Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н. И. Лобачевского»

Институт информационных технологий, математики и механики

Направление: «Программная инженерия»

Отчет

по лабораторным работам курса «Параллельное программирование»

«Умножение разреженных матриц. Элементы типа double. Формат хранения матрицы – столбцовый (CCS).»

Выполнила:

студентка группы 381608

Горелова К.А.

Проверил:

Доцент кафедры МОиСТ, к.т.н.

А.В. Сысоев

Нижний Новгород

2019

Оглавление

1. Введение 3

2. Постановка задачи 4

3. Описание алгоритма 5

3.1. Описание формата хранения 5

3.2. Алгоритм 5

3.3. Последовательная версия алгоритма умножения 6

3.4. Алгоритм транспонирования 9

4. Параллельная реализация 10

4.1. Параллельная реализация с использованием OpenMP 10

4.2. Параллельная реализация с использованием TBB. 12

5. Заключение 15

Приложение 1 16

Приложение 2 19

Приложение 3 24

1. Введение

Алгебра разреженных матриц является важным разделом математики, имеющая практическое применение. Разреженные матрицы встречаются при постановке и решении задач в различных научных и технических областях. Например, при возникновении оптимизационных задач большой размерности с линейными ограничениями. Такие матрицы формируются при численном решении дифференциальных уравнений в частных производных, а также в теории графов. В первую очередь разберемся с понятием разреженная матрица.

В одном из источников дается такое понятие разреженной матрицы: разреженной называют матрицу, имеющую малый процент ненулевых элементов. В ряде других источников встречается такая формулировка: матрица N*N называется разреженной, если количество ее ненулевых элементов есть O(N). Нельзя сказать, что данные определения являются точными в математическом смысле. Причиной этому можно считать то, что как показывает практика, классификация матрицы зависит не только от количества ненулевых элементов, но и от размера матрицы, распределения ненулевых элементов, архитектуры вычислительной системы и используемых алгоритмов.

2. Постановка задачи

Целью данной работы является изучение столбцового способа хранения и алгоритмов обработки разреженных матриц.

Данная цель предполагает решение следующих основных задач:

1) Изучение столбцового способа хранения разреженных матриц.
2) Разработка последовательной версии разреженного матричного умножения.
3) Распараллеливание разреженного матричного умножения в системах с общей памятью с использованием OpenMP и Intel Threading Building Blocks (TBB).

Будем для упрощения рассматривать задачу для случая квадратных матриц. Пусть A и B – квадратные матрицы размера N x N, в которых процент ненулевых элементов мал. Будем считать, что элементы матриц A и B – числа с плавающей точкой двойной точности.

Требуется найти матрицу C = A * B, где символ * соответствует матричному умножению.

Ненулевые элементы расположены таким образом, что в результирующей матрице C большинство элементов равны нулю. Заметим, что так будет не всегда. Нередко в процессе умножения двух разреженных матриц получится плотная матрица. В рамках данной работы предполагается, что матрица C также является разреженной.

3. Описание алгоритма

3.1. Описание формата хранения

CCS – столбцовый формат хранения матриц. Элементы матрицы и ее структура хранятся в трех массивах. Первый массив хранит значения элементов по столбцам (столбцы рассматриваются по порядку слева направо), второй – номера строк для каждого элемента, а третий - индексы начала каждого столбца.

Количество элементов массива ColIndex равно N + 1. При этом элементы столбца i в массиве val находятся по индексам от ColIndex[i] до ColIndex [i + 1] – 1 включительно (i-ый элемент массива ColIndex указывает на начало i-ого столбца). Исходя из этого обрабатывается случай пустых столбцов, а также добавляется «лишний» элемент в массив ColIndex – устраняется особенность при доступе к элементам последнего столбца. Данный элемент хранит номер последнего ненулевого элемента матрицы A + 1, что соответствует количеству ненулевых элементов NZ.

Оценим объем необходимой памяти.
Плотное представление: $M = 8 N^2$ байт.
В координатном формате: M = 16 NZ байт.
В формате CCS: M = 8 NZ + 4 NZ + 4 (N + 1) = 12 NZ + 4 N + 4.

Далее в работе мы будем использовать формат CCS в виде трех массивов, расположенных в памяти в порядке val, row, ColIndex. Индексация массивов в стиле языка C – с нуля. Элементы в каждом столбце упорядочиваются по номеру строки. матрице хранятся числа с плавающей точкой двойной точности (тип данных double)

Будем использовать следующую структуру данных:

```
struct ccsMatrix {
    int N;
    int NZ;
    double* val;
    int* row;
    int* ColIndex;
};
```

3.2. Алгоритм

Будем формировать матрицы A и B при помощи датчика случайных чисел. Данная задача включает два этапа – построение портрета (шаблона) матрицы и наполнение этого

портрета конкретными значениями. Рассмотрим пример портрета матрицы, а также его представление в формате CCS. CCS-представление портрета определяется парой массивов row и ColIndex, массив val заполняется по мере наполнения матрицы конкретными значениями. Для формирования портрета матрицы A применим следующую схему: будем генерировать случайным образом позиции cntInCol ненулевых элементов в каждой строке матрицы.

Для комфортной работы нам необходимо добавить вспомогательные функции.

1) Инициализация матрицы - выделение памяти под структуру данных для хранения матрицы в формате CCS.

```
void initMatrix(ccsMatrix *mtx, int nz, int n) {
        mtx->NZ = nz;
        mtx->N = n;
        mtx->val = new double[nz];
        mtx->row = new int[nz];
        mtx->ColIndex = new int[n + 1];
 }
```

2) Удаление матрицы – освобождение выделенной ранее памяти.

```
void freeMatrix(ccsMatrix *mtx) {
        delete[] mtx->val;
        delete[] mtx->row;
        delete[] mtx->ColIndex;
}
```

3.3. Последовательная версия алгоритма умножения

Используемая структура данных, построенная на основе формата CCS, предполагает хранение только ненулевых элементов, что несколько усложняет программирование вычисления скалярного произведения, но одновременно уменьшает количество арифметических операций. При вычислении скалярных произведений нет необходимости умножать нули и накапливать полученный нуль в частичную сумму, что положительно влияет на сокращение времени счета.

