Федеральное агентство по образованию Российской Федерации

Государственное образовательное учреждение

высшего профессионального образования

Нижегородский государственный университет им. Н.И. Лобачевского

Факультет Вычислительной математики и кибернетики

Отчёт по лабораторной работе

Умножение разреженных матриц с комплексными элементами

Выполнил:

студент ф-та ВМК гр. 84-03

Савичев М.Р.

Проверил:

ассистент каф. МО ЭВМ, ВМК

Кустикова В.Д.

Нижний Новгород

2015 г.

Содержание

[Постановка задачи 3](#_Toc418630224)

[Описание алгоритма 4](#_Toc418630225)

[Алгоритм транспонирования 4](#_Toc418630226)

[Алгоритм умножения матриц 5](#_Toc418630227)

[Схема распараллеливания 6](#_Toc418630228)

[Программная реализация 7](#_Toc418630229)

[Последовательная версия 7](#_Toc418630230)

[Реализация с помощью OpenMP 8](#_Toc418630231)

[Реализация с помощью TBB 9](#_Toc418630232)

[Реализация с помощью MPI + OpenMP 11](#_Toc418630233)

[Проверка корректности 14](#_Toc418630234)

[Результаты экспериментов 15](#_Toc418630235)

[Выводы 17](#_Toc418630236)

# Постановка задачи

Разработать приложение, реализующее алгоритм умножения разреженных матриц с комплексными элементами. Матрицы должны храниться в строковым CRS формате. Исходными данными будут являться – порядок матрицы и количество ненулевых элементов в строке.

Необходимо реализовать последовательную версию алгоритма. Далее распараллелить её с помощью OpenMP и Intel Threading Building Blocks (TBB) для выполнения на системах с общей памятью. А также с помощью интерфейса для передачи информации MPI для систем с распределенной памятью.

Помимо этого, в постановку задачи входит реализация автоматизированного анализа корректности путем сравнения алгоритма с эталонной версией из библиотеки Intel Math Kernel Library (MKL).

И последнее, необходимо провести эксперименты с целью оценки масштабируемости каждого из версий алгоритма.

# Описание алгоритма

Строковый CRS формат хранения разреженной матрицы с комплексными элементами представляет собой четыре массива:

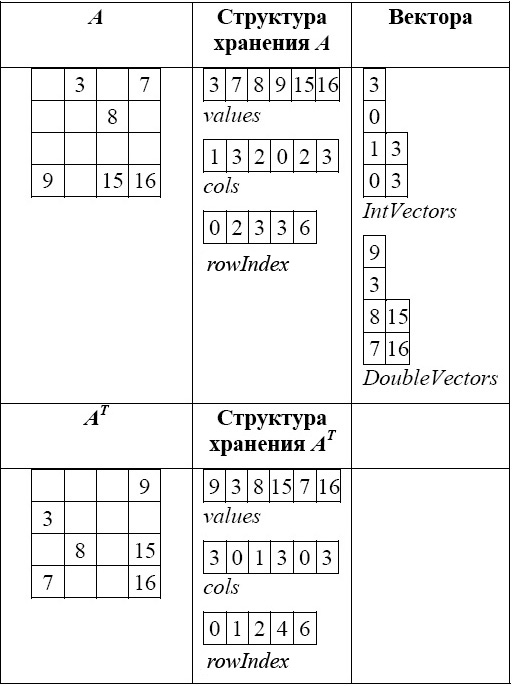
* valuesRe - массив для хранения действительных частей ненулевых элементов матрицы
* valuesIm – массив для хранения мнимый частей ненулевых элементов матрицы
* cols – массив для хранения номеров столбцов для соответствующих элементов
* rowIndex – массив для указателей позиций, с которых начинается строка (i-я строка начинается с элемента с номером rowIndex[i] до rowIndex[i]-1 включительно

Из-за такого необычного хранения матриц, нельзя воспользоваться обычным алгоритмом умножения матриц (строчка на столбец), потому что, чтобы получить столбец, необходимо просмотреть всю матрицу, что приводит к квадратичной трудоемкости алгоритма. Для того, чтобы решить данную проблему, мы перед тем, как умножать матрицы, применим транспонирование.

