МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**Институт информационных технологий, математики и механики**

**Кафедра: математического обеспечения и суперкомпьютерных технологий**

Направление подготовки: Фундаментальная информатика и информационные технологии

**ОТЧЕТ**

по лабораторной работе

на тему:

**«Умножение разреженных матриц в столбцовом формате»**

**Выполнил:** студент группы 381506-3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Панов А.А.

Подпись

**Научный руководитель:**

доцент каф. МОСТ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Сысоев А.В.

Подпись

Нижний Новгород  
2018

**Содержание**

[Введение 3](#_Toc518280605)

[Методы решения систем с разреженной матрицей 3](#_Toc518280606)

[Хранение разреженной матрицы 3](#_Toc518280607)

[Разреженный строчный формат 3](#_Toc518280608)

[Транспонирование матрицы 4](#_Toc518280609)

[Умножение матрицы в строчном формате на матрицу в столбцовом формате 5](#_Toc518280610)

[Постановка задачи 6](#_Toc518280611)

[Руководство пользователя 7](#_Toc518280612)

[Руководство по сборке 7](#_Toc518280613)

[Руководство по запуску 7](#_Toc518280614)

[Руководство программиста 10](#_Toc518280615)

[Описание структуры программы 10](#_Toc518280616)

[Описание структур данных 11](#_Toc518280617)

[class Matrix 11](#_Toc518280618)

[class MatrixCCS 11](#_Toc518280619)

[Параллельное умножение разряженных матриц 11](#_Toc518280620)

[Заключение 12](#_Toc518280621)

[Время работы алгоримов 12](#_Toc518280622)

[Ускорение 12](#_Toc518280623)

[Масштабируемость 12](#_Toc518280624)

[Литература 14](#_Toc518280625)

[Приложение. Фрагменты исходного кода программы 15](#_Toc518280626)

# Введение

## Методы решения систем с разреженной матрицей

Существует много понятий разреженной матрицы, основная идея в том, что в разреженной матрице "много" нулевых элементов. Обычно говорят, что матрица разрежена, если она содержит  отличных от нуля элементов. В противном случае матрица считается плотной. Типичным случаем разреженности является ограниченность числа ненулевых элементов в одной строке от 1 до k, где k значительно меньше n. Задачи линейной алгебры с разреженными матрицами возникают во многих областях, например, при решении дифференциальных уравнений в частных производных, при решении многомерных задач локальной оптимизации. Очевидно, что любую разреженную матрицу можно обрабатывать как плотную, и наоборот. При правильной реализации алгоритмов в обоих случаях будут получены правильные результаты, однако вычислительные затраты будут существенно отличаться. Поэтому приписывание матрице свойства разреженности эквивалентно утверждению о существовании алгоритма, использующего ее разреженность и делающего операции с ней эффективнее по сравнению со стандартными алгоритмами.

Многие алгоритмы, тривиальные для случая плотных матриц, в разреженном случае требуют более тщательного подхода.

## Хранение разреженной матрицы

Существуют различные форматы хранения разреженных матриц. Рассмотрим строчный и столбцовый форматы хранения.

### Разреженный строчный формат

Для хранения матрицы A требуется три одномерных массива:

* массив ненулевых элементов матрицы A, в котором они перечислены по строкам от первой до последней (обозначим его как values);
* массив номеров столбцов для соответствующих элементов массива values (обозначим его как cols);
* массив указателей позиций, с которых начинается описание очередной строки (обозначим его pointer). Описание k-й строки хранится в позициях с pointer[k] по (pointer[k+1]–1) массивов values и cols. Если pointer[k]=pointer[k+1], то k-я строка пустая. Если матрица A состоит из n строк, то длина массива pointer будет n+1.

Данный способ представления также является полным, и упорядоченным, поскольку элементы каждой строки хранятся в соответствии с возрастанием столбцовых индексов.

