МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
Высшего образования
«Нижегородский государственный университет им. Н.И. Лобачевского»
Национальный исследовательский университет

Институт информационных технологий, математики и механики
Кафедра математического обеспечения и суперкомпьютерных технологий

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ

«Умножение разреженных матриц. Элементы типа double. Формат хранения матрицы – строковый (CRS)»

Выполнил:
студент группы 381706-1
_____Суслов Е. И.

Научный руководитель:
Доцент, канд. техн. наук
_____Сысоев А. В.

Нижний Новгород
2019.

Содержание

1. Введение

Алгебра разреженных матриц (Sparse algebra) – важный раздел математики, имеющий очевидное практическое применение. Разреженные матрицы возникают естественным образом при постановке и решении задач из различных научных и инженерных областей. В частности, при формулировании оптимизационных задач большой размерности с линейными ограничениями матрица системы ограничений нередко оказывается разреженной. Матрицы с существенно преобладающим количеством нулевых элементов формируются при численном решении дифференциальных уравнений в частных производных. Такие матрицы возникают в теории графов.

Эффективные методы хранения и обработки разреженных матриц на протяжении последних десятилетий вызывают интерес у широкого круга исследователей. Выясняется, что для учета разреженной структуры приходится существенно усложнять как методы хранения, так и алгоритмы обработки. Многие тривиальные с точки зрения программирования алгоритмы в разреженном случае становятся весьма сложными.

В данной работе рассматривается один из достаточно показательных алгоритмов – умножение двух разреженных матриц с хранением итогового результата в разреженной матрице. Учитывая сложность и многогранность проблемы, в работе не ставится цель разработать или продемонстрировать оптимальный в некотором смысле алгоритм, равно как и дать полный обзор текущего состояния дел в данной области.

2. Постановка задачи

Цель данной работы:

1. Создания удобной и эффективной структуры хранения разряженных матриц с элементами типа «double» в формате CRS
2. Написать один из возможных алгоритмов умножения разряженных матриц (алгоритм Густавсона), с хранением результата в аналогичной разряженной матрице
3. Реализовать распараллеливание алгоритма Густавсона с использованием средства межпроцессного взаимодействия MPI.
4. Сравнить эффективность последовательной версии с параллельной, а также с другими реализациями перемножения разряженных матриц
5. Реализовать функцию генерации разряженных матриц
6. Проверить правильность алгоритма с помощью «Google Tests» и наглядного представления матриц малой размерности

Будем для упрощения рассматривать задачу для случая квадратных матриц. Пусть A и B – квадратные матрицы размера N x N, в которых процент ненулевых NZ << N x N . Будем считать, что элементы матриц A и B – вещественные числа (далее в программной реализации будем использовать числа с плавающей точкой двойной точности).

3. Метод решения

При запуске программы пользователю предлагается ввести два числа, первое будет использоваться для задания количества строк и столбцов в матрице (квадратной матрице), второе будет использоваться для задания количества ненулевых элементов в каждой строке.

Затем программа выводит два числа:

1. Время работы алгоритма Густавсона
2. Время работы алгоритма Густавсона выполняющегося параллельно

Матрицы будут хранится в CRS формате

Формат хранения, широко распространенный под названием CSR (Compressed Sparse Rows) или CRS (Compressed Row Storage), призван устранить некоторые недоработки координатного представления. Так, попрежнему используются три массива. Первый массив хранит значения элементов построчно (строки рассматриваются по порядку сверху вниз), второй – номера столбцов для каждого элемента, а третий заменяет номера строк, используемые в координатном формате, на индекс начала каждой строки (рис. 3). Отметим, что количество элементов массива RowIndex равно N + 1. При этом элементы строки i в массиве Value находятся по индексам от RowIndex[i] до RowIndex[i + 1] – 1 включительно (i-ый элемент массива RowIndex указывает на начало i-ой строки). Исходя из этого обрабатывается случай пустых строк, а также добавляется «лишний» элемент в массив RowIndex – устраняется особенность при доступе к элементам последней строки. Данный элемент хранит номер последнего ненулевого.