Необходимо научиться выделять вектора в матрицах A и B. В соответствии с определением, речь идет о столбцах матрицы A и строках матрицы B. Выделить столбец матрицы в формате CCS не представляет труда: i-ый столбец может быть легко найден, так как ссылки на первый элемент (ColIndex[i]) и последний элемент (ColIndex[i+1]-1) известны, что позволяет получить доступ к значениям элементов и номерам строк, хранящихся в массивах val и row соответственно. Таким образом, проход по столбцу

выполняется за время, пропорциональное числу ненулевых элементов в указанном столбце, а проход по всем столбцам – за время, пропорциональное NZ.

Проблема возникает с выделением строки. Чтобы найти элементы строки j необходимо просмотреть массив row (~NZ операций) и выделить все элементы, у которых в соответствующей ячейке массива row записано число j. Если это нужно проделать для каждой строки, необходимо ~NZ*N операций, что выглядит неэффективным.

Возможное, но не единственное решение проблемы состоит в транспонировании матрицы B. Если удастся сделать это достаточно быстро, появится возможность эффективно работать со строками исходной матрицы B. Другой вариант – для каждой CCS-матрицы, которая может понадобиться в строковом представлении, дополнительно хранить транспонированный портрет. Сэкономив время на транспонировании, придется смириться с расходами времени на поддержание дополнительного портрета в актуальном состоянии.

Также, необходимо научиться записывать посчитанные элементы в матрицу C. Учитывая, что C хранится в формате CCS, важно избежать перепаковок. Для этого нужно обеспечить пополнение матрицы C ненулевыми элементами последовательно, по столбцам – сверху вниз, слева направо.

Реализуем алгоритм умножения разреженных матриц с помощью двухфазного алгоритма. Он заключается в том, что мы разделяем вычисления на 2 фазы: символическую и символьную. В рамках символической части, которая выполняется однократно, производится построение портрета результата, а численная часть состоит в заполнении портрета значениями. При многократных однотипных операциях с одинаковыми портретами достигается огромный выигрыш производительности по сравнению с обычным подходом. Символическая часть вычислений решает все вопросы, связанные с выделением памяти, что часто делает программу понятнее, а иногда и быстрее.

Описание алгоритма

Задача символической фазы – построить портрет матрицы C. Это можно сделать, исключив вычисление скалярного произведения. Так, после транспонирования матрицы B необходимо при каждом «умножении» векторов (строк матриц A и BT) определить, есть ли хотя бы одна пара элементов, находящихся в одном и том же столбце. Если такая пара найдется, нужно учесть это при формировании портрета матрицы C.

```
int multMatrix(ccsMatrix *A, ccsMatrix *B, ccsMatrix *C, int size) {
```

```
    int n = A->N;
    int NZ = 0, indexA, k, end;
    vector<int> colIndex, row;
    colIndex.push_back(0);
    int *tmp = new int[size];
    for (int i = 0; i < n; i++) {
        memset(tmp, -1, size * sizeof(int));
        for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; j++)
            tmp[A->row[j]] = j;
        for (int j = 0; j < size; j++) {
            indexA = -1;
            k = B->ColIndex[j];
            end = B->ColIndex[j + 1];
            while (indexA == -1 && k < end) {
                indexA = tmp[B->row[k]];
                k++;
            }
            if (indexA != -1) {
                row.push_back(j);
                NZ++;
            }
        }
        colIndex.push_back(NZ);
    }
    initMatrix(C, NZ, n);
    for (int j = 0; j < NZ; ++j)
        C->row[j] = row[j];
    for (int i = 0; i <= n; ++i)
        C->ColIndex[i] = colIndex[i];
    delete[] tmp;
    return 0;
}
```

Численная фаза.

1) Создадим дополнительный целочисленный массив длины N. Инициализируем его числом -1. Обнулим вещественную переменную sum и массив указателей на элементы.
2) Просмотрим в цикле все ненулевые элементы столбца и заполним массив указателей. tmp[A->row[j]] = j; значит, что a[i, НОМЕР] лежит в ячейке массива val с номером tmp[НОМЕР]
3) Построен индекс строки i матрицы A. Теперь необходимо умножить его на каждую из строк матрицы BT.
4) Если значение равно -1, то умножение выполнять не нужно, иначе умножаем и накапливаем в sum.

```
int numericMult(int size, const ccsMatrix *A, const ccsMatrix *B, ccsMatrix *C) {
int n = A->N;
int index = 0, indexA, startC, finishC, colC;
int *tmp = new int[size];
double sum = 0.0;
for (int i = 0; i < n; ++i) {
    startC = C->ColIndex[i];
    finishC = C->ColIndex[i + 1];
    if (finishC > startC) {
        memset(tmp, -1, sizeof(int)* size);
        for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; ++j)
```

```
            tmp[A->row[j]] = j;
        for (int j = startC; j < finishC; ++j, sum = 0) {
            colC = C->row[j];
            for (int k = B->ColIndex[colC]; k < B->ColIndex[colC + 1]; ++k) {
                indexA = tmp[B->row[k]];
                if (indexA != -1)
                    sum += A->val[indexA] * B->val[k];
            }
            C->val[index] = sum;
            index++;
        }
    }
}
delete[] tmp;
return 0;
}
```

Полная версия последовательного алгоритма в Приложении 1.

3.4. Алгоритм транспонирования

Для плотных матриц алгоритм достаточно прост. У тех, кто начинает работать с разреженными матрицами, иногда возникает иллюзия, что и там все очень просто.