Для примера рассмотрим представление матрицы

в разреженном строчном формате:

values=(1, -1, -2, 5, 8);

cols=(1, 2, 1, 2, 2);

pointer=(1, 3, 4, 4).

Объем памяти, требуемый для хранения вектора pointer, значительно меньше, чем для хранения вектора rows. Более того, разреженный строчный формат обеспечивает эффективный доступ к строчкам матрицы; доступ к столбцам по-прежнему затруднен. Поэтому предпочтительно использовать этот способ хранения в тех алгоритмах, в которых преобладают строчные операции.

После рассмотрения строчного формата хранения очевидным является и разреженный столбцовый формат. В этом случае ненулевые элементы матрицы A перечисляются в порядке их появления в столбцах матрицы, а не в строках. Все ненулевые элементы хранятся по столбцам в массиве values; индексы строк ненулевых элементов – в массиве rows; элементы массива pointer указывают на позиции, с которых начинается описание очередного столбца.

Столбцовые представления могут рассматриваться как строчные представления транспонированных матриц. Разреженный столбцовый формат обеспечивает эффективный доступ к столбцам матрицы; доступ к строкам затруднен. Поэтому предпочтительно использовать этот способ хранения в тех алгоритмах, в которых преобладают столбцовые операции.

Рассмотрим столбцовый формат матрицы A:

A=\begin{bmatrix}
1 & -1 & 0 & -3 & 0\\
-1 & 5 & 0 & 0 & 0\\
0 & 0 & 4 & 6 & 4\\
-3 & 0 & 6 & 7 & 0\\
0 & 0 & 4 & 0 & -5
\end{bmatrix}

может быть представлена в разреженном строчном формате как

values=(1, -1, -3, -1, 5, 4, 6, 4, -3, 6, 7, 4, -5);

rows=(1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 3, 5);

pointer=(1, 4, 6, 9, 12, 13).

### Транспонирование матрицы

Рассмотрим задачу транспонирования разреженной матрицы A. Формально матрица A^T может быть определена как

A^T[i,j]=A^T[j,i]

Будем формировать результирующую матрицу построчно. Для этого можно брать столбцы исходной матрицы и создавать из них строки результирующей матрицы. Но операция выделения из CRS-матрицы столбца №i является трудоемкой, т.к. данные в векторе values хранятся по строкам и для выборки данных по столбцу нужно просмотреть всю матрицу, что приводит к квадратичной (от числа ненулевых элементов) трудоемкости алгоритма. Необходимо другое решение. Подробно проблема транспонирования разреженной матрицы обсуждается в книге [10], здесь же рассмотрим основные идеи описанного в [10] алгоритма.

1. Сформируем N одномерных векторов для хранения целых чисел (IntVectors), а также N векторов для хранения вещественных чисел (RealVectors). 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 вещественных 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 pointer.

Алгоритм транспонирует матрицу за линейное время.

## Умножение матрицы в строчном формате на матрицу в столбцовом формате

Для умножения матриц A и B в столбцовом формате преобразуем матрицу A в строчный формат, используя транспонирование. Заведем указатели p1 и p2 указывающие на values матрицы A и values матрицы B. Пусть указатель p1 соответствует элементу (i1, j1), а указатель p2 соответствует элементу (i2, j2).

1. Если j1 = i2 умножаем элементы и накапливаем их результат в соответствующем скалярном произведении.
2. иначе если j1 > i2 сдвигаем p2 на следующий элемент, переходим к п.1
3. иначе если j1 < i2 сдвигаем на следующий элемент, переходим к п1
4. если скалярное произведение не равно 0, то записываем его в результирующую матрицу.
5. переходим на следующий столбец/строку.

# Постановка задачи

Написать программу из трех частей:

1. test-version – тестовая версия с простыми хранением и умножением матриц.
2. openmp-version – версия со столбцовом хранением матрицы и параллельным умножением в openmp.
3. tbb-version – версия со столбцовом хранением матрицы и параллельным умножением в tbb версии.