Разреженное матричное умножение элемента матрицы A плюс 1, что соответствует количеству ненулевых элементов NZ. Оценим объем необходимой памяти.

Плотное представление: M = $8N^2$ байт.

В координатном формате: M = 16 NZ байт.

В формате CRS: M = 8 NZ + 4 NZ + 4 (N + 1) = 12 NZ + 4 N + 4. В часто встречающемся случае, когда N + 1 < NZ, данный формат является более эффективным, чем координатный, с точки зрения объема используемой памяти.

4. Схема распараллеливания

4.1 Обзор алгоритмов перемножения разряженных матриц

Можно выделить три основных пути перемножения разряженных матриц A x B:

1. Наивный (долгий так как требует непоследовательного перемещения по памяти и излишнего количества операций)
2. Транспонируем B и перемножаем строки A на строки B (быстрый но требует дополнительного времени на транспонирование и в нашем случае CRS матриц имеет нетривиальную реализацию)
3. Густавсона наиболее быстрый алгоритм суть которого заключается в накоплении частичных сумм в соответствующих ячейках результирующей матрицы C.

Наивный алгоритм далее не будет рассмотрен из-за своей неэффективности.

4.2 Алгоритм транспонирования разряженной матрицы

Если создать нулевую матрицу, а далее добавлять туда по одному элементу, выбирая их из CRS-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 выделить из CRS-матрицы столбец i не так просто. Необходимо другое решение.

Обсуждение проблемы транспонирования разреженной матрицы изложено в книге [2]. Рассмотрим основные идеи описанного в [2] алгоритма, предложенного Густавсоном.

1. Сформируем N одномерных векторов для хранения целых чисел, а также N векторов для хранения вещественных чисел. 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 (Col и Value), попутно формируя массив RowIndex.

4.3 Алгоритм Густавсона

В отличие от алгоритма, рассмотренного ранее, здесь не требуется транспонировать матрицу B. Суть метода заключается в том, чтобы избежать проблемы выделения столбца в матрице B. Для этого предлагается изменить порядок вычислений: вместо того, чтобы умножать строку на столбец, – вычислять произведения каждого элемента матрицы A на все элементы соответствующей строки матрицы B, постепенно накапливая частичные суммы.

Рассмотрим i-ю строку матрицы A. Для всех значений j умножим элемент A[i, j] на все элементы j-ой строки матрицы B. Все произведения будем накапливать в ячейки, соответствующие i-ой строке матрицы C.

По окончании обработки i-ой строки матрицы A оказывается полностью вычисленной i-я строка матрицы C. Это очень удобно при параллельной реализации. Мы просто можем передавать из процесса с нулевым рангом последовательную (т.к. иначе обмен данными между процессами «убьет» производительность, которая в алгоритме нацеленном на общую память и так не на высоте) часть строк матрицы A и всю матрицу B каждому процессу. А затем принять все вычисленные процессами строки матрицы C = A x B обратно в процесс с рангом ноль.

Основная проблема заключается в правильной передачи данных процессом, и возврате подсчитанных строк в правильные позиции массивов структуры хранения CRS матриц

5. Описание программной реализации

Программа включает в себя следующие функции для реализации и демонстрации алгоритма Густавсона в его последовательной и параллельных версиях, для умножения разреженных матриц:

1. void FreeMatrix(crsMatrix* mtx); Освобождение памяти матрицы хранящейся в CRS формате
2. void InitializeMatrix(int N, int NZ, crsMatrix* mtx); Выделение памяти под матрицу CRS ормата
3. double** mult_norm_matr(double** A, double** B, int N); Наивное перемножение матриц
4. double** create_norm_mtr(crsMatrix A); Создание обычной матрицы из матрицы в CRS формате
5. int MultiplicateMPI(crsMatrix* A, crsMatrix* B, crsMatrix* C); Основной алгоритм Густавсона, использующий функционал MPI производящий перемножение матриц представленный в CRS формате
6. void MultiplicateGustafson(crsMatrix A, crsMatrix B, crsMatrix* C); Вспомогательный алгоритм Густавсона, производящий перемножение матриц представленный в CRS формате и использующийся для сравнения производительности со своей версией использующей MPI
7. void GenerateRegularCRS(int for_dif, int N, int cntInRow, crsMatrix* mtx); Создание матрицы в CRS формате с заданным порядком и количеством ненулевых элементов в каждой строке, используется для проверки в тестах
8. void print_norm_mtr(double** norm_mtr, int N); Вывод матрицы после ее преобразования из CRS формата в нормальный, наглядное представление матрицы

Так выглядит последовательная реализация алгоритма Густавсона не претендующая на единственность и оптимальность:

```
void MultiplicateGustafson(crsMatrix A, crsMatrix B, crsMatrix* C) {
    int strok_peredali_is_mtrA = A.N;
    std::vector<double> polnaya_stroka_C(A.N, 0);
    std::vector<double> crs_C_value_full;
    std::vector<int> crs_C_col_full;
```

```
    std::vector<int> crs_C_row_index_full;
    crs_C_row_index_full.push_back(0);
    for (int i = 0; i < strok_peredali_is_mtrA; i++) {
        for (int j = A.RowIndex[i]; j < A.RowIndex[i + 1]; j++) {
            int Col_elem_A = A.Col[j];
            for (int k = B.RowIndex[Col_elem_A]; k < B.RowIndex[Col_elem_A +
1]; k++) {
                polnaya_stroka_C[B.Col[k]] += A.Value[j] * B.Value[k];
            }
        }
        for (int k = 0; k < A.N; k++) {
            if (polnaya_stroka_C[k] != 0) {
                crs_C_value_full.push_back(polnaya_stroka_C[k]);
                crs_C_col_full.push_back(k);
                polnaya_stroka_C[k] = 0;
            }
        }
        crs_C_row_index_full.push_back(crs_C_value_full.size());
    }
    int row_index_size = crs_C_row_index_full.size();
    C->N = A.N;
    int add_elem_vsego = static_cast<double>(crs_C_value_full.size());
    C->NZ = add_elem_vsego;
    C->Col = new int[add_elem_vsego];
    C->Value = new double[add_elem_vsego];
    C->RowIndex = new int[row_index_size];
    for (int i = 0; i < add_elem_vsego; i++) {
        C->Col[i] = crs_C_col_full.at(i);
        C->Value[i] = crs_C_value_full.at(i);
    }
    for (int i = 0; i < row_index_size; i++) {
        C->RowIndex[i] = crs_C_row_index_full.at(i);
    }
      }
```

Т.о. мы создаем пустой вектор и начинаем накапливать в нем частичные суммы, которые получаются перемножением одной выбранной строки матрицы A на все строки матрицы B, в результате после одного такого прохода у нас получается полностью вычисленной строка результирующей матрицы C, соответствующая выбранной строке матрицы A, таким образом, пройдя все строки матрицы A, мы получим полностью вычисленную матрицу C (A x B = C).

6. Подтверждение корректности

Проверка была реализована с помощью 6-ти «Google Test», при этом результаты (для сравнения) в одну из функций проверки, предоставляемой этой программой тестирования, передавались из наглядно проверенной функции на малых значениях размерности параметров, а также на больших уже с помощью наивной реализации умножения.

Также все массивы получающейся матрицы в CRS формате передавались для сравнения с той же матрицей но вычисленной в последовательной реализации алгоритма Густавсона. Для нахождения ошибок проверка каждого из массивов была выделена в отдельный тест.