Суть операции транспонирования состоит в том, что если создать нулевую матрицу, а далее добавлять туда по одному элементу, выбирая их из CCS-структуры исходной матрицы, придется столкнуться с необходимостью большого количества перепаковок. Чтобы этого избежать, нужно формировать транспонированную матрицу построчно. Для этого можно брать столбцы исходной матрицы и создавать из них строки результирующей матрицы. Однако сделать это не так просто. Необходимо другое решение.

Основная идея состоит в использовании структур данных матрицы AT для промежуточных результатов вычислений.

```
void transMatrix(ccsMatrix *A, ccsMatrix *AT) {
    int sum = 0;
    int index, colIn;
    double v;
    memset(AT->ColIndex, 0, (AT->N + 1) * sizeof(int));
    for (int i = 0; i < A->NZ; i++)
        AT->ColIndex[A->row[i] + 1]++;
    for (int i = 1; i <= A->N; i++) {
        int tmp = AT->ColIndex[i];
        AT->ColIndex[i] = sum;
        sum += tmp;
    }
    for (int i = 0; i < A->N; i++) {
        int row = i;
        for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; j++) {
            v = A->val[j];
            colIn = A->row[j];
            index = AT->ColIndex[colIn + 1];
            AT->val[index] = v;
            AT->row[index] = row;
            AT->ColIndex[colIn + 1]++;
        }
```

```
    }
  }
```

4. Параллельная реализация

4.1. Параллельная реализация с использованием OpenMP.

Один из вариантов повышения производительности умножения разреженных матриц – распараллеливание.

Идея распараллеливания алгоритма выглядит достаточно очевидной. Во внешнем цикле мы перебираем столбцы матрицы A и далее работаем с ними независимо. Таким образом, естественный вариант параллелизма – распределение столбцов между потоками. Однако проблемы начинаются, как только мы добираемся до момента, когда необходимо записать новые значения в вектора. Продублируем все рабочие структуры по числу столбцов, соберем в них данные по каждому столбцу отдельно, а в конце одним потоком все «просуммируем».

Следующее замечание касается массива temp, который мы используем для запоминания расположения ненулевых элементов матрицы A. С ним вариант с дублированием не годится, поскольку его размер равен длине столбца. Таким образом, при попытке продублировать его по числу столбцов мы, фактически, потратим объем памяти лишь вполовину меньший, чем нужен для размещения матрицы A в плотном виде. Естественно, это неприемлемо. К счастью, для этого массива возможен простой выход, его можно продублировать по числу потоков. На самом деле мы поступим еще проще и поместим его объявление непосредственно в параллельную секцию.

Наконец, последний момент касается вектора columnIndex. С ним ситуация существенно проще, чем с остальными элементами разреженной матрицы, поскольку его размер равен «число столбцов + 1». Соответственно его можно объявить обычным массивом, работать с ним в потоках с элементами, соответствующими обрабатываемым столбцам (а, значит, бесконфликтно), накапливая число элементов в каждом столбце отдельно. И, наконец, по завершении параллельной секции, последовательно просуммировать число элементов по столбцам, получив итоговый массив.

```
int multiplyOmp(ccsMatrix A, ccsMatrix B, ccsMatrix *C) {
    if (A.N != B.N)
        return 1;
    int N = A.N;
    vector<int> rows;
    vector<double> value;
    vector<int> columnIndex;
       columnIndex.push_back(0);
    int nz = 0;
```

```
#pragma omp parallel
	{
		int *temp = new int[N];

#pragma omp for
		for (int i = 0; i < N; i++) {
			memset(temp, -1, N * sizeof(int));
			int ind1 = A.ColIndex[i], ind2 = A.ColIndex[i + 1];
			for (int j = ind1; j < ind2; j++) {
				int row = A.row[j];
				temp[row] = j;
			}
		// суммируем построчно
			for (int j = 0; j < N; j++) {
				double sum = 0;
				int ind3 = B.ColIndex[i], ind4 = B.ColIndex[i + 1];
				for (int k = ind3; k < ind4; k++) {
					int brow = B.row[k];
					int aind = temp[brow];
					if (aind != -1)
						sum += A.val[aind] * B.val[k];
				}
				if (fabs(sum) > 0) {
					rows.push_back(j);
					value.push_back(sum);
					nz++;
				}
			}
			columnIndex.push_back(nz);
		}
	}
    // собираем на одном потоке
    initMatrix(C, nz, N);
    for (int j = 0; j < nz; j++) {
        C->row[j] = rows[j];
        C->val[j] = value[j];
    }
    for (int i = 0; i <= N; i++)
        C->ColIndex[i] = columnIndex[i];
    return 0;
   }
```

Проверим корректность реализованного алгоритма сравнением полученных матриц.

```
bool compareMatrix(ccsMatrix A, ccsMatrix B) {
    if (A.NZ != B.NZ) {
        return false;
    }
    for (int i = 0; i < A.NZ; i++) {
        if (A.val[i] != B.val[i] ||A.row[i] != B.row[i]) {
            return false;
        }
    }
    return true;
}
```

```
Time usual is 4.84465
Time omp is 1.23441
Compare parallel and linear version 1
Acceleration is 3.92466
Efficienty is 0.981166
```

Полная версия параллельной реализации с использованием OpenMP в Приложении 2.

4.2. Параллельная реализация с использованием TBB.

В процессе разработки будут использованы следующие элементы библиотеки:

1) Инициализация библиотеки с использованием объекта класса task_scheduler_init
2) Распараллеливание цикла с помощью шаблонной функции parallel_for()
3) Одномерное итерационное пространство blocked_range
4) Класс-функтор, который нам предстоит разработать и который, собственно, и будет реализовывать основную часть умножения в методе operator()

Идейно разработка параллельной версии на основе TBB очень похожа на OpenMP. Продублируем по числу столбцов служебные вектора rows и value. Также создадим массив columnIndex длины «число столбцов + 1» и точно так же будем запоминать в каждом его элементе, сколько не нулей будет содержать соответствующий столбец матрицы C. Фактически отличия в реализации функции будут состоять в использовании шаблонной функции parallel_for(), которая «скроет» в себе весь содержательный код – то, что в OpenMP-версии составляло содержимое параллельной секции.

