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

Отчет по лабораторной работе

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

Выполнил:
студент группы 381906-3
Олюнин А.В.
Проверил:
доцент кафедры МОСТ,
кандидат технических наук
Сысоев А.В.

Нижний Новгород
2022

Оглавление

Введение

Изучение разреженных матриц имеет полезный не только исследовательский, но и прикладной характер. Нередко при решении разного рода задач из разных научных и инженерных областей можно столкнуться с непосредственной работой с разреженными матрицами. Далеко ходить не надо: при решении задачи оптимизации большой размерности с линейными ограничениями, матрицы ограничений нередко обладают преимущественно нулевыми элементами. Такие матрицы появляются, в том числе при решении дифференциальных уравнений в частных производных, особо применяемых в области физических моделей. Также в теории графов несложно с ними столкнуться.

Сложность проявляется в том, что не так-то просто выделить четкий критерий, по которому матрицу можно было бы считать разреженной. Причиной является неоднозначность таких критериев, так как они зависят не только от количества ненулевых элементов в матрице, но и от ряда других характеристик, таких как ее размер, распределение ненулевых элементов в матрице (равномерное, вдоль главной диагонали или), архитектура вычислительной системы, используемые алгоритмы хранения и выполнения операций с матрицей. Последние нацелены на получение максимальной выгоды из структуры матрицы.

Эффективные методы хранения и обработки разреженных матриц находят интерес у современных исследовтелей. Однако для учета особенностей матрицы обычно приходится прибегать к их усложнению. В данной работе рассматривается один из таких подходов - столбцовый метод хранения матрицы CCS, который приводит к модификации методов умножения разреженных матриц. Для анализа сложности таких вычислений воспользуемся технологиями разработки параллельных программ в системах с общей памятью.

Постановка задачи

Цель данной лабораторной состоит изучении эффективного метода хранения разреженной матрицы и вытекающих из него модификаций алгоритмов работы с ней. Перед нами стоит задача разработки программного кода, реализующего последовательную и параллельную обработку такой структуры матрицы. Распараллеливание программы необходимо выполнить с использованием разных подходов в системе с общей памятью: технологии OpenMP, библиотеки Intel Thread Building Blocks (TBB) и класса std::thread стандарта C++11. Кроме того, необходимо проверить решение на верность получаемого результата, провести качественное сравнение повышения производительности вычислений относительно скорости работы программы между последовательным и параллельным алгоритмом. Для достижения поставленной задачи необходимо использовать программное обеспечение тестирования программ Google Test.

Описание алгоритма

В данной работе для хранения разреженных матриц воспользуемся хорошо известным методом хранения, учитывающим их особенности - столбцовым методом хранения (CCS - Compressed Column data Structure). Он подразумевает хранение следующей информации о матрице:

1. Размер матрицы - size. Обычно предполагается, что матрица квадратная.
2. Количество ненулевых элементов в матрице - non.
3. Массив значений ненулевых элементов матрицы - Entry.
4. Массив индексов строк, в которых располагаются ненулевые элементы матрицы - irows. Причем, его значения упорядочены по столбцам и находятся в порядке возрастания.
5. Массив информации о расположении начала каждой первой записи о столбце в массивах Entry и irows - shtcols.

Таким образом, благодаря массивам irows и shtcols формируется полное представление о размещении ненулевых элементов в разреженной матрице. Для наглядности и уточнения деталей приведем пример использования этого подхода. Пусть нам дана матрица, содержащая комплексные элементы:

$$\begin{pmatrix} 4+20i & 0 & 0 & 2+7i \\ 0 & 0 & 18+4i & 0 \\ i & 0 & 15+11i & 0 \\ 0 & 8+4i & 0 & 0 \end{pmatrix}$$

Тогда массивы Entry, irows и shtcols заполнятся значениями: Entry = {4+20i, i, 8+4i, 18+4i, 15+11i, 2+7i}, irows = {1, 3, 4, 2, 3, 1}, shtcols = {1, 3, 4, 6, 7}. Заметим, что размер массивов Entry и irows равны количеству ненулевых элементов. Размер массива shtcols всегда равен size+1. Entry заполнен последовательно относительно столбцов, irows также упорядочен относительно стобцов и, также как shtcols, использует значения индексов строк начиная с 1.

Перейдем к рассмотрению операции умножения разреженных матриц. Для того, чтобы перемножить две матрицы необходимо рассмотреть их строки и столбцы. Далее провести операцию скалярного произведения данных

двух векторов. Столбцовый метод хранения матрицы, с одной стороны, упрощает эту операцию, так как не имеет никакого смысла хранить, умножать и накапливать нули. Однако возникают сложности другого характера.

Во-первых, необходимо получить доступ к векторам матриц. В случае столбцов все просто. Формат хранения предусматривает легкое получение значений столбца. Для этого достаточно взять расположение начала первого и последнего элемента в массиве shtcols, а затем получить прямой доступ к значениям элементов и номерам строк в массивах Entry и irows. Со строками все гораздо сложнее. Чтобы найти элементы всего одной из строк необходимо полностью просмотреть массив shtcols, выбирая только нужные элементы. Одним из простых решений этой проблемы является транспонирование матрицы. Так как доступ к строкам необходим только в первой матрице при операции произведения, то транспонировать можно только ее. Транспонирование является вполне неплохим решением в рамках данной работы, так как время, затраченное на транспонирование матрицы существенно невелико по сравнению с основным алгоритмом умножения, а рассматриваемые матрицы не слишком большие, что позволяет хранить в памяти еще одну дополнительную матрицу. В ситуациях посложнее можно выбрать более эффективный подход.

Во-вторых, так как хранение нулей не подразумевается то, чтобы находить пары пересечения элементов векторов, которые необходимо умножить, придется использовать дополнительные операции и память. Как следствие, в алгоритме появятся ветвления.

В-третьих, необходимо учитывать формат хранения результирующей матрицы. Так как метод ее хранения тоже столбцовый, то наиболее выгодно при умножении получать элементы по столбцам. Для этого нужно фиксировать столбец второй матрицы и последовательно умножать его на строки первой матрицы, получая тем самым значения столбца результирующей матрицы.

Выше было упомянуто об использовании операции транспонирования. Однако транспонировать разреженную матрицу A в столбцовом формате хранения за приемлемое время выполнения не такая уж и тривиальная задача. Поэтому, кратко опишем алгоритм, который будет использован в рамках данной работы. Ключевая идея алгоритма состоит в использовании структур данных новой транспонированной матрицы A^T для хранения промежуточных результатов вычислений.

1. Массив shtcols новой матрицы заполняется нулями. Далее просматривается массив irows матрицы A и подсчитывается количество элементов

в каждой строке. Результат вычислений сохраняется в массиве shtcols матрицы A^T.

2. Массив shtcols матрицы A^T заполняется индексами начала информации о каждом столбце, исходя из предыдущего шага, уже известно сколько в каждом столбце элементов.

3. Остается найти позицию ненулевых элементов в новой матрице A^T. Это не сложно, так как номера столбцов уже известны. Для этого значение старого индекса строки используется для получения индекса нового столбца, занести значение в который можно благодаря массиву shtcols матрицы A^T, который по своей сути хранит указатель на то, куда нужно вносить. Единственное - необходимо смещать этот указатель вперед, увеличивая его на один при каждой новой вставке.

Таким образом, были рассмотрены основные положения перемножения разреженных матриц. Теперь, в связи с тем, что умножение матриц представляет собой набор скалярных произведений векторов, опишем оптимальный, в нашем случае, алгоритм их произведения. Пусть необходимо перемножить два вектора a и b. Алгоритм разреженного вычисления следующий:

1. Вычисление стандартным способом требует N^2 просмотров массива. В нашем случае, для сокращения алгебраических операций удобно во время работы хранить дополнительный расширенный до размера N массив указателей ip. В начале он заполнен нулями.

2. Просматриваются все элементы вектора a, размер которого во много раз меньше N, так как матрица разрежена. Исходя из его значений заполняется массив ip, а именно в соответствующий элемент массива ip вписывается индекс массива a. Это позволяет хранить в массиве ip индексы позиций ненулевых элементов в векторе a и передать информацию о паре пересекающихся элементов, которые необходимо перемножить и записать в частичную сумму.

3. В конце просматривается вектор b. В случае, если соответствующий элемент массива ip не является нулевым, что означает совпадение позиций элементов в векторах a и b, то вычисляется произведение элементов массивов a и b, а результат сохраняется в частичную сумму. Результат скалярного произведения получается после полного просмотра массива b.

Приведенный выше алгоритм произведения векторов особенно эффективен при скалярном умножении одного вектора на несколько других, так как в

этом случае выполнить первые два пункта достаточно всего один раз. Такая ситуация возникает в рамках данной лабораторной работы при умножении разреженных матриц.

Описание схемы распараллеливания

В отличие от транспонирования, умножение матриц занимает существенно больше времени. К счастью, операции скалярного умножения векторов матриц могут быть выполнены независимо друг от друга. Чтобы решить проблему сбора данных достаточно ввести несколько массивов в общей памяти под соответствующие характеристики хранения разреженной матрицы.

Получается, для того, чтобы распараллелить перемножение матриц достаточно распределить вычисления скалярных произведений векторов между всеми имеющимися потоками. Так как выгодной стратегией умножения матриц при столбцовом методе хранения является фиксирование столбца второй матрицы и умножение его на все строки первой матрицы, то распределение необходимо выполнить в рамках соблюдения этого условия, то есть просто распределить вычисления по столбцам второй матрицы, отдав каждому потоку по его части.

В OpenMP реализации столбцы распределяются статически компилятором. То есть каждый поток получает несколько подряд идущих столбцов, что очень удобно в данной задаче, так как предугадать заранее количество ненулевых элементов в каждом новом столбце не представляется возможным и важно обеспечить последовательность полученных данных. Далее каждый поток выполняет свою часть вычислений с выделенными ему столбцами. Результат вычислений сохраняется в общей памяти уже после выполнения всех вычислений каждым потоком благодаря параллельной секции в OpenMP. В таком случае требуется выделение памяти под массив в общей памяти размером равным числу потоков, которое известно до выполнения параллельной секции. После параллельных вычислений требуется сбор данных с этого массива и формирование результата.

В TBB реализации столбцы распределяются динамически при помощи планировщика данной библиотеки. В таком случае каждый поток получит заранее неизвестные номера столбцов. Чтобы обеспечить верность результата придется увеличивать массив в общей памяти в сравнении с OpenMP. Теперь его размер станет равным количеству столбцов в матрице. Промежуточную запись результатов придется делать также чаще, после каждого вычисления нового столбца. Сбор результатов и формирование результата займет также больше итераций.