7. Результаты экспериментов

Вычислительные эксперименты умножения разреженных матриц с использованием технологии MPI проводились в конфигурации описанной в таблице 1.

Таблица 1. Тестовая инфраструктура

Процессор	Ibtel(R) Core™ i5 CPU M460 @ 2.53GHz
Память	4 Гб
Операционная система	Windows 10 Домашняя (x64)
Среда разработки	Visual Studio 2019
Режим сборки	Release
Разрядность решения	x32

Таблица 2. Результаты экспериментов: Густавсон + MPI (Z = 20)

Порядок матриц (N)	1 процесс, t (сек.)	8 процессов, t (сек.)	Ускорение (раз)
10000	0.40823	0.292845	1,394
20000	1.22367	0.992757	1,232
30000	1.9102	1.31953	1,447
40000	3.38818	2.09855	1,615

По результатам экспериментов, проводимых для матриц порядка N с Z ненулевыми элементами в каждой строке, можно что при возрастании порядка матрицы мы получаем прирост производительности параллельного алгоритма Густавсона , выполняемого на восьми процессах, по отношению к аналогичному алгоритму исполняемого последовательно.

Исключением является выполнение на порядке матриц 20000, это можно объяснить тем, что затраты времени на вычисления частичных сумм и их отправку увеличиваются, из-за увеличения порядка матриц, но деление строк матрицы A по процессам не приносит значительного ускорения по отношению к предыдущему порядку матриц - 10000.

Таблица 3. Результаты экспериментов: Густавсон + MPI (N = 20000)

Количество ненулевых элементов в строке (Z)	1 процесс, t (сек.)	8 процессов, t (сек.)	Ускорение (раз)
10	0.78744	0.668174	1,178495
20	1.12164	0.813817	1,378245
30	3.39878	1.7988	1,889470
40	3.46956	2.27214	1,527000

По результатам экспериментов, проводимых для матриц порядка N с Z ненулевыми элементами в каждой строке, можно что при возрастании количества ненулевых элементов в каждой строке матрицы мы получаем прирост производительности параллельного алгоритма Густавсона , выполняемого на восьми процессах, по отношению к аналогичному алгоритму исполняемого последовательно.

Как и в предыдущем эксперименте, ускорение умножения с использованием возможностей технологии MPI существует на любых входных данных матриц, но его увеличение не во всех экспериментах пропорционально увеличению элементов матрицы CRS формата.

8. Заключение

В данной работе была реализована программа, решающая задачу умножения разряженных матриц с числами типа «double» и способом хранения CRS.

Для этого были созданы:

1. Заголовочный файл хранящий прототипы функций и саму структуру хранения CRS матриц (crsMatrix) - «matrix_crs.h»
2. Файл реализации функций прототипы которых заданны в заголовке - «matrix_crs.cpp»
3. Файл кода проверкой написанных функций «main.cpp». Проверка была реализована с помощью 6 «Google Tests», при этом результаты (для сравнения) в одну из функций проверки, предоставляемой этой программой тестирования, передавались из наглядно проверенной функции на малых значениях размерности параметров, а также на больших уже с помощью наивной реализации умножения.

9. Список литературы

1. Баркалов К.А. Методы параллельных вычислений. Н. Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2011.
2. Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984.
3. Писсанецки С. Технология разреженных матриц. — М.: Мир,1988.
4. [https://software.intel.com/sites/default/files/m/d/4/1/d/8/LW_SparseMM_doc.pdf]

7. Приложения

7.1. Код «matrix_crs.h»

```
#include <vector>

struct crsMatrix {
    int N;
    int NZ;
    double* Value;
    int* Col;
    int* RowIndex;
};
void InitializeMatrix(int N, int NZ, crsMatrix* mtx);
void FreeMatrix(crsMatrix* mtx);
double** mult_norm_matr(double** A, double** B, int N);
double** create_norm_mtr(crsMatrix A);
int MultiplicateMPI(crsMatrix* A, crsMatrix* B, crsMatrix* C);
void create_part_crs_C(int row_peredali, crsMatrix* A, crsMatrix* B, crsMatrix* C);
void MultiplicateGustafson(crsMatrix A, crsMatrix B, crsMatrix* C);
void GenerateRegularCRS(int N, int cntInRow, crsMatrix* mtx)
```