Сама функция при этом будет выглядеть следующим образом:

```
int multiplyTbb(ccsMatrix A, ccsMatrix B, ccsMatrix *C) {
    if (A.N != B.N)
        return 1;
    int N = A.N;

    tbb::task_scheduler_init init(4);

    vector<int>* rows = new vector<int>[N];
    vector<double>* value = new vector<double>[N];
    int* columnIndex = new int[N + 1];

    memset(columnIndex, 0, sizeof(int) * N);
    int grainsize = 10;
    tbb::parallel_for(tbb::blocked_range<int>(0, A.N, grainsize),
        multiplicate(A, B, rows, value, columnIndex));
    int NZ = 0;
    for (int i = 0; i < N; i++) {
        int tmp = columnIndex[i];
        columnIndex[i] = NZ;
        NZ += tmp;
    }
    columnIndex[N] = NZ;
    initMatrix(C, NZ, N);
    int count = 0;
    for (int i = 0; i < N; i++) {
        int size = rows[i].size();
        memcpy(&C->row[count], &rows[i][0],
            size * sizeof(int));
        memcpy(&C->val[count], &value[i][0],
```

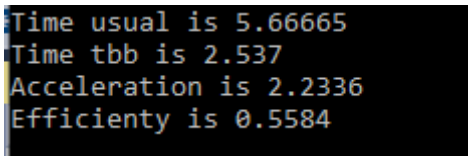
```
            size * sizeof(double));
        count += size;
    }
    memcpy(C->ColIndex, &columnIndex[0], (N + 1) * sizeof(int));

    return 0;
}
```

Использованное итерационное пространство blocked_range является встроенным. Таким образом, для завершения реализации нам необходимо создать класс-функтор multiplicate, принимающий на вход две матрицы и оперирующий рабочими векторами rows, value и columnIndex.

```
class multiplicate {
    ccsMatrix A, B;
    vector<int>* rows;
    vector<double>* value;
    int * columnIndex;
 public :
    multiplicate(ccsMatrix a, ccsMatrix b,
        vector<int>* row, vector<double>* val,
        int *colIndex) : A(a), B(b), rows(row),
        value(val), columnIndex(colIndex) {}
    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.ColIndex[i], ind2 = A.ColIndex[i + 1];
            for (j = ind1; j < ind2; j++) {
                int row = A.row[j];
                temp[row] = j;
            }
            for (j = 0; j < N; j++) {
                double sum = 0;
                int ind3 = B.ColIndex[j], ind4 = B.ColIndex[j + 1];
                for (k = ind3; k < ind4; k++) {
                    int brow = B.row[k];
                    int aind = temp[brow];
                    if (aind != -1)
                        sum += A.val[aind] * B.val[k];
                }
                if (fabs(sum) > 0) {
                    rows[i].push_back(j);
                    value[i].push_back(sum);
                    columnIndex[i]++;
                }
            }
        }
        delete[] temp;
    }
  };
```

Заметим, что согласно идеологии TBB класс-функтор должен содержать только ссылки на обрабатываемые потоками данные. Исключения могут быть сделаны для небольших по размеру переменных, копирование которых при расщеплении итерационного

пространства будет требовать мало ресурсов. В данном случае, такое исключение сделано для матриц A и B.



Как видим, OpenMP в данной задаче обогоняет TBB. Это может быть связано с оптимизацией кода в OpenMP версии.

5. Заключение

В лабораторных работах были реализованы последовательная и параллельные версии алгоритма умножения разреженных матриц с элементами типа double, столбцового формата хранения с использованием средств OpenMP и TBB. На основании результатов лабораторной работы сделаем следующий вывод: оптимальнее для распараллеливания использовать OpenMP, так как соответствующая версия алгоритма показала лучшие результаты.