Каждая часть программы должна состоять из следующих модулей:

* matrix –классы матриц, простейшие алгоритмы работы с ними;
* matrix\_multiplication – консольное приложение позволяющее умножать матрицы и замерять время работы алгоритмов;
* generator – генератор тестов для умножения разряженных матриц, генерирует исходные матрицы заданного размера, по ним простым алгоритмом вычисляет результат умножения;
* checker – сравнивает ответ полученный в модуле matrix\_multiplication с ответом полученным в generator;
* binaryToText – преобразует исходные бинарные матрицы в текстовый вид;

# Руководство пользователя

## Руководство по сборке

Сборка программы осуществляется с помощью Visual Studio. Основной проект matrix\_multiplication.sln каждой из версий программы находится в папке sln\matrix\_multiplication.

## Руководство по запуску

После того как проект собран в папке build должны появиться исполняемые файлы:

* checker.exe
* generator.exe
* binaryToTxt.exe
* matrix\_multiplication.exe

Каждая программа имеет следующие аргументы командной строки:

1. checker.exe [имя\_проверяемого\_файла] [имя\_ответа] [имя\_файла\_с\_результатом\_проверки]
2. generator.exe [имя\_файла\_для\_записи\_исходных\_данных] [имя\_файла\_для\_правильного\_ответа] [размер\_матриц]
3. binaryToTxt.exe [имя\_файла\_исходных\_матриц] [имя\_для\_записи\_текстового\_файла\_исходных\_матриц]

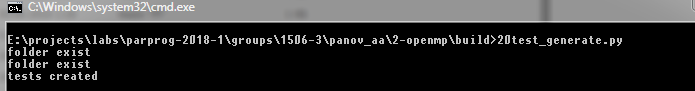
[имя\_файла\_правильного\_ответа][ [имя\_ для\_записи\_файла\_ правильного\_ответа]

Для удобной работы с программами предусмотрены следующие скрипты, написанные на питоне:

* 20test\_generate.py – скрипт генерирует 20 случайных тестов с размерами матриц от 1 до 500 (25 % ненулевых чисел от -10.0 до 10.10) и записывает их в папку tests/20tests.
* 20test\_solve\_and\_check.py – скрипт решает 20 тестов из папки tests/20tests и сравнивает полученные ответы с правильными. Результаты сравнения и полученные ответы записывает в папку tests/20tests\_result.
* 20test\_toTxt.py – скрипт записывает в текстовом виде бинарные данные 20 случайных тестов и ответов к ним.
* 1generate\_big\_test.py – скрипт генерирует тест заданного размера.
* 1solve\_and\_check\_big\_test.py – скрипт решает и проверяет тест заданного размера с нужным количеством потоком.
* 1big\_test\_to\_txt.py – скрипт записывает бинарные данные теста заданного размера в текстовом виде.

Пример работы:

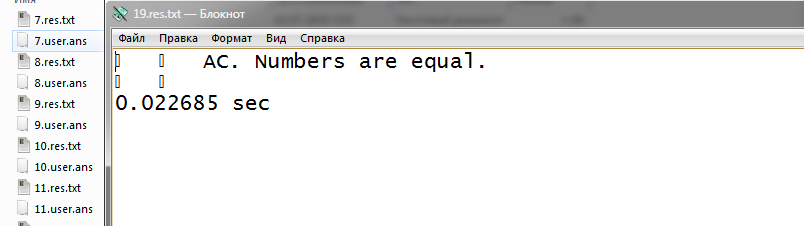
1. Создаем 20 тестов с помощью скрипта 20test\_generate.py.



