МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ФакультетКНТ

КафедраПМИ

Индивидуальная работа по курсу

«Параллельные информационные системы»

На тему: «Матричное умножение ленточной разреженной матрицы»

Руководители: Выполнил: доцент каф. ПМИ ст. гр. ИПОИм

Назарова И. А. Лысенко А. С.

Покровск 2017

Постановка задачи

1. Одна из основных задач матричных вычислений — умножение разреженных матриц. Результатом перемножения матриц **А** и **B** является матрица **С**, каждый элемент которой есть скалярное произведение соответствующих строк матрицы **A** и столбцов матрицы **B**. В данной работе ставится задача результаты работы параллельного алгоритма перемножения двух матриц, заданных произвольным образом с помощью треугольного распределения.

Параллельный алгоритм

2. Из определения операции матричного умножения следует, что вычисление всех элементов матрицы С может быть выполнено независимо друг от друга. Как результат, возможный подход для организации параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы С. Для проведения всех необходимых вычислений каждая подзадача должна содержать по одной строке матрицы А и одному столбцу матрицы В. Общее количество получаемых при таком подходе подзадач оказывается равным n2(по числу элементов матрицы С).

Алгоритм представляет собой итерационную процедуру, количество итераций которой совпадает с числом подзадач. На каждой итерации алгоритма каждая подзадача содержит по одной строке матрицы А и одному столбцу матрицы В. При выполнении итерации проводится скалярное умножение содержащихся в подзадачах строк и столбцов, что приводит к получению соответствующих элементов результирующей матрицы С. По завершении вычислений в конце каждой итерации столбцы матрицы В должны быть переданы между подзадачами с тем, чтобы в каждой подзадаче оказались новые столбцы матрицы В и могли быть вычислены новые элементы матрицы C. При этом данная передача столбцов между подзадачами должна быть организована таким образом, чтобы после завершения итераций алгоритма в каждой подзадаче последовательно оказались все столбцы матрицы В.  
Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. Когда размер матриц n оказывается больше, чем число процессоров p, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк и столбцов перемножаемых матриц. В этом случае исходная матрица A разбивается на ряд горизонтальных полос, а матрица B представляется в виде набора вертикальных полос. Размер полос при этом следует выбрать равным k=n/p (в предположении, что n кратно p), что позволит по-прежнему обеспечить равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

Под треугольным методом подразумевается способ генерации случайных чисел в данных матрицах. Метод заключается в том, что при увеличении данных, числа со значением ноль, выпадают с большей вероятностью, чем остальные. В данном случае мы берем случайную величину rand() и задаем ей распределение, исходя из функции треугольного распределения. Для параметров a = -7, b = 7, c = 0 график 10000000 измерений будет выглядеть так

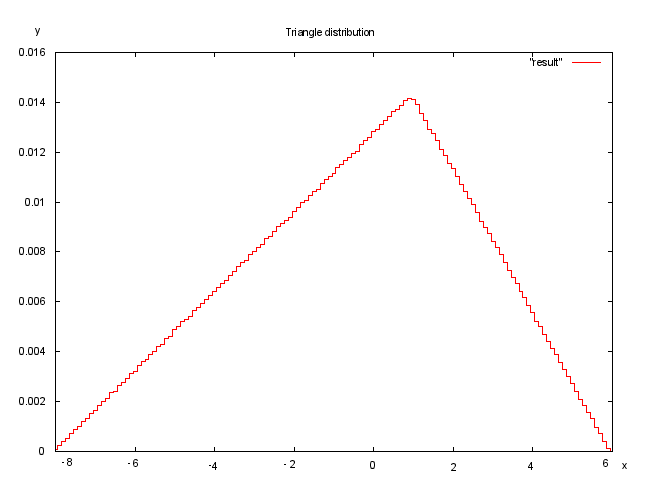


Рисунок 2.1 – Вероятность генерации значений в матрице

Анализ эффективности

3. Общая трудоемкость последовательного алгоритма является пропорциональной **n3**. Для параллельного алгоритма на каждой итерации каждый процессор выполняет умножение имеющихся на процессоре полос матрицы А и матрицы В (размер полос равен n/p, и, как результат, общее количество выполняемых при этом умножении операций равно n3/p2). Поскольку число итераций алгоритма совпадает с количеством процессоров, сложность параллельного алгоритма без учета затрат на передачу данных может быть определена при помощи выражения **T=(n3)/p**.   
C учетом этой оценки, показатель ускорения примет вид: **A=(n3)/((n3)/p=p.**