В std::thread реализации столбцы распределяются вручную программистом. Так как выгодно обеспечить последовательное получение данных, ведь

заранее число ненулевых элементов в каждом новом столбце не известно, с точки зрения уменьшения размера массива в общей памяти, а как следствие уменьшение количества итераций, то распределение столбцов произведено как в реализации OpenMp. Таким образом, массив в общей памяти имеет размер равный количеству потоков, которое известно до выполнения параллельных вычислений. В рамках различия между данной версией и OpenMP реализацией было решено не выделять приватную память под промежуточные вычисления, поэтому запись в массив в общей памяти необходимо производить после каждого вычисления скалярного произведения. Сбор данных и формирование результата не сложнее, чем в OpenMp.

Таким образом, была достигнута эффективность выполнения операции умножения разреженных матриц. На первый взгляд кажется, что фаворит очевиден, но на самом деле все не так однозначно, все реализации имеют свои плюсы и минусы, в чем мы убедимся при выполнении практических тестов.

Описание программной реализации

Программная реализация умножения разреженных матриц содержит: класс комплексных чисел, рассматриваемых в данной лабораторной работе, класс матриц, методом хранения которых является столбцовый.

Опишем наиболее важные реализации методов этих классов:

1. Метод умножения комплексных чисел

```
Complex Complex::operator*(Complex Tmp)
```

Описание: реализует обычное умножение комплексных чисел по известным формулам.

2. Метод сравнения комплексного числа с нулем

```
bool Complex::IsNotZero()
```

Описание: реализует сравнение комплексного числа с нулем с заданной точностью.

3. Метод генерации матрицы

```
Matrix& Matrix::RandomMatrix(int size, int dist, int cnt, int seed)
```

Описание: реализует равномерную генерацию случайной матрицы. В каждом столбце матрицы cnt ненулевых элементов.

4. Метод траспонирования матрицы

```
Matrix Matrix::T()
```

Описание: реализует алгоритм транспонирования разреженной матрицы столбцового типа хранения.

5. Оператор умножения

```
Matrix Matrix::operator*(Matrix B)
```

Описание: реализует умножение двух разреженных матриц столбцового типа хранения.

Тестирование параллельной программы

Для того, чтобы убедиться в корректности работы программы, производились тесты на базе платформы Google Tests. Более того, результат умножения разреженных матриц, содержащих комплексные числа был проверен на результатах онлайн калькулятора для небольших матриц размера 4 на 4 и 5 на 5. В ходе тестирования результат оказался верным, ошибок не найдено. Поэтому, в рамках данной лабораторной работы можно считать, что работа программы является верной с некоторой точностью.

Для оценки эффективности решения проводились тесты на производительность последовательной и параллельной реализаций. В первом случае размер матрицы был выбран 1000, в каждом столбце которой не более 1% ненулевых элементов, итого 10 элементов в каждом столбце. Во втором случае размер матрицы был выбран 2500, в каждом столбце которой не более 1% ненулевых элементов, итого 25 элементов в каждом столбце. В реализациях OpenMP и std::thread число потоков задавалось заранее равным 8.

Тесты были выполнены на базе следующей конфигурации системы:

- Процессор: 11th Intel Core i5-11400 2.60GHz 2.59 GHz Кол-во ядер: 6, Кол-во потоков: 12
- Операционная система: Windows 10 Домашняя
- Тип системы: 64-разрядная операционная система, процессор x64

Параллельная реализация	Время работы последовательного решения (сек.)	Время работы (сек.)	Эффективность (раз)
OMP	0.129207	0.0412561	3.13258
TBB	0.106	0.041	2.58537
STD	0.109	0.042	2.59524

Таблица 1: Случай 1

Параллельная реализация	Время работы последовательного решения (сек.)	Время работы (сек.)	Эффективность (раз)
OMP	1.48627	0.572603	2.59564
TBB	1.586	0.479	3.31106
STD	1.599	0.509	3.14145

Таблица 2: Случай 2

Анализ таблицы может привести нас к результату, что однозначного победителя в этой гонке нет. Более того, на самом деле результат каждого запуска может заметно различаться. Дело в том, что так как элементы матрицы находятся в неопределенных позициях, то заранее сказать сколько операций достанется каждому потоку попросту невозможно. Поэтому может получиться так, что одному из потоков досталось заметно больше операций. А так как матрица разреженная, то невезение потока заметно скажется на времени работы программы. Кроме того, очень сложно определить, будет ли скалярное произведение равно нулю. Из-за чего компилятору очень сложно оптимизировать такое условие и предсказывать его заранее.

Описанные факторы тем сильнее проявляют себя, чем больше размер матрицы. Поэтому при больших матрицах реализация TBB обычно показывает результат заметно лучше, чем остальные реализации. Все благодаря динамическому распределению итераций планировщиком и помощи подбору grainsize специально под данный тест. Grainsize равен 1/5 от размера матрицы. К тому же, сбор данных из общей памяти происходит по линейному алгоритму и теряет свою долю времени выполнения как раз при увеличении размера матрицы. Однако, работа планировщика требует накладных расходов, из-за чего на слишком малых матрицах результат TBB наоборот обычно заметно хуже остальных реализаций.

Реализация std::thread не многим отличается от OpenMP. Программист распределил данные между потоками по аналогичному правилу. Однако все же std::thread показывает результат ближе к TBB. Из чего можно сделать вывод, что использование частной памяти потока тоже становится менее эффективной при увеличении размера матриц, по сравнению с частой работой с общей памятью.

Заключение

В данной лабораторной работе был рассмотрен один из методов оптимизации хранения разреженных матриц с элементами комплексного типа, то есть столбцовый метод. Рассмотрены модификации операции умножения матриц, в частности? использование транспонирования, алгоритм которого был кратко сформулирован в соответствующей главе.

В ходе работы была написана программная реализация с использованием нескольких подходов к параллельному программированию в системах с общей памятью, а именно технологии OpenMP, библиотеки Intel TBB и класса std::thread. Далее была произведена пара тестов, по результатам которых можно сделать вывод о том, что у всех подходов есть свои плюсы и минусы даже в рамках одной задачи. В случае умножения небольших матриц более целесообразным решением будет использовать статическое распределение операций между процессами и частную память потоков, а в случае умножения больших матриц использовать динамическое распределение операций между потоками и более частую работу с общей памятью.

Литература

Источники информации

[1] Сысоев А. В., Мееров И. Б., Свистунов А. Н., Курылев А. Л., Сенин А. В., Шишков А. В., Корняков К. В., Сиднев А. А. Параллельное программирование в системах с общей памятью. Инструментальная поддержка, Нижний Новгород, 2007.

[2] Мееров И.Б., Сысоев А.В. при участии Сафоновой Я. Разреженное матричное умножение, Нижний Новгород, 2011.

[3] Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений, Москва «Мир», 1984.

[4] Тьюарсон Р. Разреженные матрицы, Москва «Мир», 1977.

Ресурсы сети интернет

[5] Compressed Column (CC) Sparse Matrix File Format URL: https://people.sc.fsu.edu/ jburkardt/data/cc/cc.html

[6] Обработка разреженных матриц URL: http://wwwcdl.bmstu.ru/iu7/book1/stage6.htm

Приложение

В разделе приводится код, написаный в рамках данной лабораторной работы.