Приложение 1

```
#include <iostream>
#include <vector>
#include <random>
#include <string.h>

using std::vector;
struct ccsMatrix {
    int N;
    int NZ;
    double* val;
    int* row;
    int* ColIndex;
};

void initMatrix(ccsMatrix *mtx, int nz, int n) {
    mtx->NZ = nz;
    mtx->N = n;
    mtx->val = new double[nz];
    mtx->row = new int[nz];
    mtx->ColIndex = new int[n + 1];
}

void freeMatrix(ccsMatrix *mtx) {
    delete[] mtx->val;
    delete[] mtx->row;
    delete[] mtx->ColIndex;
}

void transMatrix(ccsMatrix *A, ccsMatrix *AT) {
    int sum = 0;
    int index, colIn;
    double v;
    memset(AT->ColIndex, 0, (AT->N + 1) * sizeof(int));
    for (int i = 0; i < A->NZ; i++)
        AT->ColIndex[A->row[i] + 1]++;
    for (int i = 1; i <= A->N; i++) {
        int tmp = AT->ColIndex[i];
        AT->ColIndex[i] = sum;
        sum += tmp;
    }
    for (int i = 0; i < A->N; i++) {
        int row = i;
        for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; j++) {
            v = A->val[j];
            colIn = A->row[j];
            index = AT->ColIndex[colIn + 1];
            AT->val[index] = v;
            AT->row[index] = row;
            AT->ColIndex[colIn + 1]++;
        }
    }
}

void generateMatrix(ccsMatrix *mtx, int n, int cntInCol) {
    int rows;
    bool exist;
    int nz = cntInCol * n;
    initMatrix(mtx, nz, n);
    for (int col = 0; col < n; col++) {
        for (int row = 0; row < cntInCol; row++) {
            do {
                rows = std::rand() % n;
```

```
                exist = false;
                for (int i = 0; i < row; i++)
                    if (rows == mtx->row[col*cntInCol + i])
                        exist = true;
                break;
            } while (exist == true);
            mtx->row[col*cntInCol + row] = rows;
        }
        for (int row = 0; row < cntInCol - 1; row++)
            for (int val = 0; val < cntInCol - 1; val++)
                if (mtx->row[col*cntInCol + val] >
                    mtx->row[col*cntInCol + val + 1]) {
                    int tmp = mtx->row[col*cntInCol + val];
                    mtx->row[col*cntInCol + val] =
                        mtx->row[col*cntInCol + val + 1];
                    mtx->row[col*cntInCol + val + 1] = tmp;
                }
    }
    for (int val = 0; val < nz; val++)
        mtx->val[val] = (std::rand() % 10000) / 1000.0f;
    int c = 0;
    for (int col = 0; col <= n; col++) {
        mtx->ColIndex[col] = c;
        c += cntInCol;
    }
}

void printMatrix(int n, ccsMatrix *mtx) {
    int i;
    int k = mtx[0].NZ;
    std::cout << "\n Value: ";
    for (i = 0; i < k; i++)
        std::cout << " "<< mtx[0].val[i];
    std::cout << "\n Row: ";
    for (i = 0; i < k; i++)
        std::cout << " "<< mtx[0].row[i];
    std::cout << "\n ColIndex: ";
    for (i = 0; i < n + 1; i++)
        std::cout << " " << mtx[0].ColIndex[i];
    std::cout << std::endl;
}

int multMatrix(ccsMatrix *A, ccsMatrix *B, ccsMatrix *C, int size) {
    int n = A->N;
    int NZ = 0, indexA, k, end;
    vector<int> colIndex, row;
    colIndex.push_back(0);
    int *tmp = new int[size];
    for (int i = 0; i < n; i++) {
        memset(tmp, -1, size * sizeof(int));
        for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; j++)
            tmp[A->row[j]] = j;
        for (int j = 0; j < size; j++) {
            indexA = -1;
            k = B->ColIndex[j];
            end = B->ColIndex[j + 1];
            while (indexA == -1 && k < end) {
                indexA = tmp[B->row[k]];
                k++;
            }
            if (indexA != -1) {
                row.push_back(j);
                NZ++;
            }
        }
```

```
            colIndex.push_back(NZ);
        }
        initMatrix(C, NZ, n);
        for (int j = 0; j < NZ; ++j)
            C->row[j] = row[j];
        for (int i = 0; i <= n; ++i)
            C->ColIndex[i] = colIndex[i];
        delete[] tmp;
        return 0;
    }

    int numericMult(int size, const ccsMatrix *A, const ccsMatrix *B, ccsMatrix *C) {
        int n = A->N;
        int index = 0, indexA, startC, finishC, colC;
        int *tmp = new int[size];
        double sum = 0.0;
        for (int i = 0; i < n; ++i) {
            startC = C->ColIndex[i];
            finishC = C->ColIndex[i + 1];
            if (finishC > startC) {
                memset(tmp, -1, sizeof(int)* size);
                for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; ++j)
                    tmp[A->row[j]] = j;
                for (int j = startC; j < finishC; ++j, sum = 0) {
                    colC = C->row[j];
                    for (int k = B->ColIndex[colC]; k < B->ColIndex[colC + 1]; ++k) {
                        indexA = tmp[B->row[k]];
                        if (indexA != -1)
                            sum += A->val[indexA] * B->val[k];
                    }
                    C->val[index] = sum;
                    index++;
                }
            }
        }
        delete[] tmp;
        return 0;
    }

    int main(int argc, char** argv) {
        int N = (argc != 1) ? atoi(argv[1]) : 5;
        int cntInCol = (argc != 1) ? atoi(argv[2]) : 2;
        ccsMatrix A, B, AT, C;
        generateMatrix(&A, N, cntInCol);
        std::cout << "Matrix A: ";
        printMatrix(N, &A);
        initMatrix(&AT, A.NZ, A.N);
        transMatrix(&A, &AT);
        freeMatrix(&A);
        generateMatrix(&B, N, cntInCol);
        std::cout << "Matrix B: ";
        printMatrix(N, &B);
        initMatrix(&C, cntInCol, N);
        multMatrix(&AT, &B, &C, N);
        numericMult(N, &AT, &B, &C);
        std::cout << "Matrix C: ";
        printMatrix(N, &C);
        std::cout << A.N;
        freeMatrix(&AT);
        freeMatrix(&B);
        freeMatrix(&C);
        return 0;
    }
```

