = m[i];

}

}

return PivotRow;

}

int\* pSerialPivotPos; // The Number of pivot rows selected at the

// iterations

int\* pSerialPivotIter; // The Iterations, at which the rows were pivots

// Function for simple initialization of the matrix

// and the vector elements

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

pVector[0] = 21;

pVector[1] = 12;

pVector[2] = 29;

pVector[3] = 130;

pVector[4] = -13;

pMatrix[0][0] = 5;

pMatrix[0][1] = 2;

pMatrix[0][2] = -7;

pMatrix[0][3] = 14;

pMatrix[0][4] = 0;

pMatrix[1][0] = 5;

pMatrix[1][1] = -1;

pMatrix[1][2] = 8;

pMatrix[1][3] = -13;

pMatrix[1][4] = 3;

pMatrix[2][0] = 10;

pMatrix[2][1] = 1;

pMatrix[2][2] = -2;

pMatrix[2][3] = 7;

pMatrix[2][4] = -1;

pMatrix[3][0] = 15;

pMatrix[3][1] = 3;

pMatrix[3][2] = 15;

pMatrix[3][3] = 9;

pMatrix[3][4] = 7;

pMatrix[4][0] = 2;

pMatrix[4][1] = -1;

pMatrix[4][2] = -4;

pMatrix[4][3] = 5;

pMatrix[4][4] = -7;

}

// Function for random initialization of the matrix

// and the vector elements

void RandomDataInitialization(double\*\* pMatrix, double\* pVector,

int Size) {

int i, j; // Loop variables

srand(unsigned(clock()));

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

pVector[i] = rand() / double(1000);

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

if (j <= i)

pMatrix[i ][ j] = rand() / double(1000);

else

pMatrix[i ][j] = 0;

}

}

}

void ProcessInitialization(double\*\*& pMatrix, double\*& pVector,

double\*& pResult, int& Size, int NProc, int ProcId) {

// Setting the size of the matrix and the vector

if (ProcId == 0) {

do {

printf("\nEnter size of the matrix and the vector: ");

// scanf\_s("%d", &Size);

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

if (Size <= 0)

printf("\nSize of objects must be greater than 0!\n");

} while (Size <= 0);

}

// Memory allocation

pMatrix = new double\*[Size];

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

pMatrix[i] = new double[Size];

}

pVector = new double[Size];

pResult = new double[Size];

// Initialization of the matrix and the vector elements

if (ProcId == 0) {

//DummyDataInitialization(pMatrix, pVector, Size);

RandomDataInitialization(pMatrix, pVector, Size);

}

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

MPI\_Bcast(&(pMatrix[i][0]), Size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&(pVector[0]), Size, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

}

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 ][ j]);

printf("\n");

}

}

// Function for formatted vector output

void PrintVector(double\* pVector, int Size) {

int i;

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

printf("%7.4f ", pVector[i]);

}

// Column elimination

void ParallelColumnElimination(double\*\* pMatrix, double\* pVector,

int Pivot, int Iter, int Size, int NProc, int ProcId) {

double PivotValue, PivotFactor;

int n1 = Size / NProc;

int n2 = (ProcId + 1) \* n1;

if (NProc == ProcId + 1) {

n2 = Size;

}

int st = ProcId \* n1;

int h1 = (Pivot + n1) / n1 - 1;

if (h1 >= NProc) h1 = NProc - 1;

MPI\_Bcast(&(pMatrix[Pivot][0]), Size, MPI\_DOUBLE, h1, MPI\_COMM\_WORLD);

MPI\_Bcast(&(pVector[Pivot]), 1, MPI\_DOUBLE, h1, MPI\_COMM\_WORLD);

PivotValue = pMatrix[Pivot][Iter];

//#pragma omp parallel for private(PivotFactor) schedule(dynamic,1)

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

int h = (i + n1) / n1 - 1;

if (h >= NProc) h = NProc - 1;

if (i >= st && i < n2) {

if (pPivotIter[i] == -1) {

PivotFactor = pMatrix[i][Iter] / PivotValue;

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

pMatrix[i][j] -= PivotFactor \* pMatrix[Pivot][j];

}

pVector[i] -= PivotFactor \* pVector[Pivot];

}

}

//MPI\_Bcast(&(pMatrix[i][0]), Size, MPI\_DOUBLE, h, MPI\_COMM\_WORLD);

//MPI\_Bcast(&(pVector[i]), 1, MPI\_DOUBLE, h, MPI\_COMM\_WORLD);

}

}

// Gaussian elimination

void ParallelGaussianElimination(double\*\* pMatrix, double\* pVector,

int Size, int NProc, int ProcId) {

int Iter; // The number of the iteration of the Gaussian

// elimination

int PivotRow; // The number of the current pivot row

// cout << endl << endl;

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

//PrintMatrix(pMatrix, Size, Size);