Последовательная версия

```
// mult_sparse_cc_complex_mat.h
// Copyright 2022 Olynin Alexander
#ifndef
    MODULES_TASK_1_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_
#define
    MODULES_TASK_1_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_

#include <time.h>
#include <random>
#include <vector>

class Complex {
 private:
    double rl;
    double im;
 public:
    explicit Complex(double _rl = 0, double _im = 0): rl(_rl), im(_im) {}
    Complex(const Complex& Tmp): rl(Tmp.rl), im(Tmp.im) {}
    Complex& operator=(Complex Tmp);
    std::vector<Complex> InitVec(std::vector<double> rls =
                                 std::vector<double>(),
                                 std::vector<double> ims =
                                 std::vector<double>());
    double GetRl() { return this->rl; }
    double GetIm() { return this->im; }
    void SetRl(double tmp) { this->rl = tmp; }
    void SetIm(double tmp) { this->im = tmp; }
    Complex operator+(Complex Tmp);
    Complex& operator+=(Complex Tmp);
    Complex operator*(Complex Tmp);
    bool operator==(Complex Tmp);
    bool operator!=(Complex Tmp) { return !(*this == Tmp); }
    bool IsNotZero();
    ~Complex() {}
};

class Matrix {
 private:
    int size;
    int non;
    std::vector<Complex> Entry;
    std::vector<int> irows;
    std::vector<int> shtcols;

 public:
    Matrix(int _size = 0, int _non = 0, const std::vector<Complex>& _Entry =
           std::vector<Complex>(), const std::vector<int>& _irows =
```

```
            std::vector<int>(), const std::vector<int>& _shtcols =
            std::vector<int>()): size(_size), non(_non), Entry(_Entry),
            irows(_irows), shtcols(_shtcols) {}
    Matrix(const Matrix& Tmp): size(Tmp.size), non(Tmp.non), Entry(Tmp.Entry),
                               irows(Tmp.irows), shtcols(Tmp.shtcols) {}
    Matrix& operator=(const Matrix& Tmp);
    int GetSize() { return this->size; }
    int GetNon() { return this-> non; }
    std::vector<Complex> GetEntry() { return this->Entry; }
    std::vector<int> GetIrows() { return this->irows; }
    std::vector<int> GetShtcols() { return this->shtcols; }
    Matrix& ClearMatrix();
    Matrix& RandomMatrix(int size, int dist, int cnt = -1, int seed = 0);
    bool operator==(Matrix Tmp);
    bool operator!=(const Matrix& Tmp) { return !(*this == Tmp); }
    Matrix T();
    Matrix operator*(Matrix B);
    ~Matrix();
};

#endif  //
   MODULES_TASK_1_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_



// mult_sparse_cc_complex_mat.cpp
// Copyright 2022 Olynin Alexander
#include
    "../../modules/task_1/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

std::vector<Complex> Complex::InitVec(std::vector<double> rls,
                                      std::vector<double> ims) {
    std::vector<Complex> Ent(rls.size());
    if (rls.size() == ims.size()) {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], ims[i]);
        }
    } else {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], 0);
        }
    }
    return Ent;
}
Complex& Complex::operator=(Complex Tmp) {
    this->rl = Tmp.rl;
    this->im = Tmp.im;
    return *this;
}
Complex Complex::operator*(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl * Tmp.rl - this->im * Tmp.im;
    Ans.im = this->rl * Tmp.im + this->im * Tmp.rl;
    return Ans;
}
Complex Complex::operator+(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl + Tmp.rl;
```

```
    Ans.im = this->im + Tmp.im;
    return Ans;
}
Complex& Complex::operator+=(Complex Tmp) {
    this->rl += Tmp.rl;
    this->im += Tmp.im;
    return *this;
}
bool Complex::operator==(Complex Tmp) {
    if ((this->rl - Tmp.rl < 0.00001) && (this->im - Tmp.im < 0.00001)) {
        return true;
    } else {
        return false;
    }
}
bool Complex::IsNotZero() {
    bool ans = false;
    const double ZeroLike = 0.000001;
    if ((fabs(this->rl) > ZeroLike) || (fabs(this->im) > ZeroLike)) {
        ans = true;
    }
    return ans;
}
Matrix& Matrix::operator=(const Matrix& Tmp) {
    this->size = Tmp.size;
    this->non = Tmp.non;
    this->Entry = Tmp.Entry;
    this->irows = Tmp.irows;
    this->shtcols = Tmp.shtcols;

    return *this;
}
Matrix& Matrix::ClearMatrix() {
    this->size = 0;
    this->non = 0;
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();

    return *this;
}
Matrix& Matrix::RandomMatrix(int size, int dist, int cnt, int seed) {
    this->ClearMatrix();
    this->size = size;
    std::mt19937 gen(time(0));
    gen.seed(seed);
    if (cnt < 0) {
        cnt = static_cast<int>(size * 0.01);
    }
    this->non = cnt * size;
    this->Entry.resize(cnt * size);
    this->irows.resize(cnt * size);
    this->shtcols.resize(size + 1);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < cnt; j++) {
            bool flag;
            do {
                this->irows[i * cnt + j] = gen() % size + 1;
```

```
                flag = false;
                for (int k = 0; k < j; k++) {
                    if (this->irows[i * cnt + j] == this->irows[i * cnt + k]) {
                        flag = true;
                    }
                }
            } while (flag);
        }
        for (int j = 0; j < cnt - 1; j++) {
            for (int k = 0; k < cnt - 1; k++) {
                if (this->irows[i * cnt + k] > this->irows[i * cnt + k]) {
                    int tmp = this->irows[i * cnt + k];
                    this->irows[i * cnt + k] = this->irows[i * cnt + k + 1];
                    this->irows[i * cnt + k] = tmp;
                }
            }
        }
    }
    for (int i = 0; i < cnt * size; i++) {
        this->Entry[i].SetRl(gen() % dist + 1);
        this->Entry[i].SetIm(gen() % dist + 1);
    }
    int sum = 1;
    for (int i = 0; i < size + 1; i++) {
        this->shtcols[i] = sum;
        sum += cnt;
    }

    return *this;
}
bool Matrix::operator==(Matrix Tmp) {
    bool ans = true;
    if (this->non != Tmp.non) {
        return false;
    }
    if (this->size != Tmp.size) {
        return false;
    }
    for (int i = 0; i < this->non; i++) {
        if (this->Entry[i] != Tmp.Entry[i]) {
            return false;
        }
    }
    if (this->irows != Tmp.irows) {
        return false;
    }
    if (this->shtcols != Tmp.shtcols) {
        return false;
    }
    return ans;
}
Matrix Matrix::T() {
    Matrix Ans;
    Ans.non = this->non;
    Ans.size = this->size;
    Ans.Entry.resize(this->non);
    Ans.irows.resize(this->non);
    Ans.shtcols.resize(this->size + 1);
```

```
    for (int i = 0; i < this->non; i++) {
        Ans.shtcols[this->irows[i] - 1]++;
    }
    int sum = 1;
    for (int i = 0; i < this->size + 1; i++) {
        int tmp = Ans.shtcols[i];
        Ans.shtcols[i] = sum;
        sum += tmp;
    }
    std::vector<int> shtcols_tmp = Ans.shtcols;
    for (int i = 0; i < this->size; i++) {
        for (int j = this->shtcols[i]; j < this->shtcols[i + 1]; j++) {
            int r_index = this->irows[j - 1];
            int i_index = shtcols_tmp[r_index - 1];
            Ans.irows[i_index - 1] = i + 1;
            Ans.Entry[i_index - 1] = this->Entry[j - 1];
            shtcols_tmp[r_index - 1]++;
        }
    }
    return Ans;
}
Matrix Matrix::operator*(Matrix B) {
    if (this->size != B.size) {
        return Matrix();
    }
    Matrix A = this->T();

    std::vector<Complex> EntRes;
    std::vector<int> irows_res;
    std::vector<int> shtcol_res = { 1 };

    int non_counter = 1;
    for (int j = 0; j < B.size; j++) {
        std::vector<int> ip(A.size, 0);
        for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
            ip[B.irows[i-1]-1] = i;
        }
        for (int i = 0; i < A.size; i++) {
            Complex Sum;
            for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                int p = ip[A.irows[k-1]-1];
                if (p) {
                    Sum += B.Entry[p-1] * A.Entry[k-1];
                }
            }
            if (Sum.IsNotZero()) {
                    irows_res.push_back(i+1);
                    EntRes.push_back(Sum);
                    non_counter++;
            }
        }
        shtcol_res.push_back(non_counter);
    }
    Matrix Ans(A.size, non_counter-1, EntRes, irows_res, shtcol_res);
    return Ans;
}
Matrix::~Matrix() {
```

```
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();
}



// main.cpp
// Copyright 2022 Olynin Alexander
#include <gtest/gtest.h>
#include <time.h>
#include <vector>
#include
    "../../modules/task_1/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

TEST(Class_Complex, Complex_creation) {
    EXPECT_NO_THROW(Complex A(1.2, 2.3));
}

TEST(Class_Complex, Complex_getters) {
    Complex A(1.2, 2.3);

    EXPECT_EQ(A.GetRl(), 1.2);
    EXPECT_EQ(A.GetIm(), 2.3);
}

TEST(Class_Complex, Complex_comparison) {
    Complex A(1.2, 2.3);
    Complex B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Complex, Complex_multiplication) {
    Complex A(1.2, 2.3);
    Complex B(3.1, -3.2);
    Complex Res = A * B;
    Complex Res_exp(11.08, 3.29);

    EXPECT_TRUE(Res == Res_exp);
}

TEST(Class_Matrix, Sparse_matrix_creation) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    EXPECT_NO_THROW(Matrix A(size, non, Ent, irows, shtcol));
}

TEST(Class_Matrix, Sparse_matrix_comparison) {
    int non = 6;
    int size = 4;
```

```
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };
    Matrix A(size, non, Ent, irows, shtcol);
    Matrix B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Matrix, Sparse_matrix_transposition) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */
    A = A.T();
    /*
    * Matrix A.T should be
    *
    * 0  0  0   9
    * 3  0  0   0
    * 0  8  0  15
    * 7  0  0  16
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({3, 7, 8, 9, 15, 16});
    std::vector<int> irows_res = { 2, 4, 3, 1, 3, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 4, 7 };
    Matrix Res(size, non, EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(A == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_onlyreal) {
    int non = 6;
    int size = 4;
    Complex Tmp;
```

```
    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */

    Ent = Tmp.InitVec({ 2, 3, 1, 8, 4, 5 });
    irows = { 1, 3, 1, 2, 3, 4 };
    shtcol = { 1, 3, 4, 6, 7 };

    Matrix B(size, non, Ent, irows, shtcol);

    /*
    * Matrix B is
    *
    * 2  1  0  0
    * 0  0  8  0
    * 3  0  4  0
    * 0  0  0  5
    *
    */

    Matrix C = A * B;

    /*
    * Matrix C should be
    *
    * 0   0  24  35
    * 24  0  32   0
    * 0   0   0   0
    * 63  9  60  80
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ 24, 63, 9, 24, 32, 60, 35, 80 });
    std::vector<int> irows_res = { 2, 4, 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 7, 9 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex) {
    int non = 5;
    int size = 4;
    Complex Tmp;
```

```
    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 4, 8, 18, 15, 2 }, { 20, 4, 4, 11, 7 });
    std::vector<int> irows = { 1, 4, 2, 3, 1 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 6 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 4+20i  0    0     2+7i
    * 0      0    18+4i   0
    * 0      0    15+11i  0
    * 0   8+4i    0       0
    *
    */

    Ent = Tmp.InitVec({ 4, 1, 12, 20, 9 }, { 14, 3, 2, 13, 10 });
    irows = { 2, 2, 4, 1, 1 };
    shtcol = { 1, 2, 4, 5, 6 };

    Matrix B(size, non, Ent, irows, shtcol);

    /*
    * Matrix B is
    *
    * 0      0      20+13i  9+10i
    * 4+14i  1+3i   0           0
    * 0      0      0           0
    * 0      12+2i  0           0
    *
    */

    Matrix C = A * B;

    /*
    * Matrix C should be
    *
    * 0          10+88i   -180+452i  -164+220i
    * 0          0        0                  0
    * 0          0        0                  0
    * -24+128i   -4+28i   0                  0
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ -24, 10, -4, -180, -164 },
                         { 128, 88, 28, 452, 220 });
    std::vector<int> irows_res = { 4, 1, 4, 1, 1 };
    std::vector<int> shtcol_res = { 1, 2, 4, 5, 6 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_random_matrix_creation) {
    Matrix A;
```

```
    int size = 10;
    int dist = 100;
    int cnt = 1;
    int seed = 0;

    EXPECT_NO_THROW(A.RandomMatrix(size, dist, cnt, seed));
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex_small) {
    int size = 10;
    int dist = 100;
    int cnt = 1;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);

    EXPECT_NO_THROW(A * B);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex_meduim) {
    int size = 100;
    int dist = 1000;
    int cnt = 1;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);

    EXPECT_NO_THROW(A * B);
}
```