Приложение 2

```
#include <omp.h>
#include <iostream>
#include <vector>
#include <random>
#include <cstring>
#include <ctime>

using std::vector;
struct ccsMatrix {
    int N;
    int NZ;
    double* val;
    int* row;
    int* ColIndex;
};

void initMatrix(ccsMatrix *mtx, int nz, int n) {
    mtx->NZ = nz;
    mtx->N = n;
    mtx->val = new double[nz];
    mtx->row = new int[nz];
    mtx->ColIndex = new int[n + 1];
}

void freeMatrix(ccsMatrix *mtx) {
    delete[] mtx->val;
    delete[] mtx->row;
    delete[] mtx->ColIndex;
}

void transMatrix(ccsMatrix A, ccsMatrix *AT) {
    int sum = 0;
    int index, colIn;
    double v;
    memset(AT->ColIndex, 0, (AT->N + 1) * sizeof(int));
    for (int i = 0; i < A.NZ; i++)
        AT->ColIndex[A.row[i] + 1]++;
    for (int i = 1; i <= A.N; i++) {
        int tmp = AT->ColIndex[i];
        AT->ColIndex[i] = sum;
        sum += tmp;
    }
    for (int i = 0; i < A.N; i++) {
        int row = i;
        for (int j = A.ColIndex[i]; j < A.ColIndex[i + 1]; j++) {
            v = A.val[j];
            colIn = A.row[j];
            index = AT->ColIndex[colIn + 1];
            AT->val[index] = v;
            AT->row[index] = row;
            AT->ColIndex[colIn + 1]++;
        }
    }
}

void generateMatrix(ccsMatrix *mtx, int n, int cntInCol) {
    int rows;
    bool exist;
    int nz = cntInCol * n;
    initMatrix(mtx, nz, n);
    for (int col = 0; col < n; col++) {
        for (int row = 0; row < cntInCol; row++) {
```

```
            do {
                rows = std::rand() % n;
                exist = false;
                for (int i = 0; i < row; i++)
                    if (rows == mtx->row[col*cntInCol + i])
                        exist = true;
                break;
            } while (exist == true);
            mtx->row[col*cntInCol + row] = rows;
        }
        for (int row = 0; row < cntInCol - 1; row++)
            for (int val = 0; val < cntInCol - 1; val++)
                if (mtx->row[col*cntInCol + val] >
                    mtx->row[col*cntInCol + val + 1]) {
                    int tmp = mtx->row[col*cntInCol + val];
                    mtx->row[col*cntInCol + val] =
                        mtx->row[col*cntInCol + val + 1];
                    mtx->row[col*cntInCol + val + 1] = tmp;
                }
    }
    for (int val = 0; val < nz; val++)
        mtx->val[val] = (std::rand() % 10000) / 1000.0f;
    int c = 0;
    for (int col = 0; col <= n; col++) {
        mtx->ColIndex[col] = c;
        c += cntInCol;
    }
}

void printMatrix(int n, ccsMatrix *mtx) {
    int i;
    int k = mtx[0].NZ;
    std::cout << "\n Value: ";
    for (i = 0; i < k; i++)
        std::cout << " " << mtx[0].val[i];
    std::cout << "\n Row: ";
    for (i = 0; i < k; i++)
        std::cout << " " << mtx[0].row[i];
    std::cout << "\n ColIndex: ";
    for (i = 0; i < n + 1; i++)
        std::cout << " " << mtx[0].ColIndex[i];
    std::cout << std::endl;
}

int multMatrix(ccsMatrix *A, ccsMatrix *B, ccsMatrix *C, int size) {
    int n = A->N;
    int NZ = 0, indexA, k, end;
    vector<int> colIndex, row;
    colIndex.push_back(0);
    int *tmp = new int[size];
    for (int i = 0; i < n; i++) {
        memset(tmp, -1, size * sizeof(int));
        for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; j++)
            tmp[A->row[j]] = j;
        for (int j = 0; j < size; j++) {
            indexA = -1;
            k = B->ColIndex[j];
            end = B->ColIndex[j + 1];
            while (indexA == -1 && k < end) {
                indexA = tmp[B->row[k]];
                k++;
            }
            if (indexA != -1) {
                row.push_back(j);
                NZ++;
```

```
            }
        }
        colIndex.push_back(NZ);
    }
    initMatrix(C, NZ, n);
    for (int j = 0; j < NZ; ++j)
        C->row[j] = row[j];
    for (int i = 0; i <= n; ++i)
        C->ColIndex[i] = colIndex[i];
    delete[] tmp;
    return 0;
}

int numericMult(int size, const ccsMatrix *A, const ccsMatrix *B, ccsMatrix *C) {
    int n = A->N;
    int index = 0, indexA, startC, finishC, colC;
    int *tmp = new int[size];
    double sum = 0.0;
    for (int i = 0; i < n; ++i) {
        startC = C->ColIndex[i];
        finishC = C->ColIndex[i + 1];
        if (finishC > startC) {
            memset(tmp, -1, sizeof(int)* size);
            for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; ++j)
                tmp[A->row[j]] = j;
            for (int j = startC; j < finishC; ++j, sum = 0) {
                colC = C->row[j];
                for (int k = B->ColIndex[colC]; k < B->ColIndex[colC + 1]; ++k) {
                    indexA = tmp[B->row[k]];
                    if (indexA != -1)
                        sum += A->val[indexA] * B->val[k];
                }
                C->val[index] = sum;
                index++;
            }
        }
    }
    delete[] tmp;
    return 0;
}


int multiplyOmp(ccsMatrix A, ccsMatrix B, ccsMatrix *C) {
    if (A.N != B.N)
        return 1;
    int N = A.N;
    vector<int> rows;
    vector<double> value;
    vector<int> columnIndex;
       columnIndex.push_back(0);
    int nz = 0;
#pragma omp parallel
       {
              int *temp = new int[N];

#pragma omp for
              for (int i = 0; i < N; i++) {
                     memset(temp, -1, N * sizeof(int));
                     int ind1 = A.ColIndex[i], ind2 = A.ColIndex[i + 1];
                     for (int j = ind1; j < ind2; j++) {
                            int row = A.row[j];
                            temp[row] = j;
                     }
            // суммируем построчно
                     for (int j = 0; j < N; j++) {
```

```
                                double sum = 0;
                                int ind3 = B.ColIndex[i], ind4 = B.ColIndex[i + 1];
                                for (int k = ind3; k < ind4; k++) {
                                        int brow = B.row[k];
                                        int aind = temp[brow];
                                        if (aind != -1)
                                                sum += A.val[aind] * B.val[k];
                                }
                                if (fabs(sum) > 0) {
                                        rows.push_back(j);
                                        value.push_back(sum);
                                        nz++;
                                }
                        }
                        columnIndex.push_back(nz);
                }
        }
    // собираем на одном потоке
    initMatrix(C, nz, N);
    for (int j = 0; j < nz; j++) {
        C->row[j] = rows[j];
        C->val[j] = value[j];
    }
    for (int i = 0; i <= N; i++)
        C->ColIndex[i] = columnIndex[i];
    return 0;
}

bool compareMatrix(ccsMatrix A, ccsMatrix B) {
    if (A.NZ != B.NZ) {
        return false;
    }
    for (int i = 0; i < A.NZ; i++) {
        if (A.val[i] != B.val[i] ||A.row[i] != B.row[i]) {
            return false;
        }
    }
    return true;
}
int main(int argc, char** argv) {
    omp_set_num_threads(4);
    int N = (argc != 1) ? atoi(argv[1]) : 1000;
    int cntInCol = (argc != 1) ? atoi(argv[2]) : 500;
    ccsMatrix A, B, AT, C1, C2;
    generateMatrix(&A, N, cntInCol);
    initMatrix(&AT, A.NZ, A.N);
    transMatrix(A, &AT);
    freeMatrix(&A);
    generateMatrix(&B, N, cntInCol);
    initMatrix(&C1, cntInCol, N);
    double t1 = omp_get_wtime();
    //multiply(AT, B, &C1);
    multMatrix(&AT, &B, &C1, N);
    numericMult(N, &AT, &B, &C1);
    double t2 = omp_get_wtime();
    std::cout << "Time usual is " << t2 - t1 << std::endl;

    initMatrix(&C2, cntInCol, N);
    double ompT1 = omp_get_wtime();
    multiplyOmp(AT, B, &C2);
    double ompT2 = omp_get_wtime();
    std::cout << "Time omp is " << ompT2 - ompT1 << std::endl;
    std::cout << "Compare parallel and linear version 1" << std::endl;
    double acc = (t2 - t1) / (ompT2 - ompT1);
    std::cout << "Acceleration is " << acc << std::endl;
```

```
    std::cout << "Efficienty is " << acc / 4 << std::endl;
    freeMatrix(&AT);
    freeMatrix(&B);
    freeMatrix(&C1);
    freeMatrix(&C2);
    return 0;
}
```