## Алгоритм транспонирования

Будем формировать результирующую матрицу построчно. Для этого можно брать столбцы исходной матрицы и создавать из них строки результирующей матрицы. Считаем, что размерность матрицы – N, и, для простоты, её элементы вещественные.

1. Сформируем N одномерных векторов для хранения целых чисел (IntVectors), а также N векторов для хранения вещественных чисел (DoubleVectors). N в данном случае соответствует числу столбцов исходной матрицы.
2. В цикле просмотрим все строки исходной матрицы, для каждой строки – все ее элементы. Пусть текущий элемент находится в строке i, столбце j, его значение равно v. Тогда добавим числа i и v в j-ые вектора для хранения целых и комплексных чисел (соответственно). Тем самым в векторах мы сформируем строки транспонированной матрицы.
3. Последовательно скопируем данные из векторов в CRS структуру транспонированной матрицы (сols и values), попутно формируя массив rowIndex.



1. Транспонирование матрицы

## Алгоритм умножения матриц

После того, как вторая матрица транспонирована, нетрудно провести само умножение, производя скалярное умножение «строки на строку». Однако остается вопрос, как же проводить скалярное умножение разреженных векторов. Заметим, что умножаемый вектор является упорядоченным в соответствии с выбранной структурой хранения матрицы. Это означает, что может быть применен алгоритм похожий на слияние двух отсортированных массивов с сохранением порядка. Алгоритм для вещественных матриц выглядит следующим образом.

1. Встать на начало обоих векторов (ks **= …**, **ls = …**).
2. Сравнить текущие элементы **A.сol[ks]** и **B.сol[ls]**. Если значения совпадают, накопить в сумму произведение **A.values[ks] \* B.values[ls]** и увеличить оба индекса, в противном случае – увеличить один из индексов, в зависимости от того, какое значение больше (например, **A.col[ks] > B.col[ls]** → **ls++**).

Шаг 2 выполняется до тех пор, пока не кончатся элементы хотя бы в одном из векторов.

Для того, чтобы применить этот алгоритм для комплексных матриц, нужно изменить лишь способ вычисления произведения.

# Схема распараллеливания

В связи с тем, что каждая строка первой матрицы А в процессе умножения на матрицу В должна быть умножена на всю матрицу В, самым наивным способом параллелизма является распределение строк матрицы А между потоками. Такое распараллеливание возможно, так как со строками матрицы А мы работаем независимо.

Таким образом, для реализации параллелизма с помощью OpenMP или TBB достаточно лишь продублировать независимые промежуточные переменный и дописать программные директивы.

При использовании технологии MPI нулевому процессу необходимо передать части матрицы А другим процессам, а также всю матрицу В. После умножения частей матрицы А на матрицу В, первый процесс принимает произведения и собирает всё в единую результирующую матрицу произведения.

При использовании комбинации технологий (например MPI + OpenMP), можно цикл умножения в MPI версии распараллелить точно так же, как в версии с использованием OpenMP.

# Программная реализация

## Последовательная версия

1. Функция транспонирования матрицы

void transposeCRSComplexMatrix(CRSComplexMatrix M, CRSComplexMatrix &out, double &time)

{

clock\_t start = clock();

InitializeCRSComplexMatrix(M.N, M.NZ, out);

out.rowIndex[0] = 0;

for (int i = 0; i < M.N + 1; i++)

out.rowIndex[i] = 0;

for (int i = 0; i < M.NZ; i++)

out.rowIndex[M.col[i] + 1]++;

int S = 0, tmp = 0;

for (int i = 1; i <= M.N; i++)

{

tmp = out.rowIndex[i];

out.rowIndex[i] = S;