Параллельная OpenMP версия

```
// mult_sparse_cc_complex_mat.h
// Copyright 2022 Olynin Alexander
#ifndef
    MODULES_TASK_2_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_
#define
    MODULES_TASK_2_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_

#include <time.h>
#include <random>
#include <vector>

class Complex {
 private:
    double rl;
    double im;
 public:
    explicit Complex(double _rl = 0, double _im = 0): rl(_rl), im(_im) {}
    Complex(const Complex& Tmp): rl(Tmp.rl), im(Tmp.im) {}
    Complex& operator=(Complex Tmp);
    std::vector<Complex> InitVec(std::vector<double> rls =
                                 std::vector<double>(),
                                 std::vector<double> ims =
```

```
                                    std::vector<double>());
    double GetRl() { return this->rl; }
    double GetIm() { return this->im; }
    void SetRl(double tmp) { this->rl = tmp; }
    void SetIm(double tmp) { this->im = tmp; }
    Complex operator+(Complex Tmp);
    Complex& operator+=(Complex Tmp);
    Complex operator*(Complex Tmp);
    bool operator==(Complex Tmp);
    bool operator!=(Complex Tmp) { return !(*this == Tmp); }
    bool IsNotZero();
    ~Complex() {}
};

class Matrix {
 private:
    int size;
    int non;
    std::vector<Complex> Entry;
    std::vector<int> irows;
    std::vector<int> shtcols;

 public:
    Matrix(int _size = 0, int _non = 0, const std::vector<Complex>& _Entry =
           std::vector<Complex>(), const std::vector<int>& _irows =
           std::vector<int>(), const std::vector<int>& _shtcols =
           std::vector<int>()): size(_size), non(_non), Entry(_Entry),
           irows(_irows), shtcols(_shtcols) {}
    Matrix(const Matrix& Tmp): size(Tmp.size), non(Tmp.non), Entry(Tmp.Entry),
                               irows(Tmp.irows), shtcols(Tmp.shtcols) {}
    Matrix& operator=(const Matrix& Tmp);
    int GetSize() { return this->size; }
    int GetNon() { return this-> non; }
    std::vector<Complex> GetEntry() { return this->Entry; }
    std::vector<int> GetIrows() { return this->irows; }
    std::vector<int> GetShtcols() { return this->shtcols; }
    Matrix& ClearMatrix();
    Matrix& RandomMatrix(int size, int dist, int cnt = -1, int seed = 0);
    bool operator==(Matrix Tmp);
    bool operator!=(const Matrix& Tmp) { return !(*this == Tmp); }
    Matrix T();
    Matrix operator^(Matrix B);
    Matrix operator*(Matrix B);
    ~Matrix();
};

#endif  //
   MODULES_TASK_2_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_



// mult_sparse_cc_complex_mat.cpp
// Copyright 2022 Olynin Alexander
#include <omp.h>
#include
    "../../modules/task_2/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

std::vector<Complex> Complex::InitVec(std::vector<double> rls,
```

```
                                        std::vector<double> ims) {
    std::vector<Complex> Ent(rls.size());
    if (rls.size() == ims.size()) {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], ims[i]);
        }
    } else {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], 0);
        }
    }
    return Ent;
}
Complex& Complex::operator=(Complex Tmp) {
    this->rl = Tmp.rl;
    this->im = Tmp.im;
    return *this;
}
Complex Complex::operator*(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl * Tmp.rl - this->im * Tmp.im;
    Ans.im = this->rl * Tmp.im + this->im * Tmp.rl;
    return Ans;
}
Complex Complex::operator+(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl + Tmp.rl;
    Ans.im = this->im + Tmp.im;
    return Ans;
}
Complex& Complex::operator+=(Complex Tmp) {
    this->rl += Tmp.rl;
    this->im += Tmp.im;
    return *this;
}
bool Complex::operator==(Complex Tmp) {
    if ((this->rl - Tmp.rl < 0.00001) && (this->im - Tmp.im < 0.00001)) {
        return true;
    } else {
        return false;
    }
}
bool Complex::IsNotZero() {
    bool ans = false;
    const double ZeroLike = 0.000001;
    if ((fabs(this->rl) > ZeroLike) || (fabs(this->im) > ZeroLike)) {
        ans = true;
    }
    return ans;
}
Matrix& Matrix::operator=(const Matrix& Tmp) {
    this->size = Tmp.size;
    this->non = Tmp.non;
    this->Entry = Tmp.Entry;
    this->irows = Tmp.irows;
    this->shtcols = Tmp.shtcols;

    return *this;
```

```
}
Matrix& Matrix::ClearMatrix() {
    this->size = 0;
    this->non = 0;
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();

    return *this;
}
Matrix& Matrix::RandomMatrix(int size, int dist, int cnt, int seed) {
    this->ClearMatrix();
    this->size = size;
    std::mt19937 gen(time(0));
    gen.seed(seed);
    if (cnt < 0) {
        cnt = static_cast<int>(size * 0.01);
    }
    this->non = cnt * size;
    this->Entry.resize(cnt * size);
    this->irows.resize(cnt * size);
    this->shtcols.resize(size + 1);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < cnt; j++) {
            bool flag;
            do {
                this->irows[i * cnt + j] = gen() % size + 1;
                flag = false;
                for (int k = 0; k < j; k++) {
                    if (this->irows[i * cnt + j] == this->irows[i * cnt + k]) {
                        flag = true;
                    }
                }
            } while (flag);
        }
        for (int j = 0; j < cnt - 1; j++) {
            for (int k = 0; k < cnt - 1; k++) {
                if (this->irows[i * cnt + k] > this->irows[i * cnt + k + 1]) {
                    int tmp = this->irows[i * cnt + k];
                    this->irows[i * cnt + k] = this->irows[i * cnt + k + 1];
                    this->irows[i * cnt + k + 1] = tmp;
                }
            }
        }
    }
    for (int i = 0; i < cnt * size; i++) {
        this->Entry[i].SetRl(gen() % dist + 1);
        this->Entry[i].SetIm(gen() % dist + 1);
    }
    int sum = 1;
    for (int i = 0; i < size + 1; i++) {
        this->shtcols[i] = sum;
        sum += cnt;
    }

    return *this;
}
bool Matrix::operator==(Matrix Tmp) {
```

```
    bool ans = true;
    if (this->non != Tmp.non) {
        return false;
    }
    if (this->size != Tmp.size) {
        return false;
    }
    for (int i = 0; i < this->non; i++) {
        if (this->Entry[i] != Tmp.Entry[i]) {
            return false;
        }
    }
    if (this->irows != Tmp.irows) {
        return false;
    }
    if (this->shtcols != Tmp.shtcols) {
        return false;
    }
    return ans;
}
Matrix Matrix::T() {
    Matrix Ans;
    Ans.non = this->non;
    Ans.size = this->size;
    Ans.Entry.resize(this->non);
    Ans.irows.resize(this->non);
    Ans.shtcols.resize(this->size + 1);

    for (int i = 0; i < this->non; i++) {
        Ans.shtcols[this->irows[i] - 1]++;
    }
    int sum = 1;
    for (int i = 0; i < this->size + 1; i++) {
        int tmp = Ans.shtcols[i];
        Ans.shtcols[i] = sum;
        sum += tmp;
    }
    std::vector<int> shtcols_tmp = Ans.shtcols;
    for (int i = 0; i < this->size; i++) {
        for (int j = this->shtcols[i]; j < this->shtcols[i + 1]; j++) {
            int r_index = this->irows[j - 1];
            int i_index = shtcols_tmp[r_index - 1];
            Ans.irows[i_index - 1] = i + 1;
            Ans.Entry[i_index - 1] = this->Entry[j - 1];
            shtcols_tmp[r_index - 1]++;
        }
    }
    return Ans;
}
Matrix Matrix::operator^(Matrix B) {
    if (this->size != B.size) {
        return Matrix();
    }
    Matrix A = this->T();

    std::vector<Complex> EntRes;
    std::vector<int> irows_res;
    std::vector<int> shtcol_res = { 1 };
```

```
    int non_counter = 1;
    for (int j = 0; j < B.size; j++) {
        std::vector<int> ip(A.size, 0);
        for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
            ip[B.irows[i-1]-1] = i;
        }
        for (int i = 0; i < A.size; i++) {
            Complex Sum;
            for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                int irow = A.irows[k-1];
                int p = ip[irow-1];
                if (p) {
                    Sum += B.Entry[p-1] * A.Entry[k-1];
                }
            }
            if (Sum.IsNotZero()) {
                    irows_res.push_back(i+1);
                    EntRes.push_back(Sum);
                    non_counter++;
            }
        }
        shtcol_res.push_back(non_counter);
    }
    Matrix Ans(A.size, non_counter-1, EntRes, irows_res, shtcol_res);
    return Ans;
}
Matrix Matrix::operator*(Matrix B) {
    int num_threads = 8;

    if (this->size != B.size) {
        return Matrix();
    }
    Matrix A = this->T();

    if (A.size < num_threads) {
        num_threads = A.size;
    }
    std::vector<std::vector<Complex>> Entry_shared(num_threads);
    std::vector<std::vector<int>> irows_shared(num_threads);