Приложение 3

```
 #include <tbb/tbb.h>
#include <iostream>
#include <vector>
#include <random>
#include <cstring>
#include <ctime>

#include "tbb/task_scheduler_init.h"
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using std::vector;
struct ccsMatrix {
    int N;
    int NZ;
    double* val;
    int* row;
    int* ColIndex;
};

class multiplicate {
    ccsMatrix A, B;
    vector<int>* rows;
    vector<double>* value;
    int * columnIndex;
 public :
    multiplicate(ccsMatrix a, ccsMatrix b,
        vector<int>* row, vector<double>* val,
        int *colIndex) : A(a), B(b), rows(row),
        value(val), columnIndex(colIndex) {}
    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.ColIndex[i], ind2 = A.ColIndex[i + 1];
            for (j = ind1; j < ind2; j++) {
                int row = A.row[j];
                temp[row] = j;
            }
            for (j = 0; j < N; j++) {
                double sum = 0;
                int ind3 = B.ColIndex[j], ind4 = B.ColIndex[j + 1];
                for (k = ind3; k < ind4; k++) {
                    int brow = B.row[k];
                    int aind = temp[brow];
                    if (aind != -1)
                        sum += A.val[aind] * B.val[k];
                }
                if (fabs(sum) > 0) {
                    rows[i].push_back(j);
                    value[i].push_back(sum);
                    columnIndex[i]++;
                }
            }
        }
        delete[] temp;
    }
};
```

```
void initMatrix(ccsMatrix *mtx, int nz, int n) {
    mtx->NZ = nz;
    mtx->N = n;
    mtx->val = new double[nz];
    mtx->row = new int[nz];
    mtx->ColIndex = new int[n + 1];
}

void freeMatrix(ccsMatrix *mtx) {
    delete[] mtx->val;
    delete[] mtx->row;
    delete[] mtx->ColIndex;
}

void transMatrix(ccsMatrix A, ccsMatrix *AT) {
    int sum = 0;
    int index, colIn;
    double v;
    memset(AT->ColIndex, 0, (AT->N + 1) * sizeof(int));
    for (int i = 0; i < A.NZ; i++)
        AT->ColIndex[A.row[i] + 1]++;
    for (int i = 1; i <= A.N; i++) {
        int tmp = AT->ColIndex[i];
        AT->ColIndex[i] = sum;
        sum += tmp;
    }
    for (int i = 0; i < A.N; i++) {
        int row = i;
        for (int j = A.ColIndex[i]; j < A.ColIndex[i + 1]; j++) {
            v = A.val[j];
            colIn = A.row[j];
            index = AT->ColIndex[colIn + 1];
            AT->val[index] = v;
            AT->row[index] = row;
            AT->ColIndex[colIn + 1]++;
        }
    }
}

void generateMatrix(ccsMatrix *mtx, int n, int cntInCol) {
    int rows;
    bool exist;
    int nz = cntInCol * n;
    initMatrix(mtx, nz, n);
    for (int col = 0; col < n; col++) {
        for (int row = 0; row < cntInCol; row++) {
            do {
                rows = std::rand() % n;
                exist = false;
                for (int i = 0; i < row; i++)
                    if (rows == mtx->row[col*cntInCol + i])
                        exist = true;
                break;
            } while (exist == true);
            mtx->row[col*cntInCol + row] = rows;
        }
        for (int row = 0; row < cntInCol - 1; row++)
            for (int val = 0; val < cntInCol - 1; val++)
                if (mtx->row[col*cntInCol + val] >
                    mtx->row[col*cntInCol + val + 1]) {
                    int tmp = mtx->row[col*cntInCol + val];
                    mtx->row[col*cntInCol + val] =
                        mtx->row[col*cntInCol + val + 1];
                    mtx->row[col*cntInCol + val + 1] = tmp;
```

```
                }
        }
        for (int val = 0; val < nz; val++)
            mtx->val[val] = (std::rand() % 10000) / 1000.0f;
        int c = 0;
        for (int col = 0; col <= n; col++) {
            mtx->ColIndex[col] = c;
            c += cntInCol;
        }
    }

    void printMatrix(int n, ccsMatrix *mtx) {
        int i;
        int k = mtx[0].NZ;
        std::cout << "\n Value: ";
        for (i = 0; i < k; i++)
            std::cout << " " << mtx[0].val[i];
        std::cout << "\n Row: ";
        for (i = 0; i < k; i++)
            std::cout << " " << mtx[0].row[i];
        std::cout << "\n ColIndex: ";
        for (i = 0; i < n + 1; i++)
            std::cout << " " << mtx[0].ColIndex[i];
        std::cout << std::endl;
    }