**Демонстрация**

**4. Вычисление диагональных элементов.**

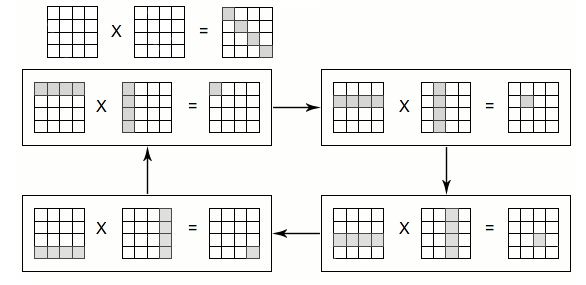


Рисунок 4.1 - **Первая итерация.**

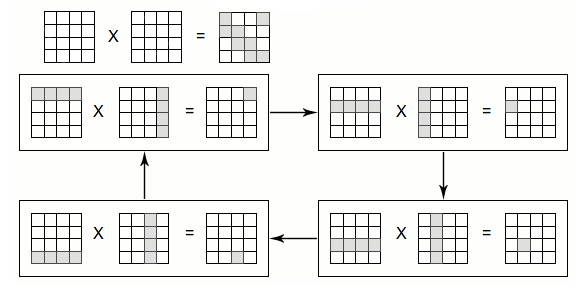


Рисунок 4.2 - **Вторая итерация.**

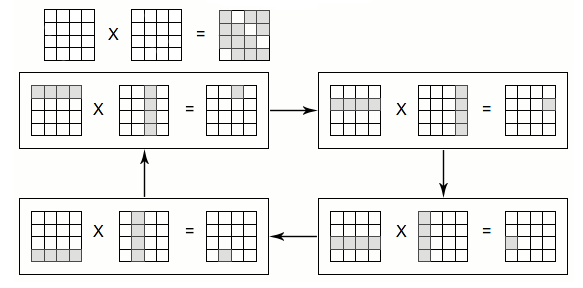


Рисунок 4.3 - **Третья итерация.**

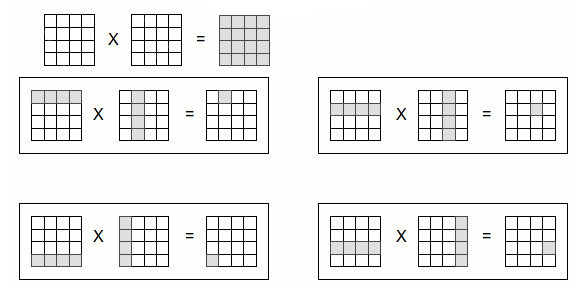


Рисунок 4.4 - **Четвертая итерация.**

Результаты вычисления экспериментов

5. Таблица сравнения последовательного и параллельного алгоритма (время работы)

Таблица 5.1 Сравнение работы алгоритмов

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы | Последовательный алгоритм (время работы) | Параллельный алгоритм (время работы) | Ускорение |
| 500 | 1,4218 | 0,3625 | 3,9222 |
| 1000 | 11,7435 | 2,9056 | 4,0416 |
| 1500 | 38,2275 | 9,7706 | 3,9125 |
| 2000 | 91,7441 | 23,1699 | 3,9596 |

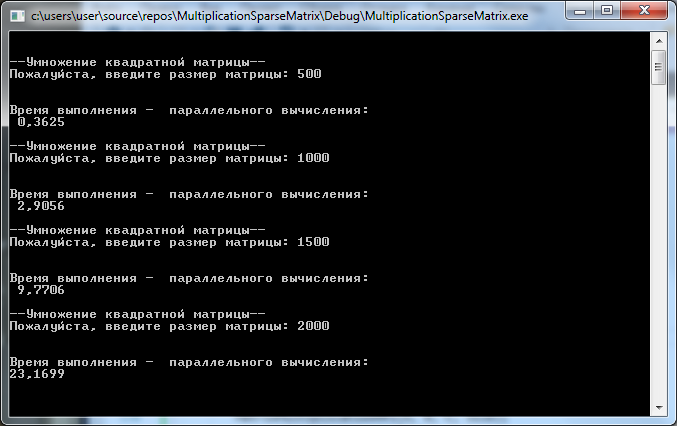


Рисунок 5.1 – Результат работы программы

Листинг программы на Си

#include<mpi.h>

#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

#include<math.h>

#include<time.h>

#include <clocale>

int ProcNum;

int ProcRank;

int flag = 0;

int Size;

double \*A; double \*B; double \*C;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

printf("\n");