    std::vector<int> counter(A.size);
    #pragma omp parallel num_threads(num_threads)
    {
        std::vector<Complex> Entry_private;
        std::vector<int> irows_private;
        int ind = omp_get_thread_num();
        #pragma omp for
        for (int j = 0; j < B.size; j++) {
            std::vector<int> ip(A.size, 0);
            int non_counter = 0;
            for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
                int irow = B.irows[i-1];
                ip[irow-1] = i;
            }
            for (int i = 0; i < A.size; i++) {
                Complex Sum;
                for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                    int irow = A.irows[k-1];
```

```
                    int p = ip[irow-1];
                    if (p) {
                        Sum += B.Entry[p-1] * A.Entry[k-1];
                    }
                }
                if (Sum.IsNotZero()) {
                    Entry_private.push_back(Sum);
                    irows_private.push_back(i+1);
                    non_counter++;
                }
            }
            counter[j] += non_counter;
        }
        Entry_shared[ind] = Entry_private;
        irows_shared[ind] = irows_private;
    }
    std::vector<Complex> EntryRes;
    std::vector<int> irowsres;
    for (int i = 0; i < num_threads; i++) {
        EntryRes.insert(EntryRes.end(),
                        Entry_shared[i].begin(), Entry_shared[i].end());
        irowsres.insert(irowsres.end(),
                        irows_shared[i].begin(), irows_shared[i].end());
    }
    std::vector<int> shtcolsres = { 1 };
    int sum = 1;
    for (int i = 0; i < A.size; i++) {
        sum += counter[i];
        shtcolsres.push_back(sum);
    }
    Matrix Ans(A.size, EntryRes.size(), EntryRes, irowsres, shtcolsres);
    return Ans;
}
Matrix::~Matrix() {
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();
}



// main.cpp
// Copyright 2022 Olynin Alexander
#include <gtest/gtest.h>
#include <time.h>
#include <omp.h>
#include <vector>
#include
    "../../modules/task_2/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

TEST(Class_Complex, Complex_creation) {
    EXPECT_NO_THROW(Complex A(1.2, 2.3));
}

TEST(Class_Complex, Complex_getters) {
    Complex A(1.2, 2.3);

    EXPECT_EQ(A.GetRl(), 1.2);
```

```
    EXPECT_EQ(A.GetIm(), 2.3);
}

TEST(Class_Complex, Complex_comparison) {
    Complex A(1.2, 2.3);
    Complex B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Complex, Complex_multiplication) {
    Complex A(1.2, 2.3);
    Complex B(3.1, -3.2);
    Complex Res = A * B;
    Complex Res_exp(11.08, 3.29);

    EXPECT_TRUE(Res == Res_exp);
}

TEST(Class_Matrix, Sparse_matrix_creation) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    EXPECT_NO_THROW(Matrix A(size, non, Ent, irows, shtcol));
}

TEST(Class_Matrix, Sparse_matrix_comparison) {
    int non = 6;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };
    Matrix A(size, non, Ent, irows, shtcol);
    Matrix B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Matrix, Sparse_matrix_transposition) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);
```

```
    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */
    A = A.T();
    /*
    * Matrix A.T should be
    *
    * 0  0  0   9
    * 3  0  0   0
    * 0  8  0  15
    * 7  0  0  16
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({3, 7, 8, 9, 15, 16});
    std::vector<int> irows_res = { 2, 4, 3, 1, 3, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 4, 7 };
    Matrix Res(size, non, EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(A == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_onlyreal) {
    int non = 6;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */

    Ent = Tmp.InitVec({ 2, 3, 1, 8, 4, 5 });
    irows = { 1, 3, 1, 2, 3, 4 };
    shtcol = { 1, 3, 4, 6, 7 };

    Matrix B(size, non, Ent, irows, shtcol);
```

```
    /*
    * Matrix B is
    *
    * 2  1  0  0
    * 0  0  8  0
    * 3  0  4  0
    * 0  0  0  5
    *
    */

    Matrix C = A * B;

    /*
    * Matrix C should be
    *
    * 0   0  24  35
    * 24  0  32   0
    * 0   0   0   0
    * 63  9  60  80
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ 24, 63, 9, 24, 32, 60, 35, 80 });
    std::vector<int> irows_res = { 2, 4, 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 7, 9 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex) {
    int non = 5;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 4, 8, 18, 15, 2 }, { 20, 4, 4, 11, 7 });
    std::vector<int> irows = { 1, 4, 2, 3, 1 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 6 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 4+20i  0    0     2+7i
    * 0      0    18+4i   0
    * 0      0    15+11i  0
    * 0   8+4i    0       0
    *
    */

    Ent = Tmp.InitVec({ 4, 1, 12, 20, 9 }, { 14, 3, 2, 13, 10 });
    irows = { 2, 2, 4, 1, 1 };
    shtcol = { 1, 2, 4, 5, 6 };

    Matrix B(size, non, Ent, irows, shtcol);
```

```
    /*
    * Matrix B is
    *
    * 0      0      20+13i  9+10i
    * 4+14i  1+3i   0           0
    * 0      0      0           0
    * 0      12+2i  0           0
    *
    */

    Matrix C = A * B;

    /*
    * Matrix C should be
    *
    * 0          10+88i   -180+452i   -164+220i
    * 0          0        0                  0
    * 0          0        0                  0
    * -24+128i   -4+28i   0                  0
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ -24, 10, -4, -180, -164 },
                         { 128, 88, 28, 452, 220 });
    std::vector<int> irows_res = { 4, 1, 4, 1, 1 };
    std::vector<int> shtcol_res = { 1, 2, 4, 5, 6 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_random_matrix_creation) {
    Matrix A;
    int size = 10;
    int dist = 100;
    int cnt = 1;
    int seed = 0;

    EXPECT_NO_THROW(A.RandomMatrix(size, dist, cnt, seed));
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex_small) {
    int size = 10;
    int dist = 1000;
    int cnt = 1;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 4);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);

    Matrix C_seq = A ^ B;
    Matrix C_par = A * B;
    EXPECT_TRUE(C_seq == C_par);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_meduim_time_perfomance) {
```

```
    double start, end;
    double seq_time, omp_time;
    int size = 500;
    int dist = 1000;
    int cnt = 5;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = omp_get_wtime();
    Matrix C_seq = A ^ B;
    end = omp_get_wtime();
    std::cout << "SEQ Spended time -> " << end - start + .0
              << " <- seconds" << std::endl;
    seq_time = end - start + .0;

    start = omp_get_wtime();
    Matrix C_omp = A * B;
    end = omp_get_wtime();
    std::cout << "OMP Spended time -> " << end - start + .0
              << " <- seconds" << std::endl;
    omp_time = end - start + .0;
    std::cout << "Performance improvement by -| " << seq_time / omp_time
              << " |- times" << std::endl;
    EXPECT_TRUE(C_seq == C_omp);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_large_time_perfomance) {
    double start, end;
    double seq_time, omp_time;
    int size = 1000;
    int dist = 1000;
    int cnt = 10;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = omp_get_wtime();
    Matrix C_seq = A ^ B;
    end = omp_get_wtime();
    std::cout << "SEQ Spended time -> " << end - start + .0
              << " <- seconds" << std::endl;
    seq_time = end - start + .0;

    start = omp_get_wtime();
    Matrix C_omp = A * B;
    end = omp_get_wtime();
    std::cout << "OMP Spended time -> " << end - start + .0
              << " <- seconds" << std::endl;
    omp_time = end - start + .0;
    std::cout << "Performance improvement by -| " << seq_time / omp_time
              << " |- times" << std::endl;
    EXPECT_TRUE(C_seq == C_omp);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_ext_large_time_perfomance) {
    double start, end;
    double seq_time, omp_time;
```

```
    int size = 2500;
    int dist = 100000;
    int cnt = 25;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = omp_get_wtime();
    Matrix C_seq = A ^ B;
    end = omp_get_wtime();
    std::cout << "SEQ Spended time -> " << end - start + .0
              << " <- seconds" << std::endl;
    seq_time = end - start + .0;

    start = omp_get_wtime();
    Matrix C_omp = A * B;
    end = omp_get_wtime();
    std::cout << "OMP Spended time -> " << end - start + .0
              << " <- seconds" << std::endl;
    omp_time = end - start + .0;
    std::cout << "Performance improvement by - | " << seq_time / omp_time
              << " | - times" << std::endl;
    EXPECT_TRUE(C_seq == C_omp);
}
```