S = S + tmp;

}

int Col, IIndex, RIndex;

double VRe, VIm;

for (int i = 0; i < M.N; i++)

{

Col = i;

for (int j = M.rowIndex[i]; j < M.rowIndex[i + 1]; j++)

{

VRe = M.valueRe[j];

VIm = M.valueIm[j];

RIndex = M.col[j];

IIndex = out.rowIndex[RIndex + 1];

out.valueRe[IIndex] = VRe;

out.valueIm[IIndex] = VIm;

out.col[IIndex] = Col;

out.rowIndex[RIndex + 1]++;

}

}

clock\_t finish = clock();

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

}

2. Функция умножения матриц

int multiplicateCRSComplexMatrix(CRSComplexMatrix A, CRSComplexMatrix B, CRSComplexMatrix &C, double &time)

{

if (A.N != B.N)

return 1;

int N = A.N;

std::vector<int> columns;

std::vector<double> valuesRe;

std::vector<double> valuesIm;

std::vector<int> row\_index;

clock\_t start = clock();

int NZ = 0;

row\_index.push\_back(0);

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

{

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

{

double sumRe, sumIm ;

int ks = A.rowIndex[i];

int ls = B.rowIndex[j];

int kf = A.rowIndex[i + 1] - 1;

int lf = B.rowIndex[j + 1] - 1;

while ((ks <= kf) && (ls <= lf))

{

if (A.col[ks] < B.col[ls])

ks++;

else

if (A.col[ks] > B.col[ls])

ls++;

else

{

sumRe += A.valueRe[ks] \* B.valueRe[ls] - A.valueIm[ks] \* B.valueIm[ls];

sumIm += A.valueIm[ks] \* B.valueRe[ks] + A.valueRe[ks] \* B.valueIm[ks];

ks++;

ls++;

}

}

if (fabs(sumRe) > ZERO\_IN\_CRS || fabs(sumIm) > ZERO\_IN\_CRS)

{

columns.push\_back(j);

valuesRe.push\_back(sumRe);

valuesIm.push\_back(sumIm);

NZ++;

}

}

row\_index.push\_back(NZ);

}

InitializeCRSComplexMatrix(N, NZ, C);

for (unsigned int j = 0; j < columns.size(); j++)

{

C.col[j] = columns[j];

C.valueRe[j] = valuesRe[j];

C.valueIm[j] = valuesIm[j];

}

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

C.rowIndex[i] = row\_index[i];

clock\_t finish = clock();

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

return 0;

}

## Реализация с помощью OpenMP

int multiplicateCRSComplexMatrixOMP(CRSComplexMatrix A, CRSComplexMatrix B, CRSComplexMatrix &C, double &time, int numThreads)

{

if (A.N != B.N)

return 1;

int N = A.N;

int j;

clock\_t start = clock();

std::vector<int>\* columns = new std::vector<int>[N];

std::vector<double> \*valuesRe = new std::vector<double>[N];

std::vector<double> \*valuesIm = new std::vector<double>[N];

int\* row\_index = new int[N + 1];

memset(row\_index, 0, sizeof(int) \* N);

omp\_set\_num\_threads(numThreads);

#pragma omp parallel for private(j)

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

{

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

{

double sumRe = 0, sumIm = 0;

int ks = A.rowIndex[i];

int ls = B.rowIndex[j];

int kf = A.rowIndex[i + 1] - 1;

int lf = B.rowIndex[j + 1] - 1;

while ((ks <= kf) && (ls <= lf))

{

if (A.col[ks] < B.col[ls])

ks++;

else

if (A.col[ks] > B.col[ls])

ls++;

else

{

sumRe += A.valueRe[ks] \* B.valueRe[ls] - A.valueIm[ks] \* B.valueIm[ls];

sumIm += A.valueIm[ks] \* B.valueRe[ls] + A.valueRe[ks] \* B.valueIm[ls];

ks++;

ls++;

