

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н. И. Лобачевского»
(ННГУ)**
Институт информационных технологий, математики и механики
Кафедра программной инженерии
Направление подготовки: «Программная инженерия»

ОТЧЕТ

по лабораторным работам курса «Параллельное
программирование» на тему

**«Умножение плотных матриц. Элементы типа
double. Блочная схема, алгоритм Фокса»**

Выполнила: студентка
группы 381608

Поздеева В. В.

Проверил:

Доцент кафедры
МОиСТ, к.т.н. Сысоев А. В.

Нижний Новгород

2019

Содержание

Введение	3
Постановка задачи	4
Метод решения	5
Описание алгоритма	5
Упрощение алгоритма для параллельной версии	6
Результаты экспериментов	8
Результаты параллельной программы с использованием OpenMP	8
Результаты параллельной программы с использованием TBB	9
Выводы	9
Заключение	11
Литература и источники	12
Приложение	13
Приложение 1	13
Приложение 2	15
Приложение 3	17

Введение

Умножение матриц это одна из основных операций, производимых над матрицами и является одной из основных задач матричных вычислений. Сейчас существует множество алгоритмов реализующих эту операцию, они бывают ленточными и блочными. Есть два наиболее известных блочных алгоритма: Кэннона и Фокса. В данной работе будет более подробно рассмотрен алгоритм Фокса.

Умножение матриц широко применяется в различных сферах. Алгоритм умножения является одним из немногих, которые позволяют эффективно использовать вычислительные ресурсы. Поэтому многие другие алгоритмы стараются привести к матричному умножению.

Постановка задачи

В данной работе необходимо реализовать умножение матриц с помощью блочного алгоритма Фокса. На вход поступает две матрицы A и B, со значениями типа double, на выходе – матрица $C = A * B$.

Таким образом, для достижения поставленной задачи необходимо:

1. Реализовать параллельный алгоритм Фокса с помощью технологий OpenMP и TBB.
2. Реализовать последовательное умножение матриц, для проверки корректности работы параллельного алгоритма и оценки эффективности.

Метод решения

Описание алгоритма

Алгоритм Фокса предполагает разбиение исходных матриц (**A** и **B**) и результирующую матрицу (**C**) на блоки. Будем предполагать, что матрицы являются квадратными размера $N \times N$. Тогда количество блоков(**q**), по вертикали и горизонтали равно. Размер блока (**k**) равен N/q .

Разделим решение на подзадачи. Каждая подзадача будет отвечать за вычисление блока **C_{ij}**. В качестве нумерации подзадач будем использовать индексы размещаемых в ней блоков. То есть в ходе вычислений на каждой подзадаче (**i, j**) располагается 4 блока:

1. **C_{ij}** – блок результата
2. **A[~]_{ij}** - блок, получаемый в ходе вычислений.
3. **B[~]_{ij}** - блок, получаемый в ходе вычислений (в нем же размещается блок матрицы **B** перед началом вычислений)
4. **A_{ij}** – блок, получаемый до начала вычислений, остается неизменным.

В соответствии с алгоритмом:

1. Блоки **A_{ij}** и **B_{ij}** рассылаются по подзадачам (**i, j**), блоки **C_{ij}** обнуляются.
2. На каждой итерации **i=1, 2, ..., q** выполняется:
 - а. Блок **A_{lm}** пересылается всем подзадачам строки **l**, где **m = (i + l) mod q**
 - б. К блоку **C_{ij}** добавляется произведение блоков **A[~]_{ij}** и **B[~]_{ij}**
 - в. Блок **B[~]_{ij}** пересылается подзадаче, расположенной выше (блоки подзадач из первой строки посылаются блокам подзадач из последней строки).

Предположим есть матрицы **A** и **B**, количество блоков равно 3 и количество потоков равно 9, тогда после первой итерации цикла потоки будут иметь следующие блоки (каждая клеточка таблицы отвечает за один поток):

A00	A00	A00
A11	A11	A11
A22	A22	A22

B00	B01	B02
B10	B11	B12
B20	B21	B22

После выполнения второй итерации цикла:

A01	A01	A01	B10	B11	B12
A12	A12	A12	B20	B21	B22
A20	A20	A20	B00	B01	B02

После выполнения третьей итерации:

A02	A02	A02	B20	B21	B22
A10	A10	A10	B00	B01	B02
A21	A21	A21	B10	B11	B12

Упрощение алгоритма для параллельной версии

Технологии OpenMP и TBB предназначены для реализации параллельных программ для систем с общей памятью. Алгоритм Фокса разрабатывался для систем с распределённой памятью. Он позволяет максимально распараллелить операцию умножения матриц, при этом не происходит одновременного обращения потоков к одним и тем же блокам данных. В системах с общей памятью нет необходимости распределять данные между вычислительными элементами, каждый поток имеет к ним доступ, поэтому алгоритм Фокса немного упрощается. Необходимо: чтоб каждый поток обрабатывал только свою строку блоков матрицы A. Соответственно получается, что потоки имеют доступ к обеим матрицам A и B, но запись результатов осуществляют в разные блоки, каждый в свою строку матрицы C, то есть не происходит конфликтов с данными.

Таким образом, необходимо обеспечить обработку каждым потоком только предназначенных ему строк, а для этого нужно распараллелить цикл умножения матриц.

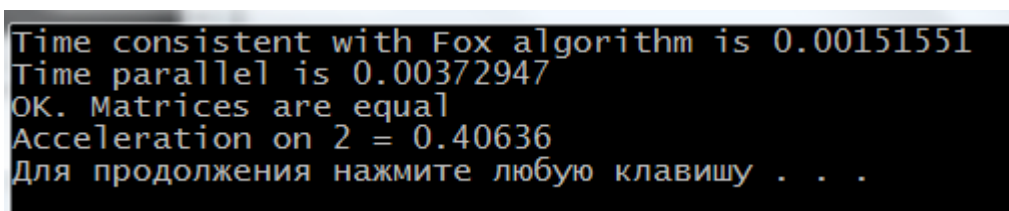
В версии программы с использованием технологии OpenMP, для этого используется директива `#pragma omp parallel for`. Она позволяет распределять итерации цикла между потоками.

В версии с применением технологии TBB, для распараллеливания цикла используется функция `tbb::parallel_for()`. Первым параметром функции является итерационное пространство, для задания количества итераций цикла, вторым – функтор, класс для реализации вычислений.

Полные коды реализаций представлены в приложении. Последовательная версия – Приложение 1, версия с OpenMP – Приложение 2, версия с TBB – Приложение 3.

Для подтверждения корректности в программе осуществляется сравнение результатов, полученных параллельным путем и последовательным. После выполнения выводится соответствующее сообщение: в случае совпадения «OK. Matrices are equal», в случае ошибки «ERROR. Matrices are not equal».

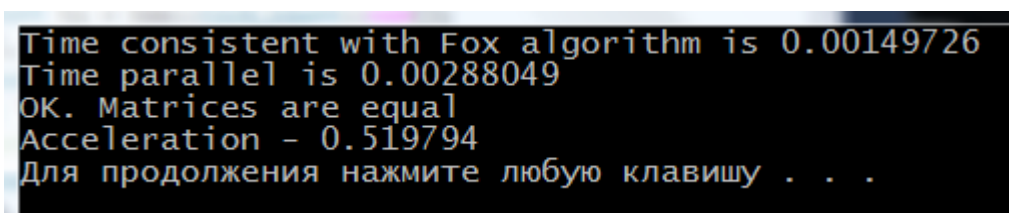
Запустим программу для матрицы 100 на 100, для версии с OpenMP.



```
Time consistent with Fox algorithm is 0.00151551
Time parallel is 0.00372947
OK. Matrices are equal
Acceleration on 2 = 0.40636
Для продолжения нажмите любую клавишу . . .
```

Рисунок 1. Результаты проверки корректности. Версия OpenMP

Запустим программу для матрицы 100 на 100, для версии с TBB.



```
Time consistent with Fox algorithm is 0.00149726
Time parallel is 0.00288049
OK. Matrices are equal
Acceleration - 0.519794
Для продолжения нажмите любую клавишу . . .
```

Рисунок 2. Результаты проверки корректности. Версия TBB

Видим, что результаты одинаковы, а это значит, что алгоритм работает корректно в обоих случаях.

Результаты экспериментов

Были проведены несколько запусков программы для сравнения эффективности параллельных версий. Запуски проводились на персональном компьютере со следующими характеристиками:

Windows 7, x64

Процессор: Intel® Pentium® CPU 2117U

Логических процессов: 2

Оперативная память: 8 Гб

Результаты экспериментов были получены при умножении двух матриц, с элементами типа double, размеров: 100, 500 и 1000 элементов. Замеры проводились на 1, 2, 4, 8 потоках. В таблицах 1 и 2 представлено среднее ускорение за 10 экспериментов. Также, для наглядности, были построены графики зависимостей ускорения от числа потоков.