Параллельная TBB версия

```
// mult_sparse_cc_complex_mat.h
// Copyright 2022 Olynin Alexander
#ifndef
    MODULES_TASK_3_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_
#define
    MODULES_TASK_3_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_

#include <time.h>
#include <random>
#include <vector>

class Complex {
 private:
    double rl;
    double im;
 public:
    explicit Complex(double _rl = 0, double _im = 0): rl(_rl), im(_im) {}
    Complex(const Complex& Tmp): rl(Tmp.rl), im(Tmp.im) {}
    Complex& operator=(Complex Tmp);
    std::vector<Complex> InitVec(std::vector<double> rls =
                                 std::vector<double>(),
                                 std::vector<double> ims =
                                 std::vector<double>());
    double GetRl() { return this->rl; }
    double GetIm() { return this->im; }
    void SetRl(double tmp) { this->rl = tmp; }
    void SetIm(double tmp) { this->im = tmp; }
    Complex operator+(Complex Tmp);
    Complex& operator+=(Complex Tmp);
```

```
    Complex operator*(Complex Tmp);
    bool operator==(Complex Tmp);
    bool operator!=(Complex Tmp) { return !(*this == Tmp); }
    bool IsNotZero();
    ~Complex() {}
};

class Matrix {
 private:
    int size;
    int non;
    std::vector<Complex> Entry;
    std::vector<int> irows;
    std::vector<int> shtcols;

 public:
    Matrix(int _size = 0, int _non = 0, const std::vector<Complex>& _Entry =
           std::vector<Complex>(), const std::vector<int>& _irows =
           std::vector<int>(), const std::vector<int>& _shtcols =
           std::vector<int>()): size(_size), non(_non), Entry(_Entry),
           irows(_irows), shtcols(_shtcols) {}
    Matrix(const Matrix& Tmp): size(Tmp.size), non(Tmp.non), Entry(Tmp.Entry),
                               irows(Tmp.irows), shtcols(Tmp.shtcols) {}
    Matrix& operator=(const Matrix& Tmp);
    int GetSize() { return this->size; }
    int GetNon() { return this-> non; }
    std::vector<Complex> GetEntry() { return this->Entry; }
    std::vector<int> GetIrows() { return this->irows; }
    std::vector<int> GetShtcols() { return this->shtcols; }
    Matrix& ClearMatrix();
    Matrix& RandomMatrix(int size, int dist, int cnt = -1, int seed = 0);
    bool operator==(Matrix Tmp);
    bool operator!=(const Matrix& Tmp) { return !(*this == Tmp); }
    Matrix T();
    Matrix operator^(Matrix B);
    Matrix operator*(Matrix B);
    ~Matrix();
};

#endif  //
    MODULES_TASK_3_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_



// mult_sparse_cc_complex_mat.cpp
// Copyright 2022 Olynin Alexander
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"
#include
    "../../modules/task_3/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

std::vector<Complex> Complex::InitVec(std::vector<double> rls,
                                      std::vector<double> ims) {
    std::vector<Complex> Ent(rls.size());
    if (rls.size() == ims.size()) {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], ims[i]);
        }
```

```
    } else {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], 0);
        }
    }
    return Ent;
}
Complex& Complex::operator=(Complex Tmp) {
    this->rl = Tmp.rl;
    this->im = Tmp.im;
    return *this;
}
Complex Complex::operator*(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl * Tmp.rl - this->im * Tmp.im;
    Ans.im = this->rl * Tmp.im + this->im * Tmp.rl;
    return Ans;
}
Complex Complex::operator+(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl + Tmp.rl;
    Ans.im = this->im + Tmp.im;
    return Ans;
}
Complex& Complex::operator+=(Complex Tmp) {
    this->rl += Tmp.rl;
    this->im += Tmp.im;
    return *this;
}
bool Complex::operator==(Complex Tmp) {
    if ((this->rl - Tmp.rl < 0.00001) && (this->im - Tmp.im < 0.00001)) {
        return true;
    } else {
        return false;
    }
}
bool Complex::IsNotZero() {
    bool ans = false;
    const double ZeroLike = 0.000001;
    if ((fabs(this->rl) > ZeroLike) || (fabs(this->im) > ZeroLike)) {
        ans = true;
    }
    return ans;
}
Matrix& Matrix::operator=(const Matrix& Tmp) {
    this->size = Tmp.size;
    this->non = Tmp.non;
    this->Entry = Tmp.Entry;
    this->irows = Tmp.irows;
    this->shtcols = Tmp.shtcols;

    return *this;
}
Matrix& Matrix::ClearMatrix() {
    this->size = 0;
    this->non = 0;
    this->Entry.clear();
    this->irows.clear();
```

```
    this->shtcols.clear();

    return *this;
}
Matrix& Matrix::RandomMatrix(int size, int dist, int cnt, int seed) {
    this->ClearMatrix();
    this->size = size;
    std::mt19937 gen(time(0));
    gen.seed(seed);
    if (cnt < 0) {
        cnt = static_cast<int>(size * 0.01);
    }
    this->non = cnt * size;
    this->Entry.resize(cnt * size);
    this->irows.resize(cnt * size);
    this->shtcols.resize(size + 1);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < cnt; j++) {
            bool flag;
            do {
                this->irows[i * cnt + j] = gen() % size + 1;
                flag = false;
                for (int k = 0; k < j; k++) {
                    if (this->irows[i * cnt + j] == this->irows[i * cnt + k]) {
                        flag = true;
                    }
                }
            } while (flag);
        }
        for (int j = 0; j < cnt - 1; j++) {
            for (int k = 0; k < cnt - 1; k++) {
                if (this->irows[i * cnt + k] > this->irows[i * cnt + k + 1]) {
                    int tmp = this->irows[i * cnt + k];
                    this->irows[i * cnt + k] = this->irows[i * cnt + k + 1];
                    this->irows[i * cnt + k + 1] = tmp;
                }
            }
        }
    }
    for (int i = 0; i < cnt * size; i++) {
        this->Entry[i].SetRl(gen() % dist + 1);
        this->Entry[i].SetIm(gen() % dist + 1);
    }
    int sum = 1;
    for (int i = 0; i < size + 1; i++) {
        this->shtcols[i] = sum;
        sum += cnt;
    }

    return *this;
}
bool Matrix::operator==(Matrix Tmp) {
    bool ans = true;
    if (this->non != Tmp.non) {
        return false;
    }
    if (this->size != Tmp.size) {
        return false;
```

```
    }
    for (int i = 0; i < this->non; i++) {
        if (this->Entry[i] != Tmp.Entry[i]) {
            return false;
        }
    }
    if (this->irows != Tmp.irows) {
        return false;
    }
    if (this->shtcols != Tmp.shtcols) {
        return false;
    }
    return ans;
}
Matrix Matrix::T() {
    Matrix Ans;
    Ans.non = this->non;
    Ans.size = this->size;
    Ans.Entry.resize(this->non);
    Ans.irows.resize(this->non);
    Ans.shtcols.resize(this->size + 1);

    for (int i = 0; i < this->non; i++) {
        Ans.shtcols[this->irows[i] - 1]++;
    }
    int sum = 1;
    for (int i = 0; i < this->size + 1; i++) {
        int tmp = Ans.shtcols[i];
        Ans.shtcols[i] = sum;
        sum += tmp;
    }
    std::vector<int> shtcols_tmp = Ans.shtcols;
    for (int i = 0; i < this->size; i++) {
        for (int j = this->shtcols[i]; j < this->shtcols[i + 1]; j++) {
            int r_index = this->irows[j - 1];
            int i_index = shtcols_tmp[r_index - 1];
            Ans.irows[i_index - 1] = i + 1;
            Ans.Entry[i_index - 1] = this->Entry[j - 1];
            shtcols_tmp[r_index - 1]++;
        }
    }
    return Ans;
}
Matrix Matrix::operator^(Matrix B) {
    if (this->size != B.size) {
        return Matrix();
    }
    Matrix A = this->T();

    std::vector<Complex> EntRes;
    std::vector<int> irows_res;
    std::vector<int> shtcol_res = { 1 };

    int non_counter = 1;
    for (int j = 0; j < B.size; j++) {
        std::vector<int> ip(A.size, 0);
        for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
            ip[B.irows[i-1]-1] = i;
```

```
        }
        for (int i = 0; i < A.size; i++) {
            Complex Sum;
            for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                int irow = A.irows[k-1];
                int p = ip[irow-1];
                if (p) {
                    Sum += B.Entry[p-1] * A.Entry[k-1];
                }
            }
            if (Sum.IsNotZero()) {
                    irows_res.push_back(i+1);
                    EntRes.push_back(Sum);
                    non_counter++;
            }
        }
        shtcol_res.push_back(non_counter);
    }
    Matrix Ans(A.size, non_counter-1, EntRes, irows_res, shtcol_res);
    return Ans;
}
Matrix Matrix::operator*(Matrix B) {
    if (this->size != B.size) {
        return Matrix();
    }
    Matrix A = this->T();

    std::vector<std::vector<Complex>> Entry_col(A.size);
    std::vector<std::vector<int>> irows_col(A.size);
    std::vector<int> counter(A.size);
    int grainsize = B.size / 5;
    if (grainsize == 0) {
        grainsize = 1;
    }
    parallel_for(tbb::blocked_range<int>(0, B.size, grainsize),
                    [&](const tbb::blocked_range<int>& range) {
        for (int j = range.begin(); j < range.end(); j++) {
            std::vector<int> ip(A.size, 0);
            int non_counter = 0;
            std::vector<Complex> Entry_thread;
            std::vector<int> irows_thread;
            for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
                ip[B.irows[i-1]-1] = i;
            }
            for (int i = 0; i < A.size; i++) {
                Complex Sum;
                for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                    int irow = A.irows[k-1];
                    int p = ip[irow-1];
                    if (p) {
                        Sum += B.Entry[p-1] * A.Entry[k-1];
                    }
                }
                if (Sum.IsNotZero()) {
                    Entry_thread.push_back(Sum);
                    irows_thread.push_back(i+1);
                    non_counter++;
                }
```

```
            }
            counter[j] += non_counter;
            Entry_col[j] = Entry_thread;
            irows_col[j] = irows_thread;
        }
    });
    std::vector<Complex> EntryRes;
    std::vector<int> irowsres;
    std::vector<int> shtcolsres = { 1 };
    int sum = 1;
    for (int i = 0; i < A.size; i++) {
        sum += counter[i];
        shtcolsres.push_back(sum);
        EntryRes.insert(EntryRes.end(),
                        Entry_col[i].begin(),
                        Entry_col[i].end());
        irowsres.insert(irowsres.end(),
                        irows_col[i].begin(),
                        irows_col[i].end());
    }
    Matrix Ans(A.size, EntryRes.size(), EntryRes, irowsres, shtcolsres);
    return Ans;
}
Matrix::~Matrix() {
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();
}



// main.cpp
// Copyright 2022 Olynin Alexander
#include <gtest/gtest.h>
#include <time.h>
#include <vector>
#include
    "../../modules/task_3/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

TEST(Class_Complex, Complex_creation) {
    EXPECT_NO_THROW(Complex A(1.2, 2.3));
}

TEST(Class_Complex, Complex_getters) {
    Complex A(1.2, 2.3);

    EXPECT_EQ(A.GetRl(), 1.2);
    EXPECT_EQ(A.GetIm(), 2.3);
}

TEST(Class_Complex, Complex_comparison) {
    Complex A(1.2, 2.3);
    Complex B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Complex, Complex_multiplication) {
```

```
    Complex A(1.2, 2.3);
    Complex B(3.1, -3.2);
    Complex Res = A * B;
    Complex Res_exp(11.08, 3.29);

    EXPECT_TRUE(Res == Res_exp);
}

TEST(Class_Matrix, Sparse_matrix_creation) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    EXPECT_NO_THROW(Matrix A(size, non, Ent, irows, shtcol));
}

TEST(Class_Matrix, Sparse_matrix_comparison) {
    int non = 6;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };
    Matrix A(size, non, Ent, irows, shtcol);
    Matrix B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Matrix, Sparse_matrix_transposition) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */
    A = A.T();
```

```
    /*
    * Matrix A.T should be
    *
    * 0  0  0   9
    * 3  0  0   0
    * 0  8  0  15
    * 7  0  0  16
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({3, 7, 8, 9, 15, 16});
    std::vector<int> irows_res = { 2, 4, 3, 1, 3, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 4, 7 };
    Matrix Res(size, non, EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(A == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_onlyreal) {
    int non = 6;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */

    Ent = Tmp.InitVec({ 2, 3, 1, 8, 4, 5 });
    irows = { 1, 3, 1, 2, 3, 4 };
    shtcol = { 1, 3, 4, 6, 7 };

    Matrix B(size, non, Ent, irows, shtcol);

    /*
    * Matrix B is
    *
    * 2  1  0  0
    * 0  0  8  0
    * 3  0  4  0
    * 0  0  0  5
    *
    */

    Matrix C = A * B;
```

```
    /*
    * Matrix C should be
    *
    * 0   0  24  35
    * 24  0  32   0
    * 0   0   0   0
    * 63  9  60  80
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ 24, 63, 9, 24, 32, 60, 35, 80 });
    std::vector<int> irows_res = { 2, 4, 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 7, 9 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex) {
    int non = 5;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 4, 8, 18, 15, 2 }, { 20, 4, 4, 11, 7 });
    std::vector<int> irows = { 1, 4, 2, 3, 1 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 6 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 4+20i  0   0    2+7i
    * 0      0   18+4i  0
    * 0      0   15+11i 0
    * 0   8+4i   0      0
    *
    */

    Ent = Tmp.InitVec({ 4, 1, 12, 20, 9 }, { 14, 3, 2, 13, 10 });
    irows = { 2, 2, 4, 1, 1 };
    shtcol = { 1, 2, 4, 5, 6 };

    Matrix B(size, non, Ent, irows, shtcol);

    /*
    * Matrix B is
    *
    * 0      0      20+13i  9+10i
    * 4+14i  1+3i   0          0
    * 0      0      0          0
    * 0      12+2i  0          0
    *
    */
```

```
    Matrix C = A * B;

    /*
    * Matrix C should be
    *
    * 0          10+88i   -180+452i   -164+220i
    * 0          0        0                  0
    * 0          0        0                  0
    * -24+128i   -4+28i   0                  0
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ -24, 10, -4, -180, -164 },
                         { 128, 88, 28, 452, 220 });
    std::vector<int> irows_res = { 4, 1, 4, 1, 1 };
    std::vector<int> shtcol_res = { 1, 2, 4, 5, 6 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_random_matrix_creation) {
    Matrix A;
    int size = 10;
    int dist = 100;
    int cnt = 1;
    int seed = 0;

    EXPECT_NO_THROW(A.RandomMatrix(size, dist, cnt, seed));
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex_small) {
    int size = 10;
    int dist = 1000;
    int cnt = 1;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 4);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);

    Matrix C_seq = A ^ B;
    Matrix C_par = A * B;
    EXPECT_TRUE(C_seq == C_par);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_meduim_time_perfomance) {
    clock_t start, end;
    double seq_time, tbb_time;
    int size = 500;
    int dist = 1000;
    int cnt = 5;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = clock();
    Matrix C_seq = A ^ B;
```

```
    end = clock();
    std::cout << "SEQ Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    seq_time = (end - start + .0) / CLOCKS_PER_SEC;

    start = clock();
    Matrix C_tbb = A * B;
    end = clock();
    std::cout << "TBB Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    tbb_time = (end - start + .0) / CLOCKS_PER_SEC;
    std::cout << "Performance improvement by -| " << seq_time / tbb_time
            << " |- times" << std::endl;
    EXPECT_TRUE(C_seq == C_tbb);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_large_time_perfomance) {
    clock_t start, end;
    double seq_time, tbb_time;
    int size = 1000;
    int dist = 1000;
    int cnt = 10;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = clock();
    Matrix C_seq = A ^ B;
    end = clock();
    std::cout << "SEQ Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    seq_time = (end - start + .0) / CLOCKS_PER_SEC;

    start = clock();
    Matrix C_tbb = A * B;
    end = clock();
    std::cout << "TBB Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    tbb_time = (end - start + .0) / CLOCKS_PER_SEC;
    std::cout << "Performance improvement by -| " << seq_time / tbb_time
            << " |- times" << std::endl;
    EXPECT_TRUE(C_seq == C_tbb);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_ext_large_time_perfomance) {
    clock_t start, end;
    double seq_time, tbb_time;
    int size = 2500;
    int dist = 100000;
    int cnt = 25;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = clock();
    Matrix C_seq = A ^ B;
    end = clock();
    std::cout << "SEQ Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
```

```
            << " <- seconds" << std::endl;
    seq_time = (end - start + .0) / CLOCKS_PER_SEC;

    start = clock();
    Matrix C_tbb = A * B;
    end = clock();
    std::cout << "TBB Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    tbb_time = (end - start + .0) / CLOCKS_PER_SEC;
    std::cout << "Performance improvement by -| " << seq_time / tbb_time
            << " |- times" << std::endl;
    EXPECT_TRUE(C_seq == C_tbb);
}
```