}

}

if (fabs(sumRe) > ZERO\_IN\_CRS || fabs(sumIm) > ZERO\_IN\_CRS)

{

columns[i].push\_back(j);

valuesRe[i].push\_back(sumRe);

valuesIm[i].push\_back(sumIm);

row\_index[i]++;

}

}

}

int NZ = 0;

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

{

int tmp = row\_index[i];

row\_index[i] = NZ;

NZ += tmp;

}

row\_index[N] = NZ;

InitializeCRSComplexMatrix(N, NZ, C);

int count = 0;

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

{

int size = columns[i].size();

memcpy(&C.col[count], &columns[i][0], size \* sizeof(int));

memcpy(&C.valueRe[count], &valuesRe[i][0], size \* sizeof(double));

memcpy(&C.valueIm[count], &valuesIm[i][0], size \* sizeof(double));

count += size;

}

memcpy(C.rowIndex, &row\_index[0], (N + 1) \* sizeof(int));

delete [] row\_index;

delete [] columns;

delete [] valuesRe;

delete [] valuesIm;

clock\_t finish = clock();

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

return 0;

}

## Реализация с помощью TBB

Необходимо сказать, что идейно реализация в этом случае ничем не отличается от реализации с помощью OpenMP. Отличия лишь в программных директивах. В случае TBB, нужно создать отдельный класс с перегруженной функцией operator(), в которой будет скрыт цикл умножения, и использовать его при вызове шаблонной функции parallel\_for().

class MultiplicatorCRSComplexMatrix

{

CRSComplexMatrix A, B;

std::vector<int>\* columns;

std::vector<double>\* valuesRe;

std::vector<double>\* valuesIm;

int \*row\_index;

public:

MultiplicatorCRSComplexMatrix(CRSComplexMatrix& \_A, CRSComplexMatrix& \_B, std::vector<int>\* &\_columns, std::vector<double>\* &\_valuesRe, std::vector<double>\* &\_valuesIm , int \*\_row\_index)

: A(\_A), B(\_B), columns(\_columns), valuesRe(\_valuesRe), valuesIm(\_valuesIm), row\_index(\_row\_index)

{

}

void operator()(const tbb::blocked\_range<int>& r) const

{

int begin = r.begin();

int end = r.end();

int N = A.N; int i, j, k;

int \*temp = new int[N];

for (i = begin; i < end; i++)

{

memset(temp, -1, N \* sizeof(int));

int ind1 = A.rowIndex[i], ind2 = A.rowIndex[i + 1];

for (j = ind1; j < ind2; j++)

{

int col = A.col[j];

temp[col] = j;

}

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

{

double sumRe = 0, sumIm = 0;

int ind3 = B.rowIndex[j],

ind4 = B.rowIndex[j + 1];

for (k = ind3; k < ind4; k++)

{

int bcol = B.col[k];

int aind = temp[bcol];

if (aind != -1)

{

sumRe += A.valueRe[aind] \* B.valueRe[k] - A.valueIm[aind] \* B.valueIm[k];

sumIm += A.valueIm[aind] \* B.valueRe[k] + A.valueRe[aind] \* B.valueIm[k];

}

}

if (fabs(sumRe) > ZERO\_IN\_CRS || fabs(sumIm) > ZERO\_IN\_CRS)

{

columns[i].push\_back(j);

valuesRe[i].push\_back(sumRe);

valuesIm[i].push\_back(sumIm);

row\_index[i]++;

}

}

} delete [] temp;

}

};

int multiplicateCRSComplexMatrixTBB(CRSComplexMatrix A, CRSComplexMatrix B, CRSComplexMatrix &C, double &time)