Результаты параллельной программы с использованием OpenMP

Число элементов Количество потоков	100	500	1000
1	0,51	0,68	0,9
2	0,58	1,62	1,92
4	0,79	1,35	1,79
8	0,74	1,45	1,73

Таблица 1. Ускорение при использовании OpenMP(сек)

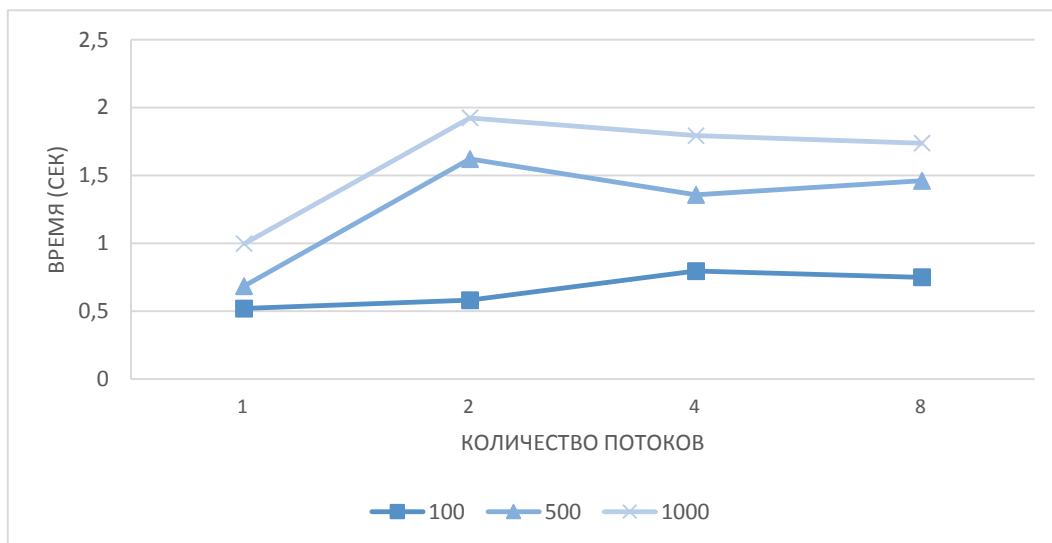


График 1. Зависимость ускорения от числа потоков.

Результаты параллельной программы с использованием TBB

Число элементов Количество потоков	100	500	1000
1	0,518	0,92	0,99
2	0,58	1,96	1,85
4	0,5	1,83	1,79
8	0,48	1,86	7,16

Таблица 2. Ускорение при использовании TBB (сек)

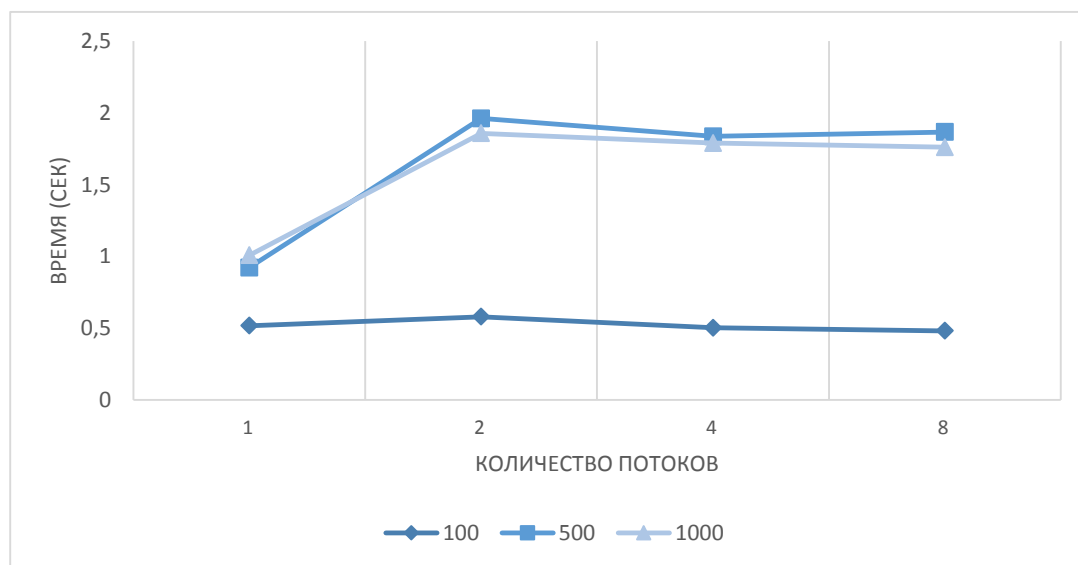


График 2. Зависимость ускорения от числа потоков

Выводы

На основании проведенных опытов можно сделать несколько выводов.