Параллельная std::thread версия

```
// mult_sparse_cc_complex_mat.h
// Copyright 2022 Olynin Alexander
#ifndef
    MODULES_TASK_4_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_
#define
    MODULES_TASK_4_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_

#include <time.h>
#include <random>
#include <vector>

class Complex {
 private:
    double rl;
    double im;
 public:
    explicit Complex(double _rl = 0, double _im = 0): rl(_rl), im(_im) {}
    Complex(const Complex& Tmp): rl(Tmp.rl), im(Tmp.im) {}
    Complex& operator=(Complex Tmp);
    std::vector<Complex> InitVec(std::vector<double> rls =
                                 std::vector<double>(),
                                 std::vector<double> ims =
                                 std::vector<double>());
    double GetRl() { return this->rl; }
    double GetIm() { return this->im; }
    void SetRl(double tmp) { this->rl = tmp; }
    void SetIm(double tmp) { this->im = tmp; }
    Complex operator+(Complex Tmp);
    Complex& operator+=(Complex Tmp);
    Complex operator*(Complex Tmp);
    bool operator==(Complex Tmp);
    bool operator!=(Complex Tmp) { return !(*this == Tmp); }
    bool IsNotZero();
    ~Complex() {}
};

class Matrix {
 private:
    int size;
    int non;
```

```
    std::vector<Complex> Entry;
    std::vector<int> irows;
    std::vector<int> shtcols;

 public:
    Matrix(int _size = 0, int _non = 0, const std::vector<Complex>& _Entry =
           std::vector<Complex>(), const std::vector<int>& _irows =
           std::vector<int>(), const std::vector<int>& _shtcols =
           std::vector<int>()): size(_size), non(_non), Entry(_Entry),
           irows(_irows), shtcols(_shtcols) {}
    Matrix(const Matrix& Tmp): size(Tmp.size), non(Tmp.non), Entry(Tmp.Entry),
                               irows(Tmp.irows), shtcols(Tmp.shtcols) {}
    Matrix& operator=(const Matrix& Tmp);
    int GetSize() { return this->size; }
    int GetNon() { return this-> non; }
    std::vector<Complex> GetEntry() { return this->Entry; }
    std::vector<int> GetIrows() { return this->irows; }
    std::vector<int> GetShtcols() { return this->shtcols; }
    Matrix& ClearMatrix();
    Matrix& RandomMatrix(int size, int dist, int cnt = -1, int seed = 0);
    bool operator==(Matrix Tmp);
    bool operator!=(const Matrix& Tmp) { return !(*this == Tmp); }
    Matrix T();
    Matrix operator^(Matrix B);
    Matrix operator*(Matrix B);
    ~Matrix();
};

#endif  //
   MODULES_TASK_4_OLYNIN_A_MULT_SPARSE_CC_COMPLEX_MAT_MULT_SPARSE_CC_COMPLEX_MAT_H_



// mult_sparse_cc_complex_mat.cpp
// Copyright 2022 Olynin Alexander
#include "../../../3rdparty/unapproved/unapproved.h"
#include
    "../../modules/task_4/olynin_a_mult_sparse_cc_complex_mat/mult_sparse_cc_complex_mat

std::vector<Complex> Complex::InitVec(std::vector<double> rls,
                                      std::vector<double> ims) {
    std::vector<Complex> Ent(rls.size());
    if (rls.size() == ims.size()) {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], ims[i]);
        }
    } else {
        for (size_t i = 0; i < rls.size(); i++) {
            Ent[i] = Complex(rls[i], 0);
        }
    }
    return Ent;
}
Complex& Complex::operator=(Complex Tmp) {
    this->rl = Tmp.rl;
    this->im = Tmp.im;
    return *this;
}
```

```
Complex Complex::operator*(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl * Tmp.rl - this->im * Tmp.im;
    Ans.im = this->rl * Tmp.im + this->im * Tmp.rl;
    return Ans;
}
Complex Complex::operator+(Complex Tmp) {
    Complex Ans;
    Ans.rl = this->rl + Tmp.rl;
    Ans.im = this->im + Tmp.im;
    return Ans;
}
Complex& Complex::operator+=(Complex Tmp) {
    this->rl += Tmp.rl;
    this->im += Tmp.im;
    return *this;
}
bool Complex::operator==(Complex Tmp) {
    if ((this->rl - Tmp.rl < 0.00001) && (this->im - Tmp.im < 0.00001)) {
        return true;
    } else {
        return false;
    }
}
bool Complex::IsNotZero() {
    bool ans = false;
    const double ZeroLike = 0.000001;
    if ((fabs(this->rl) > ZeroLike) || (fabs(this->im) > ZeroLike)) {
        ans = true;
    }
    return ans;
}
Matrix& Matrix::operator=(const Matrix& Tmp) {
    this->size = Tmp.size;
    this->non = Tmp.non;
    this->Entry = Tmp.Entry;
    this->irows = Tmp.irows;
    this->shtcols = Tmp.shtcols;

    return *this;
}
Matrix& Matrix::ClearMatrix() {
    this->size = 0;
    this->non = 0;
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();

    return *this;
}
Matrix& Matrix::RandomMatrix(int size, int dist, int cnt, int seed) {
    this->ClearMatrix();
    this->size = size;
    std::mt19937 gen(time(0));
    gen.seed(seed);
    if (cnt < 0) {
        cnt = static_cast<int>(size * 0.01);
    }
```

```
    this->non = cnt * size;
    this->Entry.resize(cnt * size);
    this->irows.resize(cnt * size);
    this->shtcols.resize(size + 1);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < cnt; j++) {
            bool flag;
            do {
                this->irows[i * cnt + j] = gen() % size + 1;
                flag = false;
                for (int k = 0; k < j; k++) {
                    if (this->irows[i * cnt + j] == this->irows[i * cnt + k]) {
                        flag = true;
                    }
                }
            } while (flag);
        }
        for (int j = 0; j < cnt - 1; j++) {
            for (int k = 0; k < cnt - 1; k++) {
                if (this->irows[i * cnt + k] > this->irows[i * cnt + k + 1]) {
                    int tmp = this->irows[i * cnt + k];
                    this->irows[i * cnt + k] = this->irows[i * cnt + k + 1];
                    this->irows[i * cnt + k + 1] = tmp;
                }
            }
        }
    }
    for (int i = 0; i < cnt * size; i++) {
        this->Entry[i].SetRl(gen() % dist + 1);
        this->Entry[i].SetIm(gen() % dist + 1);
    }
    int sum = 1;
    for (int i = 0; i < size + 1; i++) {
        this->shtcols[i] = sum;
        sum += cnt;
    }

    return *this;
}
bool Matrix::operator==(Matrix Tmp) {
    bool ans = true;
    if (this->non != Tmp.non) {
        return false;
    }
    if (this->size != Tmp.size) {
        return false;
    }
    for (int i = 0; i < this->non; i++) {
        if (this->Entry[i] != Tmp.Entry[i]) {
            return false;
        }
    }
    if (this->irows != Tmp.irows) {
        return false;
    }
    if (this->shtcols != Tmp.shtcols) {
        return false;
    }
```

```
    return ans;
}
Matrix Matrix::T() {
    Matrix Ans;
    Ans.non = this->non;
    Ans.size = this->size;
    Ans.Entry.resize(this->non);
    Ans.irows.resize(this->non);
    Ans.shtcols.resize(this->size + 1);

    for (int i = 0; i < this->non; i++) {
        Ans.shtcols[this->irows[i] - 1]++;
    }
    int sum = 1;
    for (int i = 0; i < this->size + 1; i++) {
        int tmp = Ans.shtcols[i];
        Ans.shtcols[i] = sum;
        sum += tmp;
    }
    std::vector<int> shtcols_tmp = Ans.shtcols;
    for (int i = 0; i < this->size; i++) {
        for (int j = this->shtcols[i]; j < this->shtcols[i + 1]; j++) {
            int r_index = this->irows[j - 1];
            int i_index = shtcols_tmp[r_index - 1];
            Ans.irows[i_index - 1] = i + 1;
            Ans.Entry[i_index - 1] = this->Entry[j - 1];
            shtcols_tmp[r_index - 1]++;
        }
    }
    return Ans;
}
Matrix Matrix::operator^(Matrix B) {
    if (this->size != B.size) {
        return Matrix();
    }
    Matrix A = this->T();

    std::vector<Complex> EntRes;
    std::vector<int> irows_res;
    std::vector<int> shtcol_res = { 1 };

    int non_counter = 1;
    for (int j = 0; j < B.size; j++) {
        std::vector<int> ip(A.size, 0);
        for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
            ip[B.irows[i-1]-1] = i;
        }
        for (int i = 0; i < A.size; i++) {
            Complex Sum;
            for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                int irow = A.irows[k-1];
                int p = ip[irow-1];
                if (p) {
                    Sum += B.Entry[p-1] * A.Entry[k-1];
                }
            }
            if (Sum.IsNotZero()) {
                    irows_res.push_back(i+1);
```

```
                    EntRes.push_back(Sum);
                    non_counter++;
                }
            }
            shtcol_res.push_back(non_counter);
        }
        Matrix Ans(A.size, non_counter-1, EntRes, irows_res, shtcol_res);
        return Ans;
    }
    Matrix Matrix::operator*(Matrix B) {
        int num_threads = 8;
        if (this->size != B.size) {
            return Matrix();
        }
        Matrix A = this->T();
        if (A.size < num_threads) {
            num_threads = A.size;
        }

        std::vector<std::thread> threads;
        std::vector<std::vector<Complex>> Entry_col(num_threads);
        std::vector<std::vector<int>> irows_col(num_threads);
        std::vector<int> counter(A.size);
        int thread_index = 0;
        int group = ceil(static_cast<float>(A.size) /
                         static_cast<float>(num_threads));

        for (int _j = 0; _j < B.size; _j += group) {
            threads.push_back(std::thread([&](int ind, int start, int end) {
                for (int j = start; j < end; j++) {
