Министерство образования и науки Российской Федерации
Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчет по лабораторной работе

«Умножение плотных матриц. Элементы типа double. Блочная схема, Алгоритм Кэнона»

Выполнил:
Студент группы 381606-1

_____Н.Р. Касмазюк

Подпись

Проверил:

Проверил: к.ф.-м.н., доц.

_____ А.В. Сысоев

Подпись

Нижний Новгород
2019

Оглавление

Введение

Перемножение матриц является одной из основных операций с матрицами. Чтобы получить произведение матриц *A* и *B* необходимо, чтобы количество столбцов матрицы *A* было равно количеству строк матрицы *B*. Если условие выполняется, произведение матриц определено. Результатом умножения матрицы $A(n \times m)$ и матрицы $B(m \times k)$ будет матрица *C* размерностью $(n \times k)$. Каждый элемент $C_{i,j}$ результирующей матрицы получается умножением строки *i* матрицы *A* на столбец *j* матрицы *B*.

Операция перемножения матриц требует большое количество вычислений. Поэтому для выполнения операции перемножения матриц придумано множество параллельных алгоритмов. Одним из таких алгоритмов является алгоритм Кэннона. Данный алгоритм основан на блочном разбиении матриц. Идея алгоритма состоит в изменении схемы начального распределения блоков перемножаемых матриц между процессорами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки на процессорах могли бы быть перемножены без каких-либо дополнительных передач данных между процессорами. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между процессорами в ходе вычислений может осуществляться с использованием простых коммуникационных операций.

Постановка задачи

Цель данной работы – реализация алгоритма Кэннона, вычисляющего произведение плотных квадратных матриц $n \times n$. Результатом работы программы должна стать матрица размерности $n \times n$, получающаяся произведением первой матрицы на вторую, и время работы алгоритма.

Таким образом, для достижения поставленной цели необходимо выполнить следующие задачи:

1) Реализовать параллельный алгоритм Кэннона перемножения матриц средствами OpenMP.
2) Реализовать параллельный алгоритм Кэннона перемножения матриц средствами Intel TBB.
3) Реализовать последовательный алгоритм перемножения матриц для проверки корректности работы параллельного.
4) Сравнить время работы последовательного и параллельного алгоритмов.
5) Сравнить время работы параллельного алгоритма на матрицах различного порядка и на разном количестве доступных программе процессов для оценки масштабируемости алгоритма.

Данная программа должна поддерживать следующие возможности:

- Ввод размера матрицы.
- Генерация матрицы.
- Подсчет времени исполнения последовательной и параллельной части.
- Проверка корректности результата.

Метод решения

Последовательный алгоритм

Последовательный алгоритм перемножения двух плотных матриц *A* и *B* размерности (*n x n*)
реализуется по формуле:

$$C(i,j) = \sum_{k=0}^{n-1} A(i,k) * B(k,j) \ , \ 0 \le i \ , \ j < n$$

При представлении матриц одномерным массивом размером *n*n* элементов данный алгоритм можно реализовать следующей функцией:

```
void MultMatrix(double* pAMatrix, double* pBMatrix, double* pCMatrix, int blockSize, int
N) {
    for (int i = 0; i < blockSize; ++i)
        for (int j = 0; j < blockSize; ++j)
            for (int k = 0; k < blockSize; ++k) {
                pCMatrix[i * N + j] += pAMatrix[i * N + k] * pBMatrix[k * N + j];
            }
}
```

Тогда сложность такого алгоритма – $O(n^3)$.

Параллельный алгоритм Кэннона

Для работы алгоритма Кэннона процессы выстраиваются в квадратную решетку. Задача каждого процесса – вычислить соответствующий блок результирующей матрицы с помощью операций сложения и перемножения блоков исходных матриц.

Для перемножения блоков исходных матриц может использоваться тот же последовательный алгоритм перемножения матриц.