При небольших размерах матриц (порядка двух сотен) использование параллельной версии алгоритма не имеет смысла. Это можно объяснить тем, что доля вычислений, расходуемая на накладные расходы по созданию потоков достаточно большая. Поэтому эффективней использовать последовательный алгоритм.

При большом объеме данных видно, что ускорение близко к идеальному (почти равно 2). Такое хорошее ускорение достигается за счет того, что умножение матриц хорошо распараллеливается, оно имеет большое число операций, которые можно выполнять независимо. Из этого можно сделать вывод, что данный алгоритм действительно эффективно распараллеливает вычисления.

Так же можно заметить, что в случае числа потоков больших чем два, возникает не значительный спад ускорения, и далее при увеличении количества потоков ускорение остается прежним. Такие результаты наблюдаются и в случае с использованием OpenMP, и

с использованием ТВВ, их можно объяснить тем, что компьютер, на котором проводились эксперименты, ограничен только двумя ядрами процессора. Соответственно следует вывод о том, что не имеет смысла использования числа потоков больших, чем количество доступных вычислительных узлов компьютера. Кроме того, можно сделать предположение, что в случае запуска данной программы на компьютере с количеством логических процессов равных 4 либо больше, то ускорение будет также увеличиваться, однако, очевидно, что это не может продолжаться до бесконечности, потому что накладные расходы в любом случае присутствуют, и с увеличением масштабов вычислений они не уменьшаются.

Заключение

В данной работе был реализован блочный алгоритм умножения матриц методом Фокса. Представлены три реализации: последовательная, параллельная с использованием технологии OpenMP и параллельная с использованием технологии TBB. Также были проведены эксперименты для оценки ускорения параллельных версий алгоритма.

Из проведенных экспериментов следует вывод, что алгоритм имеет хорошо распараллеливаемую вычислительную часть. Однако, использование параллельных версий дает хорошее ускорение только при достаточно больших объемах данных. Это объясняется высокой долей накладных расходов, при маленьких размерах матриц. Следует заметить, что обе параллельные версии показали хорошие результаты, однако хорошее ускорение при относительно небольших размерах матриц (~500 элементов) достигается быстрее при использовании технологии TBB.

Литература и источники

1. Антонов А. С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В. А. Садовничий. - М.: Издательство Московского университета, 2012. - 344 с.
2. OpenNET. Алгоритм Фокса [Электронный ресурс]
// https://www.opennet.ru/docs/RUS/linux_parallel/node158.html

Приложение

Приложение 1

Код последовательной версии программы умножения матриц алгоритмом Фокса.

```
// Copyright 2019 Pozdeeva Varvara
#include <iostream>
#include <ctime>
#include <random>
double* CreateRandMatrix(int size) {
    double* tmp = new double[size*size];
    for (int i = 0; i < size*size; i++)
        tmp[i] = (std::rand() % 10000) / 1000.0f;
    return tmp;
}
void CheckEqual(double* A, double*B, int n) {
    for (int i = 0; i < n*n; i++) {
        if (std::fabs(A[i] - B[i]) > 0.000001) {
            std::cout << "ERROR. Matrices are not equal" << std::endl;
            return;
        }
    }
    std::cout << "OK. Matrices are equal" << std::endl;
}
void PrintMatrix(double* A, int N) {
    for (int i = 0; i < N*N; i += N) {
        for (int j = 0; j < N; j++)
            std::cout << A[i + j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
void MultiMatrix(double* A, double* B, double* C, int n, int bSize) {
    for (int i = 0; i < bSize; ++i)
        for (int j = 0; j < bSize; ++j)
            for (int k = 0; k < bSize; ++k) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
}
void Foxs(double* A, double* B, double* C, int n, int q) {
    int bSize = n / q;
    for (int i = 0; i < q; i++)
        for (int j = 0; j < q; j++)
            for (int k = 0; k < q; k++) {
                MultiMatrix(&A[(i*n+j)*bSize], &B[(j*n + k)*bSize], &C[(i*n +
k)*bSize], n, bSize);
            }
}
int main(int argc, char** argv) {
    double *A, *B, *C, *S;
    int N = 500, q = 2;
    if (argc == 3) {
        N = atoi(argv[1]);
        q = atoi(argv[2]);
    }
    srand((unsigned int)time(0));
    A = CreateRandMatrix(N);
    B = CreateRandMatrix(N);
    //PrintMatrix(A, N);
    //PrintMatrix(B, N);
    C = new double[N*N];
    for (int i = 0; i < N*N; i++)
```

```

        C[i] = 0.0;
        S = new double[N*N];
        for (int i = 0; i < N*N; i++)
            S[i] = 0.0;
        clock_t startTime = clock();
        std::cout << "Start work" << std::endl;
        MultiMatrix(A, B, S, N, N);
        clock_t finishTime = clock();
        std::cout << "Finish work. Elapsed time = " << finishTime - startTime << " ms"
<< std::endl;
        startTime = clock();
        Foxs(A, B, C, N, q);
        finishTime = clock();
        std::cout << "Finish work. Fox = " << finishTime - startTime << " ms" <<
std::endl;
        //PrintMatrix(C, N);
        CheckEqual(C, S, N);
        return 0;
    }