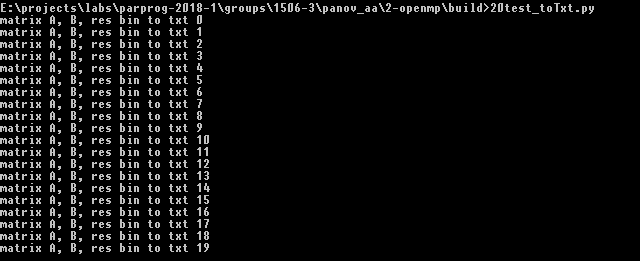
1. Решаем эти тесты с помощью скрипта 20test\_solve\_and\_check.py.

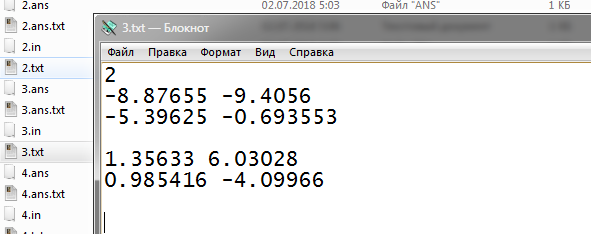


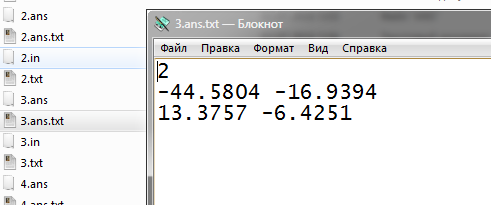
1. Смотрим результаты.

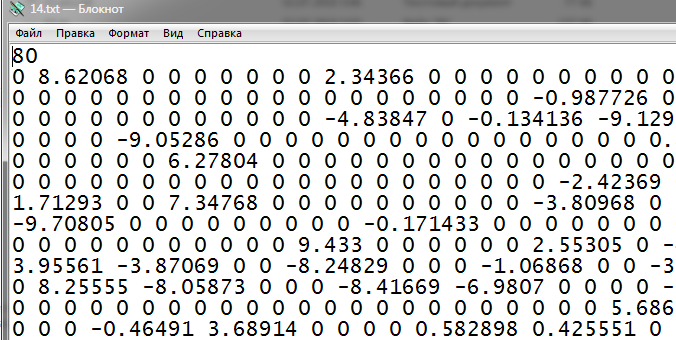


1. Преобразуем исходные данные в текстовый формат









# Руководство программиста

## Описание структуры программы

Программа состоит из трех частей:

1. test-version – тестовая версия с простыми хранением и умножением матриц.
2. openmp – столбцовое хранение матрицы и параллельное умножение в openmp версии.
3. tbb – столбцовое хранение матрицы и параллельное умножение в tbb версии.

Каждая часть программы состоит из следующих модулей:

* matrix – содержит классы матриц, алгоритмы работы с ними;
* matrix\_multiplication – консольное приложение позволяющее умножать матрицы и замерять время работы алгоритмов;
* generator – генератор тестов для умножения разряженных матриц, генерирует исходные матрицы, по ним простым алгоритмом вычисляет правильный ответ;
* checker – сравнивает ответ полученный в модуле matrix\_multiplication с ответом полученным в generator;
* binaryToText – преобразует исходные бинарные матрицы в текстовый формат;
* tests – набор юнит тестов для проверки корректности алгоритмов и структур данных.

Каждая из трех частей программы разбита на следующие папки:

* sln – visual studio проекты
* src – файлы исходного кода
* include – заголовочные файлы
* build – скрипты и исполняемые файлы

## Описание структур данных

### class Matrix

поля:

vector<Element> values;

vector<int> rows;

vector<int> pointer;

int N;

методы:

Element\* operator [] (int j)

Matrix operator \* (Matrix& m)

void transpositionMatrix()

…

Класс описывает двумерную матрицу, которая хранится «одномерно» в vector<Element> vv. Элементы в матрицу добавляются «по столбцам», например матрица A = будет храниться так: . Изначально такой формат хранения казался более удобным для сравнения умножения разряженных матриц в столбцовом формате.