// cout<<Iter<<endl;

// Finding the pivot row

PivotRow = ParallelFindPivotRow(pMatrix, Size, Iter, NProc, ProcId);

pPivotPos[Iter] = PivotRow;

pPivotIter[PivotRow] = Iter;

ParallelColumnElimination(pMatrix, pVector, PivotRow, Iter, Size, NProc, ProcId);

}

}

// Function for computational process termination

void ProcessTermination(double\*\* pMatrix, double\* pVector, double\* pResult, int Size) {

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

delete[] pMatrix[i];

}

delete[] pMatrix;

delete[] pVector;

delete[] pResult;

}

// Back substation

void ParallelBackSubstitution(double\*\* pMatrix, double\* pVector,

double\* pResult, int Size) {

int RowIndex, Row;

for (int i = Size - 1; i >= 0; i--) {

RowIndex = pPivotPos[i];

pResult[i] = pVector[RowIndex] / pMatrix[RowIndex][ i];

//#pragma omp parallel for private (Row)

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

Row = pPivotPos[j];

pVector[Row] -= pMatrix[Row][ i] \* pResult[i];

pMatrix[Row][ i] = 0;

}

}

}

// Function for the execution of Gauss algorithm

void ParallelResultCalculation(double\*\* pMatrix, double\* pVector,

double\* pResult, int Size, int NProc, int ProcId) {

// Memory allocation

pPivotPos = new int[Size];

pPivotIter = new int[Size];

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

pPivotIter[i] = -1;

}

ParallelGaussianElimination(pMatrix, pVector, Size, NProc, ProcId);

if (ProcId == 0) {

ParallelBackSubstitution(pMatrix, pVector, pResult, Size);

}

// Memory deallocation

delete[] pPivotPos;

delete[] pPivotIter;

}

// Function for testing the result

void TestResult(double\*\* pMatrix, double\* pVector,

double\* pResult, int Size) {

/\* Buffer for storing the vector, that is a result of multiplication

of the linear system matrix by the vector of unknowns \*/

double\* pRightPartVector;

// Flag, that shows wheather the right parts

// vectors are identical or not

int equal = 0;

double Accuracy = 1.e-2; // Comparison accuracy

pRightPartVector = new double[Size];

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

pRightPartVector[i] = 0;

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

pRightPartVector[i] +=

pMatrix[i ][j] \* pResult[j];

}

}

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

// std::cout << pRightPartVector[i] << " " << pVector[i] << " " << fabs(pRightPartVector[i] - pVector[i]) << std::endl;

if (fabs(pRightPartVector[i] - pVector[i]) > Accuracy)

equal = 1;

}

if (equal == 1)

printf("The result of the parallel Gauss algorithm is NOT correct."

"Check your code.");

else

printf("The result of the parallel Gauss algorithm is correct.");

delete[] pRightPartVector;

}

int main() {

MPI\_Init(NULL, NULL);

std::srand(std::time(nullptr));

double\*\* pMatrix;

double\* pVector;

double\* pResult;

int Size = 3000;

double start, finish, duration;

// Data initialization

int NProc, ProcId;

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

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

ProcessInitialization(pMatrix, pVector, pResult, Size, NProc, ProcId); start = omp\_get\_wtime();

//PrintMatrix(pMatrix, Size, Size);

ParallelResultCalculation(pMatrix, pVector, pResult, Size, NProc,ProcId); finish = omp\_get\_wtime();

duration = finish - start;

if (ProcId == 0) {

//PrintVector(pResult, Size);

//std::cout << std::endl;

// Testing the result

//TestResult(pMatrix, pVector, pResult, Size);

// Printing the time spent by parallel Gauss algorithm

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

// Program termination

//ProcessTermination(pMatrix, pVector, pResult, Size);

}

MPI\_Finalize();

return 0;}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер теста | Порядок системы | Последовательный алгоритм | Параллельный алгоритм | |
| Время | Ускорение |
| 1 | 10 | 0.000581 | 0.001454 | 0.39996 |
| 2 | 100 | 0.009 | 0.007544 | 1.193 |
| 3 | 500 | 1.979716 | 0.248342 | 7.971 |
| 4 | 1000 | 5.654 | 2.102299 | 2.6894 |
| 5 | 1500 | 17.472211 | 5.918055 | 2.9523 |
| 6 | 2000 | 42.462686 | 14.092079 | 3.0113 |
| 7 | 2500 | 76.46982 | 27.155705 | 2.816 |
| 8 | 3000 | 133.560 | 46.991220 | 2.8422 |

Основное падение производительности параллельной версии происходит при обмене сообщениями между потоками. А так как при росте размерности значительно увеличивается количество таких передач, то и размер ускорение так же уменьшается, однако остается почти трехкратным. Параллельная реализация обратного хода ускорения не приносит, поэтому в коде она не представлена.