```

Приложение 2

Код программы умножения матриц алгоритмом Фокса с использованием технологии OpenMP

```
// Copyright 2019 Pozdeeva Varvara
#include <omp.h>
#include <iostream>
#include <ctime>
#include <random>

double* CreateRandMatrix(int size) {
    double* tmp = new double[size*size];
    for (int i = 0; i < size*size; i++)
        tmp[i] = (std::rand() % 10000) / 1000.0f;
    return tmp;
}

void CheckEqual(double* A, double* B, int n) {
    for (int i = 0; i < n*n; i++) {
        if (std::fabs(A[i] - B[i]) > 0.000001) {
            std::cout << "ERROR. Matrices are not equal" << std::endl;
            return;
        }
    }
    std::cout << "OK. Matrices are equal" << std::endl;
}

int PrintMatrix(double* A, int N) {
    for (int i = 0; i < N*N; i += N) {
        for (int j = 0; j < N; j++)
            std::cout << A[i + j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
    return 1;
}

void MultiMatrix(double* A, double* B, double* C, int n, int bSize) {
    for (int i = 0; i < bSize; ++i)
        for (int j = 0; j < bSize; ++j)
            for (int k = 0; k < bSize; ++k) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
}

void Foxs(double* A, double* B, double* C, int n, int q) {
    int bSize = n / q;
    for (int i = 0; i < q; i++)
        for (int j = 0; j < q; j++)
            for (int k = 0; k < q; k++) {
                MultiMatrix(&A[(i*n+j)*bSize], &B[(j*n + k)*bSize], &C[(i*n +
k)*bSize], n, bSize);
            }
}

void FoxsOmp(double* A, double* B, double* C, int n, int q) {
    int bSize = n / q;
    #pragma omp parallel for
    for (int i = 0; i < q; i++)
        for (int j = 0; j < q; j++)
            for (int k = 0; k < q; k++) {
                MultiMatrix(&A[(i*n + j)*bSize], &B[(j*n + k)*bSize], &C[(i*n
+ k)*bSize], n, bSize);
            }
}

int main(int argc, char** argv) {
    double *A, *B, *C, *C1;
```

```

int N = 4, q = 2;
if (argc == 3) {
    N = atoi(argv[1]);
    q = atoi(argv[2]);
}
srand((unsigned int)time(0));
A = CreateRandMatrix(N);
B = CreateRandMatrix(N);
// PrintMatrix(A, N);
// PrintMatrix(B, N);
C = new double[N*N];
C1 = new double[N*N];
for (int i = 0; i < N*N; i++) {
    C[i] = 0.0;
    C1[i] = 0.0;
}

double startTime = omp_get_wtime();
Focs(A, B, C, N, q);
double finishTime = omp_get_wtime();
double t1 = finishTime - startTime;
std::cout << "Time consistent with Fox algorithm is " << t1 << std::endl;

for (int i = 0; i < N*N; i++)
    C1[i] = 0.0;
startTime = omp_get_wtime();
FocsOmp(A, B, C1, N, q);
finishTime = omp_get_wtime();
std::cout << "Time parallel is " << finishTime - startTime << std::endl;
// PrintMatrix(C1, N);
CheckEqual(C, C1, N);
std::cout << "Acceleration on 2 = " << t1 / (finishTime - startTime) << std::endl;

delete[] A;
delete[] B;
delete[] C;
delete[] C1;
return 0;
}