{

if (A.N != B.N)

return 1;

int N = A.N;

int i, j, k;

clock\_t start = clock();

std::vector<int>\* columns = new std::vector<int>[N];

std::vector<double> \*valuesRe = new std::vector<double>[N];

std::vector<double> \*valuesIm = new std::vector<double>[N];

int\* row\_index = new int[N + 1];

memset(row\_index, 0, sizeof(int) \* N);

int grainsize = 10;

tbb::parallel\_for(tbb::blocked\_range<int>(0, A.N, grainsize), MultiplicatorCRSComplexMatrix (A, B, columns, valuesRe, valuesIm, row\_index));

int NZ = 0;

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

{

int tmp = row\_index[i];

row\_index[i] = NZ;

NZ += tmp;

}

row\_index[N] = NZ;

InitializeCRSComplexMatrix(N, NZ, C);

int count = 0;

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

{

int size = columns[i].size();

memcpy(&C.col[count], &columns[i][0], size \* sizeof(int));

memcpy(&C.valueRe[count], &valuesRe[i][0], size \* sizeof(double));

memcpy(&C.valueIm[count], &valuesIm[i][0], size \* sizeof(double));

count += size;

}

memcpy(C.rowIndex, &row\_index[0], (N + 1) \* sizeof(int));

delete [] row\_index;

delete [] columns;

delete [] valuesRe;

delete [] valuesIm;

clock\_t finish = clock();

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

return 0;

}

## Реализация с помощью MPI + OpenMP

В этом случае первый процесс в начале определяет, сколько строк матрицы А нужно передать каждом процессу. В нашем случае необходимо, чтобы порядок матриц нацело делился на количество процессов.

//Вычисление количества строк для каждого процесса и отправка процессам

int rowsToWorkWith = N / Procs;

int nzElementsToWorkWith;

MPI\_Status status;

if (Rank == 0)

{

nzElementsToWorkWith = A.rowIndex[rowsToWorkWith] - A.rowIndex[0];

for (int i = 1; i < Procs; i++)

{

nzElementsToWorkWith = A.rowIndex[(i+1)\*rowsToWorkWith] - A.rowIndex[i\*rowsToWorkWith];

MPI\_Send(&nzElementsToWorkWith, 1, MPI\_INT, i, NZ\_ELEMENTS, MPI\_COMM\_WORLD);

}

}

else

{

MPI\_Recv(&nzElementsToWorkWith, 1, MPI\_INT, 0, NZ\_ELEMENTS, MPI\_COMM\_WORLD, &status);

}

double \*values1Re = new double[nzElementsToWorkWith];

double \*values1Im = new double[nzElementsToWorkWith];

double \*values2Re = new double[NZ];

double \*values2Im = new double[NZ];

int\* cols1 = new int[nzElementsToWorkWith];

int\* cols2 = new int[NZ];

int\* rowInd1 = new int[rowsToWorkWith+1];

int\* rowInd2 = new int[N + 1];

//Отправка данных процессам

if (Rank == 0)

{

for (int i = 1; i < Procs; i++)

{

MPI\_Send(&A.valueRe[A.rowIndex[i\*rowsToWorkWith]], A.rowIndex[(i+1)\*rowsToWorkWith] - A.rowIndex[i\*rowsToWorkWith], MPI\_DOUBLE, i, VALUESRE\_SEND\_1, MPI\_COMM\_WORLD);

MPI\_Send(&A.valueIm[A.rowIndex[i\*rowsToWorkWith]], A.rowIndex[(i+1)\*rowsToWorkWith] - A.rowIndex[i\*rowsToWorkWith], MPI\_DOUBLE, i, VALUESIM\_SEND\_1, MPI\_COMM\_WORLD);

MPI\_Send(BT.valueRe, NZ, MPI\_DOUBLE, i, VALUESRE\_SEND\_2, MPI\_COMM\_WORLD);

MPI\_Send(BT.valueIm, NZ, MPI\_DOUBLE, i, VALUESIM\_SEND\_2, MPI\_COMM\_WORLD);

MPI\_Send(&A.col[A.rowIndex[i\*rowsToWorkWith]], A.rowIndex[(i+1)\*rowsToWorkWith] - A.rowIndex[i\*rowsToWorkWith], MPI\_INT, i, COLS\_SEND\_1, MPI\_COMM\_WORLD);