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

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

}

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int sparseMatrixGeneration(int a, int b, int c)

{

double U = rand() / (double)RAND\_MAX;

double F = (c - a) / (b - a);

if (U <= F)

return a + sqrt(U \* (b - a) \* (c - a));

else

return b - sqrt((1 - U) \* (b - a) \* (b - c));

}

//------------------------------------------------------------

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

srand(100);

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

for (int j = 0; j<Size; j++) pMatrix[i\*Size + j] = sparseMatrixGeneration(-7, 7, 0) / double(1000);

}

}

//-------------------------------------------------

void InitProcess(double\* &A, double\* &B, double\* &C, int &Size) {

setlocale(LC\_ALL, "Rus");

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

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

if (ProcRank == 0) {

do {

printf("\n--Умножение квадратной матрицы--");

printf("\nПожалуйста, введите размер матрицы: "); scanf\_s("%d", &Size);

if (Size< ProcNum) printf("Matrix size is less than the number of processes! \n");

if (Size%ProcNum != 0) printf("Matrix size should be dividable by the number of processes! \n");

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

}

if (Size<10) flag = 1;

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

if (ProcRank == 0) {

A = new double[Size\*Size];

B = new double[Size\*Size];

C = new double[Size\*Size];

RandInit(A, Size); RandInit(B, Size);

}

}

//-------------------------------------------------

void Flip(double \*&B, int dim) {

double temp = 0.0;

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

for (int j = i + 1; j<dim; j++) {

temp = B[i\*dim + j]; B[i\*dim + j] = B[j\*dim + i]; B[j\*dim + i] = temp;

}

}

}

//-------------------------------------------------

void MatrixMultiplicationMPI(double \*&A, double \*&B, double \*&C, int &Size) {

int dim = Size;

int i, j, k, p, ind;

double temp;

MPI\_Status Status;

int ProcPartSize = dim / ProcNum;

int ProcPartElem = ProcPartSize \* dim;

double\* bufA = new double[dim\*ProcPartSize];

double\* bufB = new double[dim\*ProcPartSize];

double\* bufC = new double[dim\*ProcPartSize];

int ProcPart = dim / ProcNum, part = ProcPart \* dim;

if (ProcRank == 0) {

Flip(B, Size);

}

MPI\_Scatter(A, part, MPI\_DOUBLE, bufA, part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(B, part, MPI\_DOUBLE, bufB, part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

temp = 0.0;

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

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

for (k = 0; k<dim; k++) temp += bufA[i\*dim + k] \* bufB[j\*dim + k];

bufC[i\*dim + j + ProcPartSize \* ProcRank] = temp; temp = 0.0;

}

}

int NextProc; int PrevProc;

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

NextProc = ProcRank + 1;

if (ProcRank == ProcNum - 1) NextProc = 0;

PrevProc = ProcRank - 1;

if (ProcRank == 0) PrevProc = ProcNum - 1;

MPI\_Sendrecv\_replace(bufB, part, MPI\_DOUBLE, NextProc, 0, PrevProc, 0, MPI\_COMM\_WORLD, &Status);

temp = 0.0;

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

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

for (k = 0; k<dim; k++) {

temp += bufA[i\*dim + k] \* bufB[j\*dim + k];

}

if (ProcRank - p >= 0)

ind = ProcRank - p;

else ind = (ProcNum - p + ProcRank);

bufC[i\*dim + j + ind \* ProcPartSize] = temp;

temp = 0.0;

}

}

}

MPI\_Gather(bufC, ProcPartElem, MPI\_DOUBLE, C, ProcPartElem, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

delete[]bufA;

delete[]bufB;

delete[]bufC;

}

//--------------------------------------------------------

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

setlocale(LC\_ALL, "Rus");

double beg, end, serial, parallel = 0;

MPI\_Init(&argc, &argv);

for (int i = 0; i < 4; i++)

{

printf("\n");

InitProcess(A, B, C, Size);

beg = MPI\_Wtime();

MatrixMultiplicationMPI(A, B, C, Size);

end = MPI\_Wtime(); parallel = end - beg;

if (ProcRank == 0) {

printf("\n", &ProcNum);

printf("\nВремя выполнения - параллельного вычисления:\n");

printf("%7.4f", parallel);

if (flag) {

printf("\nМатрица C - параллельного вычисления\n");

PrintMatrix(C, Size);

printf("\n");

}

}

}

MPI\_Finalize();

delete[] A; delete[] B; delete[] C;

\_getch();

}