7.2 Код «matrix_crs.cpp»

```
void InitializeMatrix(int N, int NZ, crsMatrix* mtx) {
	mtx->N = N;
	mtx->NZ = NZ;
	mtx->Value = new double[NZ];
	mtx->Col = new int[NZ];
	mtx->RowIndex = new int[N + 1];
}

void FreeMatrix(crsMatrix* mtx) {
	delete[] mtx->Value;
	delete[] mtx->Col;
	delete[] mtx->RowIndex;
}

void GenerateRegularCRS(int N, int cntInRow, crsMatrix* mtx) {
	std::mt19937 mersenne;
	mersenne.seed(static_cast<unsigned int>(time(0)));
	int i, j, k, f, tmp, notNull, c;
	notNull = cntInRow * N;
	InitializeMatrix(N, notNull, mtx);
	for (i = 0; i < N; i++) {
		for (j = 0; j < cntInRow; j++) {
			do {
				mtx->Col[i * cntInRow + j] = mersenne() % N;
				f = 0;
				for (k = 0; k < j; k++)
					if (mtx->Col[i * cntInRow + j] ==
						mtx->Col[i * cntInRow + k])
						f = 1;
			} while (f == 1);
		}
		for (j = 0; j < cntInRow - 1; j++)
```

```
                for (k = 0; k < cntInRow - 1; k++)
                    if (mtx->Col[i * cntInRow + k] > mtx->Col[i * cntInRow + k + 1]) {
                        tmp = mtx->Col[i * cntInRow + k];
                        mtx->Col[i * cntInRow + k] =
                            mtx->Col[i * cntInRow + k + 1];
                        mtx->Col[i * cntInRow + k + 1] = tmp;
                    }
        }
        for (i = 0; i < cntInRow * N; i++)
            mtx->Value[i] = (static_cast<double>(mersenne() % 1000)/100);
        c = 0;
        for (i = 0; i <= N; i++) {
            mtx->RowIndex[i] = c;
            c += cntInRow;
        }
}

// void MultiplicateGustafson(crsMatrix A, crsMatrix B, crsMatrix* C)

int MultiplicateMPI(crsMatrix* A, crsMatrix* B, crsMatrix* C) {
        int size, rank;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        if (size == 1) {
            MultiplicateGustafson(*A, *B, C);
            return 0;
        }
        int* upakovka;
        upakovka = new int[3];
        int row_on_proc, ostatok_last_proc, vsego_elem_mtr_B;
        if (rank == 0) {
            C->N = A->N;
            row_on_proc = A->N / size;
            ostatok_last_proc = A->N % size;
            vsego_elem_mtr_B = B->RowIndex[B->N];
            upakovka[0] = A->N;
            upakovka[1] = vsego_elem_mtr_B;
            upakovka[2] = row_on_proc;
        }
        MPI_Bcast(&upakovka[0], 3, MPI_INT, 0, MPI_COMM_WORLD);
        if (rank != 0) {
            row_on_proc = upakovka[2];
            vsego_elem_mtr_B = upakovka[1];
            int N = upakovka[0];
            A->N = N;
            B->N = N;
            int size_row_index = N+1;
            A->RowIndex = new int[row_on_proc+1];
            B->Col = new int[vsego_elem_mtr_B];
            B->Value = new double[vsego_elem_mtr_B];
            B->RowIndex = new int[size_row_index];
        }
        MPI_Bcast(B->Col, vsego_elem_mtr_B, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(B->Value, vsego_elem_mtr_B, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        MPI_Bcast(B->RowIndex, B->N+1, MPI_INT, 0, MPI_COMM_WORLD);
        if (rank == 0) {
            for (int i = 1; i < size; i++) {