Доступ к ячейкам осуществляется следующим образом:

A[номер столбца][номер строки] = (&vv[номер столбца \*кол-во столбцов])[номер строки])

Умножение матриц выполняется тройным обычным тройным циклом по j, i, z и выполняется распараллеливание по первому циклу, для более быстрой работы.

### class MatrixCCS

поля:

vector<Element> values;

vector<int> rows;

vector<int> pointer;

int N;

методы:

void convertToMatrix(Matrix &A)

void transpositionMatrix()

void unite(const MatrixCCS &m)

MatrixCCS operator \* (const MatrixCCS &m)

…

Класс описывает разряженную матрицу, хранящуюся в столбцовом формате CCS. Структура хранения и алгоритм умножения подробно описаны во введении. В алгоритме умножения есть особенность в том, что в процессе подсчитывается оставшееся количество ненулевых элементов в текущем столбце и строке. Если в столбце или строке все оставшиеся элементы нулевые, выполняется continue. Данный вариант хорошо себя показывает на сильно разряженных матрицах, но значительно усложняет код (умножение занимает 70 строк).

## Параллельное умножение разряженных матриц

Последовательный алгоритм умножения двух матриц A, B в столбцовом формате подробно описан во введении. Для умножения матрица A транспонируется и представляется как матрица в строчном виде. Затем производится умножение строк на столбцы, формирование результата.

В параллельном алгоритме матрица B делится на столбцы. Каждый поток умножает матрицу A на несколько своих столбцов матрицы B и получает часть результата в свою собственную матрицу. В конце все полученные матрицы последовательно объединятся в одну итоговую матрицу.

Каждый поток определяет, необходимые столбцы матрицы B с помощью задачи (структура Task), в ней указано с какого столбца i по какой столбец j нужно использовать из матрицы B и в какую матрицу нужно записать результат.

# Заключение

За счет того, что алгоритм подразумевает наличие N независимых задач, реализации с использованием OpenMP и TBB просты и похожи друг на друга.  
В OpenMP идет parallel\_for по независимым задачам

В TBB реализовано итерационное пространство на N независимых задачах, их выполняет функтор.

### Время работы алгоримов

Замерялась лишь параллельная часть алгоритма умножения матриц, выделение памяти, транспонирование первой матрицы и объединение результатов не учитывались, так как у них линейная сложность и при достаточно больших N их вклад будет относительно небольшим.

N = 2000, время в секундах, 25% ненулевых элементов, 5 повторений.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 поток | 2 потока | 4 потока | 8 потоков Hyper-Threading |
| OpenMP | 19 | 10,2 | 5,8 | 3,9 |
| TBB | 19,1 | 10,5 | 6,1 | 4,2 |

### Ускорение

N = 2000, 25% ненулевых элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 поток | 2 потока | 4 потока | 8 потоков hyper-threading |
| OpenMP | 1 | 1,86 | 3,27 | 4,9 |
| TBB | 1 | 1,81 | 3,13 | 4,56 |

### Масштабируемость

N = 2000, 25% ненулевых элементов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 поток | 2 потока | 4 потока | 8 потоков Hyper-Threading |
| OpenMP | 100% | 93% | 82% | 61% |
| TBB | 100% | 91% | 78% | 57% |

Тесты проводились на 4-ех ядерном процессоре с тактовой частотой 3,3 ГГЦ

Задача по умножению разряженных матриц в столбцовом формате оказалась неплохо масштабируемой, при этом даже на одном потоке и при 25% ненулевых элементах умножение разряженных матриц в 3-4 раза быстрее обычного алгоритма умножения. Eсли же в матрице N ненулевых элементов, (0,1% при N = 1000), то алгоритм умножения разряженных матриц работает быстрее обычного в 5-10 раз.