MPI\_Send(BT.col, NZ, MPI\_INT, i, COLS\_SEND\_2, MPI\_COMM\_WORLD);

MPI\_Send(&A.rowIndex[i], rowsToWorkWith + 1, MPI\_INT, i, ROW\_IND\_SEND\_1, MPI\_COMM\_WORLD);

MPI\_Send(BT.rowIndex, N + 1, MPI\_INT, i, ROW\_IND\_SEND\_2, MPI\_COMM\_WORLD);

}

}

if (Rank == 0)

{

memcpy(values1Re, &A.valueRe[A.rowIndex[0]], sizeof(double)\*nzElementsToWorkWith);

memcpy(values1Im, &A.valueIm[A.rowIndex[0]], sizeof(double)\*nzElementsToWorkWith);

memcpy(values2Re, BT.valueRe, sizeof(double)\*NZ);

memcpy(values2Im, BT.valueIm, sizeof(double)\*NZ);

memcpy(cols1, &A.col[A.rowIndex[0]], sizeof(int)\*nzElementsToWorkWith);

memcpy(cols2, BT.col, sizeof(int)\*NZ);

memcpy(rowInd1, &A.rowIndex[0], sizeof(int)\*(rowsToWorkWith + 1));

memcpy(rowInd2, BT.rowIndex, sizeof(int)\*(N + 1));

}

else

{

MPI\_Recv(values1Re, nzElementsToWorkWith, MPI\_DOUBLE, 0, VALUESRE\_SEND\_1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(values1Im, nzElementsToWorkWith, MPI\_DOUBLE, 0, VALUESIM\_SEND\_1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(values2Re, NZ, MPI\_DOUBLE, 0, VALUESRE\_SEND\_2, MPI\_COMM\_WORLD, &status);

MPI\_Recv(values2Im, NZ, MPI\_DOUBLE, 0, VALUESIM\_SEND\_2, MPI\_COMM\_WORLD, &status);

MPI\_Recv(cols1, nzElementsToWorkWith, MPI\_INT, 0, COLS\_SEND\_1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(cols2, NZ, MPI\_INT, 0, COLS\_SEND\_2, MPI\_COMM\_WORLD, &status);

MPI\_Recv(rowInd1, rowsToWorkWith + 1, MPI\_INT, 0, ROW\_IND\_SEND\_1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(rowInd2, N + 1, MPI\_INT, 0, ROW\_IND\_SEND\_2, MPI\_COMM\_WORLD, &status);

}

//Здесь должен быть параллельный цикл умножения…

//…

//Фрагмент кода сбора данных

MPI\_Gatherv(values3Re, NZnew, MPI\_DOUBLE, resValueRE, rcounts, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Gatherv(values3Im, NZnew, MPI\_DOUBLE, resValueIM, rcounts, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Gatherv(cols3, NZnew, MPI\_INT, resCol, rcounts, displs, MPI\_INT, 0, MPI\_COMM\_WORLD);

if(Rank != 0)

MPI\_Send(rowInd3, rowsToWorkWith + 1, MPI\_INT, 0, COLLECT\_ROW\_IND, MPI\_COMM\_WORLD);

if (Rank == 0)

{

memcpy(resRowInd, rowInd3, sizeof(int)\*(rowsToWorkWith + 1));

int\* rowIndBuffer = new int[rowsToWorkWith + 1];

for(int i = 1; i < Procs; i++)

{

MPI\_Recv(rowIndBuffer, rowsToWorkWith + 1, MPI\_INT, i, COLLECT\_ROW\_IND, MPI\_COMM\_WORLD, &status);

for(int j = 0; j < rowsToWorkWith + 1; j++)

rowIndBuffer[j] += resRowInd[i\*(rowsToWorkWith + 1) - i];

memcpy(&resRowInd[i\*rowsToWorkWith], rowIndBuffer, sizeof(int)\*(rowsToWorkWith + 1));

}

delete rowIndBuffer;

}

# Проверка корректности

Для того, чтобы проверить правильность работы программы, были реализованы две функции. Первая – функция умножения матриц с комплексными элементами с помощью библиотеки MKL, вторая – функция нахождения максимальной разницы между элементами двух матриц.