```


Приложение 3

Код программы умножения матриц алгоритмом Фокса с использованием технологии

TBB

```
// Copyright 2019 Pozdeeva Varvara

#include <tbb\tbb.h>

#include <iostream>
#include <ctime>
#include <random>

double* CreateRandMatrix(int size) {
    double* tmp = new double[size*size];
    for (int i = 0; i < size*size; i++)
        tmp[i] = (std::rand() % 10000) / 1000.0f;
    return tmp;
}

void CheckEqual(double* A, double*B, int n) {
    for (int i = 0; i < n*n; i++) {
        if (std::fabs(A[i] - B[i]) > 0.000001) {
            std::cout << "ERROR. Matrices are not equal" << std::endl;
            return;
        }
    }
    std::cout << "OK. Matrices are equal" << std::endl;
}

int PrintMatrix(double* A, int N) {
    for (int i = 0; i < N*N; i += N) {
        for (int j = 0; j < N; j++)
            std::cout << A[i + j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
    return 1;
}

void MultiMatrix(double* A, double* B, double* C, int n, int bSize) {
    for (int i = 0; i < bSize; ++i)
        for (int j = 0; j < bSize; ++j)
            for (int k = 0; k < bSize; ++k) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
}

class Multiplier {
    double *A, *B, *C;
    int n, q;
public:
    Multiplier(double *a, double *b, double *c,
               int N, int Q) : A(a), B(b), C(c), n(N), q(Q) {}

    void operator()(const tbb::blocked_range2d<int>& r) const {
        int bSize = n / q;
        for (int i = r.rows().begin(); i < r.rows().end(); ++i) {
            for (int j = r.cols().begin(); j < r.cols().end(); ++j) {
                for (int k = 0; k < q; ++k) {
                    MultiMatrix(&A[(i*n + k)*bSize], &B[(k*n + j)*bSize], &C[(i*n +
j)*bSize], n, bSize);
                }
            }
        }
    }
};
```

```

void Foxs(double* A, double* B, double* C, int n, int q) {
    int bSize = n / q;
    for (int i = 0; i < q; i++)
        for (int j = 0; j < q; j++) {
            for (int k = 0; k < q; k++) {
                MultiMatrix(&A[(i*n + j)*bSize], &B[(j*n + k)*bSize], &C[(i*n +
k)*bSize], n, bSize);
            }
        }
}

void FoxsTBB(double* A, double* B, double* C, int n, int q, int grainSize) {
    tbb::task_scheduler_init init;
    tbb::parallel_for(tbb::blocked_range2d<int>(0, q, grainSize, 0, q, grainSize),
Multiplier(A, B, C, n, q));
}

int main(int argc, char** argv) {
    double *A, *B, *C, *C1;
    int N = 500;
    int q = 2;
    if (argc == 3) {
        N = atoi(argv[1]);
        q = atoi(argv[2]);
    }
    srand((unsigned int)time(0));
    A = CreateRandMatrix(N);
    B = CreateRandMatrix(N);
    // PrintMatrix(A, N);
    C = new double[N*N];
    C1 = new double[N*N];
    for (int i = 0; i < N*N; i++) {
        C[i] = 0.0;
        C1[i] = 0.0;
    }
    int grainSize = 1;
    tbb::tick_count t1 = tbb::tick_count::now();
    Foxs(A, B, C, N, q);
    tbb::tick_count t2 = tbb::tick_count::now();
    std::cout << "Time consistent with Fox algorithm is " << (t2 - t1).seconds() <<
std::endl;
    // PrintMatrix(C, N);
    tbb::tick_count t1tbb = tbb::tick_count::now();
    FoxsTBB(A, B, C1, N, q, grainSize);
    tbb::tick_count t2tbb = tbb::tick_count::now();
    std::cout << "Time parallel is " << (t2tbb - t1tbb).seconds() << std::endl;
    // PrintMatrix(C1, N);
    CheckEqual(C, C1, N);

    std::cout << "Acceleration - " << ((t2 - t1).seconds()) / ((t2tbb -
t1tbb).seconds()) << std::endl;

    delete[] A;
    delete[] B;
    delete[] C;
    delete[] C1;

    return 0;
}

```