                int otsilka = A->RowIndex[row_on_proc * i + ostatok_last_proc];
                MPI_Send(&A->RowIndex[row_on_proc * i + ostatok_last_proc], row_on_proc + 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
                int kol =
A->RowIndex[row_on_proc*(i+1)+ostatok_last_proc]-A->RowIndex[row_on_proc*i+ostatok_last_proc];
```

```
            MPI_Send(&A->Col[otsilka], kol, MPI_INT, i, 1, MPI_COMM_WORLD);
            MPI_Send(&A->Value[otsilka], kol, MPI_DOUBLE, i, 2, MPI_COMM_WORLD);
        }
        crsMatrix c;
        crsMatrix* C_temp = &c;
        create_part_crs_C(row_on_proc + ostatok_last_proc, A, B, C_temp);
        int* massiv_elem_proc_helperov;
        massiv_elem_proc_helperov = new int[size];
        massiv_elem_proc_helperov[0] = C_temp->RowIndex[ostatok_last_proc + row_on_proc - 1];
        MPI_Status st;
        for (int i = 1; i < size; i++) {
            MPI_Recv(&massiv_elem_proc_helperov[i], 1, MPI_INT, i, 5, MPI_COMM_WORLD, &st);
        }
        int vsego_elem_so_vseh_proc = 0;
        for (int i = 0; i < size; i++) {
            vsego_elem_so_vseh_proc += massiv_elem_proc_helperov[i];
        }
        C->RowIndex = new int[A->N + 1];
        C->Col = new int[vsego_elem_so_vseh_proc];
        C->Value = new double[vsego_elem_so_vseh_proc];
        C->RowIndex[0] = 0;
        for (int i = 0; i < massiv_elem_proc_helperov[0]; i++) {
            C->Col[i] = C_temp->Col[i];
            C->Value[i] = C_temp->Value[i];
        }
        for (int i = 1; i <= ostatok_last_proc + row_on_proc; i++) {
            C->RowIndex[i] = C_temp->RowIndex[i-1];
        }
        MPI_Status s1, s2, s3;
        for (int i = 1; i < size; i++) {
            int chto = row_on_proc * i + ostatok_last_proc + 1;
            MPI_Recv(&C->RowIndex[chto], row_on_proc, MPI_INT, i, 1, MPI_COMM_WORLD, &s1);
            for (int j = 0; j < row_on_proc; j++) {
                C->RowIndex[chto +j] += C->RowIndex[row_on_proc * i + ostatok_last_proc];
            }
            int help = massiv_elem_proc_helperov[i];
            int index_prinytiya = C->RowIndex[row_on_proc * i + ostatok_last_proc];
            MPI_Recv(&C->Col[index_prinytiya], help, MPI_INT, i, 2, MPI_COMM_WORLD, &s2);
            MPI_Recv(&C->Value[index_prinytiya], help, MPI_DOUBLE, i, 3, MPI_COMM_WORLD, &s3);
        }
        FreeMatrix(C_temp);
        delete[] massiv_elem_proc_helperov;
    } else {
        MPI_Recv(&A->RowIndex[0], row_on_proc + 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUSES_IGNORE);
        int sdvig_strok_k_starty = A->RowIndex[0];
        for (int i = 0; i < row_on_proc + 1; i++) {
            A->RowIndex[i] -= sdvig_strok_k_starty;
        }
        A->Col = new int[A->RowIndex[row_on_proc]];
        A->Value = new double[A->RowIndex[row_on_proc]];
        MPI_Status status1, status2;
        MPI_Recv(&A->Col[0], A->RowIndex[row_on_proc], MPI_INT, 0, 1, MPI_COMM_WORLD, &status1);
        MPI_Recv(&A->Value[0], A->RowIndex[row_on_proc], MPI_DOUBLE, 0, 2, MPI_COMM_WORLD,
&status2);
        create_part_crs_C(row_on_proc, A, B, C);