Эксперименты показали, что всё три реализации алгоритмов умножения комплексных матриц работают корректно.

# Результаты экспериментов

Эксперименты проводились на ЭВМ с четырёхядерным процессором Intel Core i5-3570 3.40 GHz.

Исходные данные: две матрицы размерностью 24000x24000, 5 ненулевых элементов в каждой строке.

Для MPI версии распараллеливание цикла умножения с помощью OpenMP не применялось.

1. Время работы алгоритмов с разным количеством потоков

|  |  |  |  |
| --- | --- | --- | --- |
| Число потоков | OpenMP | TBB | MPI |
| 2 | 7.844 | 7.760 | 7.4736 |
| 3 | 5.097 | 4.997 | 5.1141 |
| 4 | 4.338 | 3.957 | 4.0661 |

1. Значения ускорения для приведенных выше времен работы

|  |  |  |  |
| --- | --- | --- | --- |
| Число потоков | OpenMP | TBB | MPI |
| 2 | 1.9112 | 1.9953 | 1.9966 |
| 3 | 2.9413 | 2.9521 | 2.9178 |
| 4 | 3.4559 | 3.7125 | 3.6699 |

1. Результаты экспериментов для OpenMP

|  |  |  |  |
| --- | --- | --- | --- |
| Порядок матриц | 1 поток | 4 потока | Ускорение |
| 5000 | 0.703 | 0.203 | 3.46 |
| 10000 | 2.761 | 0.774 | 3.56 |
| 20000 | 10.559 | 2.838 | 3.72 |
| 30000 | 24.169 | 6.344 | 3.80 |
| 40000 | 37.945 | 10.141 | 3.74 |

1. Результаты экспериментов для TBB

|  |  |  |  |
| --- | --- | --- | --- |
| Порядок матриц | 1 поток | 4 потока | Ускорение |
| 5000 | 0.703 | 0.187 | 3.75 |
| 10000 | 2.761 | 0.748 | 3.69 |
| 20000 | 10.559 | 2.999 | 3.52 |
| 30000 | 24.169 | 6.403 | 3.77 |
| 40000 | 37.945 | 10.025 | 3.78 |

1. Результаты экспериментов для MPI (OpenMP в один поток)

|  |  |  |  |
| --- | --- | --- | --- |
| Порядок матриц | 1 поток | 4 потока | Ускорение |
| 5000 | 0.708 | 0.266 | 2.66 |
| 10000 | 2.816 | 0.797 | 3.53 |
| 20000 | 10.4434 | 2.762 | 3.78 |
| 30000 | 23.6871 | 6.365 | 3.72 |
| 40000 | 37.4393 | 10.2144 | 3.66 |

Результат эксперимента по запуску MPI версии программы с различным числом потоков и процессов. Матрицы порядка 24000, 5 ненулевых элементов.

1. MPI – 4 процесса, OpenMP – 1 поток. Время: 4.01465
2. MPI – 1 процесс, OpenMP – 4 потока. Время: 4.03515
3. MPI – 2 процесса, OpenMP – 2 потока. Время: 3.9135

# Выводы

Проведенные эксперименты показали, что все три реализации параллелизма ведут себя примерно одинаково с ростом порядка матрицы и изменении количества потоков (процессов). Ускорение с увеличением количества потоков возрастает почти линейно. Значит, алгоритм имеет хорошую масштабируемость.

На реализованном алгоритме довольно трудно определить, какая из трех технологий быстрее. Возможно, это связано с тем, что выбран такой алгоритм умножения разреженных векторов.