                    std::vector<int> ip(A.size, 0);
                    int non_counter = 0;
                    for (int i = B.shtcols[j]; i < B.shtcols[j+1]; i++) {
                        int irow = B.irows[i-1];
                        ip[irow-1] = i;
                    }
                    for (int i = 0; i < A.size; i++) {
                        Complex Sum;
                        for (int k = A.shtcols[i]; k < A.shtcols[i+1]; k++) {
                            int irow = A.irows[k-1];
                            int p = ip[irow-1];
                            if (p) {
                                Sum += B.Entry[p-1] * A.Entry[k-1];
                            }
                        }
                        if (Sum.IsNotZero()) {
                            Entry_col[ind].push_back(Sum);
                            irows_col[ind].push_back(i+1);
                            non_counter++;
                        }
                    }
                    counter[j] += non_counter;
                }
            }, thread_index, _j, fmin(_j + group, B.size)));
            thread_index++;
        }
        for (size_t i = 0; i < threads.size(); i++) {
            threads[i].join();
```

```
    }

    std::vector<Complex> EntryRes;
    std::vector<int> irowsres;
    for (int i = 0; i < num_threads; i++) {
        EntryRes.insert(EntryRes.end(),
                        Entry_col[i].begin(), Entry_col[i].end());
        irowsres.insert(irowsres.end(),
                        irows_col[i].begin(), irows_col[i].end());
    }
    std::vector<int> shtcolsres = { 1 };
    int sum = 1;
    for (int i = 0; i < A.size; i++) {
        sum += counter[i];
        shtcolsres.push_back(sum);
    }

    Matrix Ans(A.size, EntryRes.size(), EntryRes, irowsres, shtcolsres);
    return Ans;
}
Matrix::~Matrix() {
    this->Entry.clear();
    this->irows.clear();
    this->shtcols.clear();
}



// main.cpp
// Copyright 2022 Olynin Alexander
#include <gtest/gtest.h>
#include <time.h>
#include <vector>
#include "./mult_sparse_cc_complex_mat.h"

TEST(Class_Complex, Complex_creation) {
    EXPECT_NO_THROW(Complex A(1.2, 2.3));
}

TEST(Class_Complex, Complex_getters) {
    Complex A(1.2, 2.3);

    EXPECT_EQ(A.GetRl(), 1.2);
    EXPECT_EQ(A.GetIm(), 2.3);
}

TEST(Class_Complex, Complex_comparison) {
    Complex A(1.2, 2.3);
    Complex B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Complex, Complex_multiplication) {
    Complex A(1.2, 2.3);
    Complex B(3.1, -3.2);
    Complex Res = A * B;
    Complex Res_exp(11.08, 3.29);
```

```
    EXPECT_TRUE(Res == Res_exp);
}

TEST(Class_Matrix, Sparse_matrix_creation) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    EXPECT_NO_THROW(Matrix A(size, non, Ent, irows, shtcol));
}

TEST(Class_Matrix, Sparse_matrix_comparison) {
    int non = 6;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };
    Matrix A(size, non, Ent, irows, shtcol);
    Matrix B(A);

    EXPECT_TRUE(A == B);
}

TEST(Class_Matrix, Sparse_matrix_transposition) {
    int non = 6;
    int size = 4;

    std::vector<Complex> Ent;
    Complex Tmp;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 0  3   0   7
    * 0  0   8   0
    * 0  0   0   0
    * 9  0  15  16
    *
    */
    A = A.T();
    /*
    * Matrix A.T should be
    *
    * 0  0  0   9
```

```
     * 3  0  0   0
     * 0  8  0  15
     * 7  0  0  16
     *
     */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({3, 7, 8, 9, 15, 16});
    std::vector<int> irows_res = { 2, 4, 3, 1, 3, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 4, 7 };
    Matrix Res(size, non, EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(A == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_onlyreal) {
    int non = 6;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 9, 3, 8, 15, 7, 16 });
    std::vector<int> irows = { 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 7 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
     * Matrix A is
     *
     * 0  3   0   7
     * 0  0   8   0
     * 0  0   0   0
     * 9  0  15  16
     *
     */

    Ent = Tmp.InitVec({ 2, 3, 1, 8, 4, 5 });
    irows = { 1, 3, 1, 2, 3, 4 };
    shtcol = { 1, 3, 4, 6, 7 };

    Matrix B(size, non, Ent, irows, shtcol);

    /*
     * Matrix B is
     *
     * 2  1  0  0
     * 0  0  8  0
     * 3  0  4  0
     * 0  0  0  5
     *
     */

    Matrix C = A * B;

    /*
     * Matrix C should be
     *
```

```
     * 0   0  24  35
     * 24  0  32   0
     * 0   0   0   0
     * 63  9  60  80
     *
     */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ 24, 63, 9, 24, 32, 60, 35, 80 });
    std::vector<int> irows_res = { 2, 4, 4, 1, 2, 4, 1, 4 };
    std::vector<int> shtcol_res = { 1, 3, 4, 7, 9 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex) {
    int non = 5;
    int size = 4;
    Complex Tmp;

    std::vector<Complex> Ent;
    Ent = Tmp.InitVec({ 4, 8, 18, 15, 2 }, { 20, 4, 4, 11, 7 });
    std::vector<int> irows = { 1, 4, 2, 3, 1 };
    std::vector<int> shtcol = { 1, 2, 3, 5, 6 };

    Matrix A(size, non, Ent, irows, shtcol);

    /*
    * Matrix A is
    *
    * 4+20i  0    0     2+7i
    * 0      0    18+4i   0
    * 0      0    15+11i  0
    * 0   8+4i    0       0
    *
    */

    Ent = Tmp.InitVec({ 4, 1, 12, 20, 9 }, { 14, 3, 2, 13, 10 });
    irows = { 2, 2, 4, 1, 1 };
    shtcol = { 1, 2, 4, 5, 6 };

    Matrix B(size, non, Ent, irows, shtcol);

    /*
    * Matrix B is
    *
    * 0      0      20+13i  9+10i
    * 4+14i  1+3i   0           0
    * 0      0      0           0
    * 0      12+2i  0           0
    *
    */

    Matrix C = A * B;

    /*
    * Matrix C should be
```

```
    *
    * 0          10+88i   -180+452i   -164+220i
    * 0          0        0                   0
    * 0          0        0                   0
    * -24+128i   -4+28i   0                   0
    *
    */

    std::vector<Complex> EntRes;
    EntRes = Tmp.InitVec({ -24, 10, -4, -180, -164 },
                         { 128, 88, 28, 452, 220 });
    std::vector<int> irows_res = { 4, 1, 4, 1, 1 };
    std::vector<int> shtcol_res = { 1, 2, 4, 5, 6 };
    Matrix Res(size, EntRes.size(), EntRes, irows_res, shtcol_res);

    EXPECT_TRUE(C == Res);
}

TEST(Class_Matrix, Sparse_matrix_random_matrix_creation) {
    Matrix A;
    int size = 10;
    int dist = 100;
    int cnt = 1;
    int seed = 0;

    EXPECT_NO_THROW(A.RandomMatrix(size, dist, cnt, seed));
}

TEST(Class_Matrix, Sparse_matrix_multiplication_complex_small) {
    int size = 10;
    int dist = 1000;
    int cnt = 1;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 4);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);

    Matrix C_seq = A ^ B;
    Matrix C_par = A * B;
    EXPECT_TRUE(C_seq == C_par);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_meduim_time_perfomance) {
    clock_t start, end;
    double seq_time, std_time;
    int size = 500;
    int dist = 1000;
    int cnt = 5;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = clock();
    Matrix C_seq = A ^ B;
    end = clock();
    std::cout << "SEQ␣Spended␣time␣->␣" << (end - start + .0) / CLOCKS_PER_SEC
            << "␣<-␣seconds" << std::endl;
    seq_time = (end - start + .0) / CLOCKS_PER_SEC;
```

```
    start = clock();
    Matrix C_std = A * B;
    end = clock();
    std::cout << "STD Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    std_time = (end - start + .0) / CLOCKS_PER_SEC;
    std::cout << "Performance improvement by - | " << seq_time / std_time
            << " | - times" << std::endl;
    EXPECT_TRUE(C_seq == C_std);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_large_time_perfomance) {
    clock_t start, end;
    double seq_time, std_time;
    int size = 1000;
    int dist = 1000;
    int cnt = 10;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = clock();
    Matrix C_seq = A ^ B;
    end = clock();
    std::cout << "SEQ Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    seq_time = (end - start + .0) / CLOCKS_PER_SEC;

    start = clock();
    Matrix C_std = A * B;
    end = clock();
    std::cout << "STD Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    std_time = (end - start + .0) / CLOCKS_PER_SEC;
    std::cout << "Performance improvement by - | " << seq_time / std_time
            << " | - times" << std::endl;
    EXPECT_TRUE(C_seq == C_std);
}

TEST(Class_Matrix, Sparse_matrix_mult_complex_ext_large_time_perfomance) {
    clock_t start, end;
    double seq_time, std_time;
    int size = 2500;
    int dist = 100000;
    int cnt = 25;
    Matrix A;
    A.RandomMatrix(size, dist, cnt, 0);
    Matrix B;
    B.RandomMatrix(size, dist, cnt, 1);
    start = clock();
    Matrix C_seq = A ^ B;
    end = clock();
    std::cout << "SEQ Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    seq_time = (end - start + .0) / CLOCKS_PER_SEC;

    start = clock();
```

```
    Matrix C_std = A * B;
    end = clock();
    std::cout << "STD Spended time -> " << (end - start + .0) / CLOCKS_PER_SEC
            << " <- seconds" << std::endl;
    std_time = (end - start + .0) / CLOCKS_PER_SEC;
    std::cout << "Performance improvement by -| " << seq_time / std_time
            << " |- times" << std::endl;
    EXPECT_TRUE(C_seq == C_std);
}
```