    int multMatrix(ccsMatrix *A, ccsMatrix *B, ccsMatrix *C, int size) {
        int n = A->N;
        int NZ = 0, indexA, k, end;
        vector<int> colIndex, row;
        colIndex.push_back(0);
        int *tmp = new int[size];
        for (int i = 0; i < n; i++) {
            memset(tmp, -1, size * sizeof(int));
            for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; j++)
                tmp[A->row[j]] = j;
            for (int j = 0; j < size; j++) {
                indexA = -1;
                k = B->ColIndex[j];
                end = B->ColIndex[j + 1];
                while (indexA == -1 && k < end) {
                    indexA = tmp[B->row[k]];
                    k++;
                }
                if (indexA != -1) {
                    row.push_back(j);
                    NZ++;
                }
            }
            colIndex.push_back(NZ);
        }
        initMatrix(C, NZ, n);
        for (int j = 0; j < NZ; ++j)
            C->row[j] = row[j];
        for (int i = 0; i <= n; ++i)
            C->ColIndex[i] = colIndex[i];
        delete[] tmp;
        return 0;
    }

    int numericMult(int size, const ccsMatrix *A, const ccsMatrix *B, ccsMatrix *C) {
        int n = A->N;
        int index = 0, indexA, startC, finishC, colC;
        int *tmp = new int[size];
        double sum = 0.0;
```

```
    for (int i = 0; i < n; ++i) {
        startC = C->ColIndex[i];
        finishC = C->ColIndex[i + 1];
        if (finishC > startC) {
            memset(tmp, -1, sizeof(int)* size);
            for (int j = A->ColIndex[i]; j < A->ColIndex[i + 1]; ++j)
                tmp[A->row[j]] = j;
            for (int j = startC; j < finishC; ++j, sum = 0) {
                colC = C->row[j];
                for (int k = B->ColIndex[colC]; k < B->ColIndex[colC + 1]; ++k) {
                    indexA = tmp[B->row[k]];
                    if (indexA != -1)
                        sum += A->val[indexA] * B->val[k];
                }
                C->val[index] = sum;
                index++;
            }
        }
    }
    delete[] tmp;
    return 0;
}


int multiplyTbb(ccsMatrix A, ccsMatrix B, ccsMatrix *C) {
    if (A.N != B.N)
        return 1;
    int N = A.N;

    tbb::task_scheduler_init init(4);

    vector<int>* rows = new vector<int>[N];
    vector<double>* value = new vector<double>[N];
    int* columnIndex = new int[N + 1];

    memset(columnIndex, 0, sizeof(int) * N);
    int grainsize = 10;
    tbb::parallel_for(tbb::blocked_range<int>(0, A.N, grainsize),
        multiplicate(A, B, rows, value, columnIndex));
    int NZ = 0;
    for (int i = 0; i < N; i++) {
        int tmp = columnIndex[i];
        columnIndex[i] = NZ;
        NZ += tmp;
    }
    columnIndex[N] = NZ;
    initMatrix(C, NZ, N);
    int count = 0;
    for (int i = 0; i < N; i++) {
        int size = rows[i].size();
        memcpy(&C->row[count], &rows[i][0],
            size * sizeof(int));
        memcpy(&C->val[count], &value[i][0],
            size * sizeof(double));
        count += size;
    }
    memcpy(C->ColIndex, &columnIndex[0], (N + 1) * sizeof(int));

    return 0;
}

int main(int argc, char** argv) {
    int N = (argc != 1) ? atoi(argv[1]) : 1000;
    int cntInCol = (argc != 1) ? atoi(argv[2]) : 500;
    srand((unsigned int)time(0));
```

```
    ccsMatrix A, B, AT, C1, C2;
    generateMatrix(&A, N, cntInCol);
    initMatrix(&AT, A.NZ, A.N);
    transMatrix(A, &AT);
    freeMatrix(&A);
    generateMatrix(&B, N, cntInCol);
    initMatrix(&C1, cntInCol, N);
    tbb::tick_count t1 = tbb::tick_count::now();
    //multiply(AT, B, &C1);
    multMatrix(&AT, &B, &C1, N);
    numericMult(N, &AT, &B, &C1);
    tbb::tick_count t2 = tbb::tick_count::now();
    std::cout << "Time usual is " << (t2 - t1).seconds() << std::endl;

    initMatrix(&C2, cntInCol, N);
    tbb::tick_count tbbT1 = tbb::tick_count::now();
    multiplyTbb(AT, B, &C2);
    tbb::tick_count tbbT2 = tbb::tick_count::now();
    double acc = ((t2 - t1).seconds()) / ((tbbT2 - tbbT1).seconds());
    std::cout << "Time tbb is " << (tbbT2 - tbbT1).seconds() << std::endl;
    std::cout << "Acceleration is " << acc << std::endl;
    std::cout << "Efficienty is " << acc / 4 << std::endl;
    freeMatrix(&AT);
    freeMatrix(&B);
    freeMatrix(&C1);
    freeMatrix(&C2);
    return 0;
}
```