        int sozdali_elem = C->RowIndex[row_on_proc - 1];
        MPI_Send(&sozdali_elem, 1, MPI_INT, 0, 5, MPI_COMM_WORLD);
        MPI_Send(&C->RowIndex[0], row_on_proc, MPI_INT, 0, 1, MPI_COMM_WORLD);
        MPI_Send(&C->Col[0], sozdali_elem, MPI_INT, 0, 2, MPI_COMM_WORLD);
        MPI_Send(&C->Value[0], sozdali_elem, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
        FreeMatrix(A);
```

```
            FreeMatrix(B);
        }
        MPI_Barrier(MPI_COMM_WORLD);
        if (rank != 0) {
            FreeMatrix(C);
        }
        return 0;
}

void create_part_crs_C(int row_peredali, crsMatrix* A, crsMatrix* B, crsMatrix* C) {
        int strok_peredali_is_mtrA = row_peredali;
        std::vector<double> polnaya_stroka_C(A->N, 0);
        std::vector<double> crs_C_value_full;
        std::vector<int> crs_C_col_full;
        std::vector<int> crs_C_row_index_full;
        for (int i = 0; i < strok_peredali_is_mtrA; i++) {
            for (int j = A->RowIndex[i]; j < A->RowIndex[i + 1]; j++) {
                int Col_elem_A = A->Col[j];
                for (int k = B->RowIndex[Col_elem_A]; k < B->RowIndex[Col_elem_A + 1]; k++) {
                    polnaya_stroka_C[B->Col[k]] += A->Value[j] * B->Value[k];
                }
            }
            for (int k = 0; k < A->N; k++) {
                if (polnaya_stroka_C[k] != 0) {
                    crs_C_value_full.push_back(polnaya_stroka_C[k]);
                    crs_C_col_full.push_back(k);
                    polnaya_stroka_C[k] = 0;
                }
            }
            crs_C_row_index_full.push_back(crs_C_value_full.size());
        }
        int row_index_size = crs_C_row_index_full.size();
        C->N = A->N;
        int add_elem_vsego = crs_C_value_full.size();
        C->Col = new int[add_elem_vsego];
        C->Value = new double[add_elem_vsego];
        C->RowIndex = new int[row_index_size];
        for (int i = 0; i < add_elem_vsego; i++) {
            C->Col[i] = crs_C_col_full.at(i);
            C->Value[i] = crs_C_value_full.at(i);
        }
        for (int i = 0; i < row_index_size; i++) {
            C->RowIndex[i] = crs_C_row_index_full.at(i);
        }
}

double** create_norm_mtr(crsMatrix A) {
        double** norm_mtr;
        norm_mtr = new double*[A.N];
        for (int j = 0; j < A.N; j++) {
            norm_mtr[j] = new double[A.N];
        }
        for (int j = 0; j < A.N; j++) {
            for (int i = 0; i < A.N; i++) {
                norm_mtr[i][j] = 0;
            }
        }
        for (int i = 0; i < A.N; i++) {
            for (int j = A.RowIndex[i]; j < A.RowIndex[i + 1]; j++) {
                norm_mtr[i][A.Col[j]] = A.Value[j];
            }
        }
```

```
	return norm_mtr;
}

double** mult_norm_matr(double** A, double** B, int N) {
	double** C;
	C = new double* [N];
	for (int j = 0; j < N; j++) {
		C[j] = new double[N];
	}
	for (int i = 0; i < N; i++) {
		for (int j = 0; j < N; j++) {
			C[i][j] = 0;
			for (int k = 0; k < N; k++)
				C[i][j] += A[i][k] * B[k][j];
		}
		std::cout << std::endl;
	}
	return C;
}
```