# Литература

1. Курс Академия Intel: Intel Parallel Programming Professional (Introduction). Константин Баркалов, Владимир Воеводин, Виктор Гергель, Евгений Козинов, Александр Кудин, Валентина Кустикова, Алексей Линев, Иосиф Мееров, Алексей Сиднев, Александр Сысоев | Нижегородский государственный университет им. Н.И.Лобачевского
2. Учебный курс «Технологии разработки параллельных программ». Раздел «Создание параллельной программы». Библиотека Intel Threading Building Blocks – краткое описание. Разработчики: А.А. Сиднев, А.В. Сысоев, И.Б. Мееров.

# Приложение. Фрагменты исходного кода программы

MatrixCCS parallelMult(const MatrixCCS &m, const int numThreads)

{

int numTask = numThreads;

if (numTask > N)

numTask = N;

vector<MatrixCCS> tmp(numTask, MatrixCCS(N));

vector<int> elCountM(numTask);

vector<int> \*cols = &rows;

vector<Task> task(numTask);

{

//prepare task

int sizeTask = m.N / numTask + (bool)(m.N % numTask);

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

task[i] = Task(i\*sizeTask, std::min((i + 1)\*sizeTask, m.N), i % numTask);

int lastPointerM = 0;

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

{

elCountM[i] = lastPointerM;

int jstart = task[i].pointerStart;

const int jend = task[i].pointerEnd;

for (jstart; jstart < jend; jstart++)

{

lastPointerM += m.pointer[jstart + 1] - m.pointer[jstart];

}

}

}

#pragma omp parallel for

for (int itask = 0; itask < task.size(); itask++)

{

for (int j = task[itask].pointerStart; j < task[itask].pointerEnd; j++)

{

int indexTask = task[itask].taskIndex;

int numElInResCol = 0;

const int numElementInCol = m.pointer[j + 1] - m.pointer[j];

if (numElementInCol == 0)

{

int size = tmp[indexTask].pointer.size();

tmp[indexTask].pointer.push\_back(tmp[indexTask].pointer[size - 1]);

continue;

}

int elCountThis = 0;

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

{

const int numElementInRow = pointer[i + 1] - pointer[i];

if (numElementInRow == 0)

{

continue;

}

int tmpNumElCol = numElementInCol;

int tmpNumElRow = numElementInRow;

Element sum = 0;

int tmpElCountM = elCountM[indexTask];

for (int z = 0; z < std::min(tmpNumElCol, tmpNumElRow);)

{

int colThis = (\*cols)[elCountThis];

int rowM = m.rows[tmpElCountM];

if (colThis == rowM)

{

sum += values[elCountThis] \* m.values[tmpElCountM];

tmpNumElCol--;

tmpNumElRow--;

tmpElCountM++;

elCountThis++;

}

else if (colThis < rowM)

{

tmpNumElRow--;

elCountThis++;

}

else

{

tmpNumElCol--;

tmpElCountM++;

}

}

for (int z = 0; z < tmpNumElRow; z++)

elCountThis++;

if (sum != 0)

{

tmp[indexTask].values.push\_back(sum);

tmp[indexTask].rows.push\_back(i);

numElInResCol++;

}

}

const int size = tmp[indexTask].pointer.size();

tmp[indexTask].pointer.push\_back(tmp[indexTask].pointer[size - 1] + numElInResCol);

elCountM[indexTask] += numElementInCol;

}

}

for (int i = 1; i < tmp.size(); i++)

{

tmp[0].unite(tmp[i]);

}

if (tmp[0].pointer.size() < N + 1)

tmp[0].pointer.push\_back(tmp[0].values.size());

return tmp[0];

}

void unite(const MatrixCCS &m)

{

int numCol = pointer.size();

for (int i = 0; i < m.values.size();i++)

{

values.push\_back(m.values[i]);

rows.push\_back(m.rows[i]);

}

for (int i = 1; i < m.pointer.size(); i++)

{

int start = pointer[pointer.size()-1];

pointer.push\_back(m.pointer[i] - m.pointer[i-1] + start);

}

}