Схема распараллеливания

Алгоритм Кэннона – блочный алгоритм перемножения матриц. Для корректной работы необходимо, чтобы число процессов p было полным квадратом.
Идея алгоритма:

1) Матрицы $A(n\ x\ n)$ и B (n x n) разбиваются на *p квадратных блоков* Ai,j и Bi,j $(0 \le i, j < \sqrt{p})$ размерностью ($n/\sqrt{p} * n/\sqrt{p}$) каждый.
2) Процессы выстраиваются в квадратную решетку с замыканием краев (решетка-тор)

Описание алгоритма

1) Начальное распределение блоков матриц между процессами:

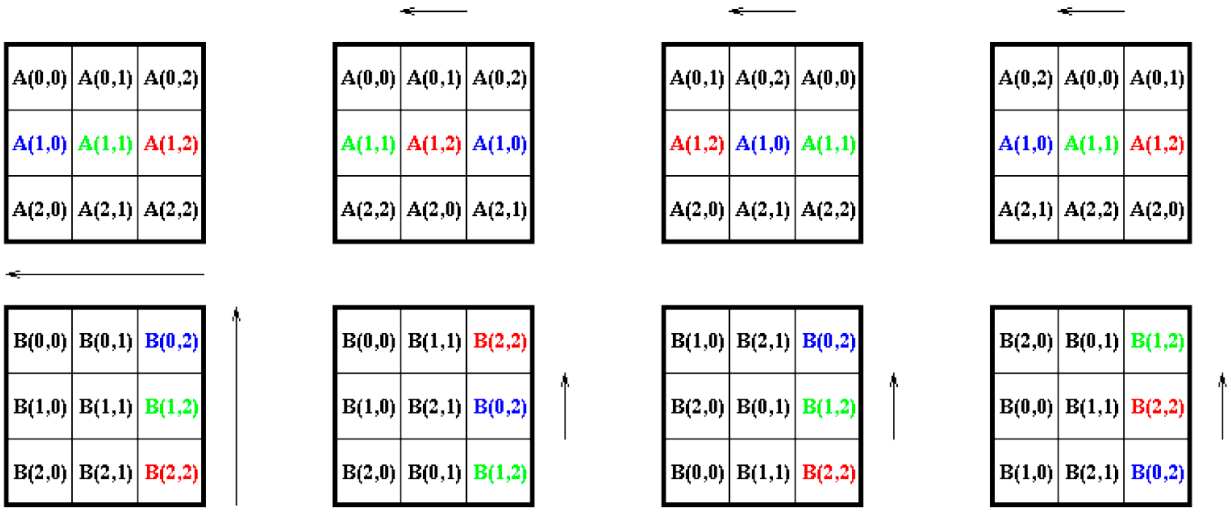
For i,j = 0 to $\sqrt{p} - 1$ do
Отправить блок Ai,j процессу $Pi,(j\text{-}i+\sqrt{p})\%\sqrt{p}$
Отправить блок Bi,j процессу $Pi,(j\text{-}i+\sqrt{p})\%\sqrt{p}$
Endfor;
Пояснение: Начальное распределение матриц следующее: сместить блок Ai,j влево на i позиций, сместить блок Bi,j вверх на j позиций.

2) Вычисление подматрицы Ci,j на процессе Pi,j:

Перемножаем полученные подматрицы и доьавляем результат к Ci,j
Циклично сдвигаем блок Ai,j влево, а блок Bi,j вверх на одну позицию по решетке процессов.
Перемножаем полученные подматрицы и добавляем результат к Ci,j

3) Собрать блоки Ci,j с процессов в результирующую матрицу C(n x n)

Пример:
*C(1,2) = A(1,0) *B(0,2) + A(1,1) * B(1,2) + A(1,2) *B(2,2)*



Первоначальные матрицы

Начальное распределение блоков между процессами

Блоки после первой итерации главного цикла

Блоки после второй итерации главного цикла

Рис. 1. Распределение блочных матриц 3x3 по процессам.

Подтверждение корректности

Подтверждение корректности можно получить, визуально сравнив выведенные матрицы на экран. Для удобства добавлена функция, которая сравнивает результат последовательного алгоритма и параллельного и выводит на экран, равны ли результаты.

В случае верного решения, когда они совпадают, то она выдаст такой результат:

```
result posl. == result par.
```

Иначе программа сообщит об ошибке, что наши результаты не совпадают.

Анализ эффективности алгоритма:

Оценка показателей ускорения и эффективности: Алгоритм Кэннона для своего выполнения требует q итераций, в ходе которых каждый процессор перемножает свой текущие блоки исходных матриц и прибавляет результат к текущему значению блока результирующей матрицы. Общее количество выполняемых при этом операций будет иметь порядок n^3/p. В результате

показатели ускорения и эффективности алгоритма имеют вид: $S_p = \frac{n^3}{(n^3/p)} = p$, $E_p = \frac{n^3}{p(n^3/p)} = 1$.

Описание программной реализации

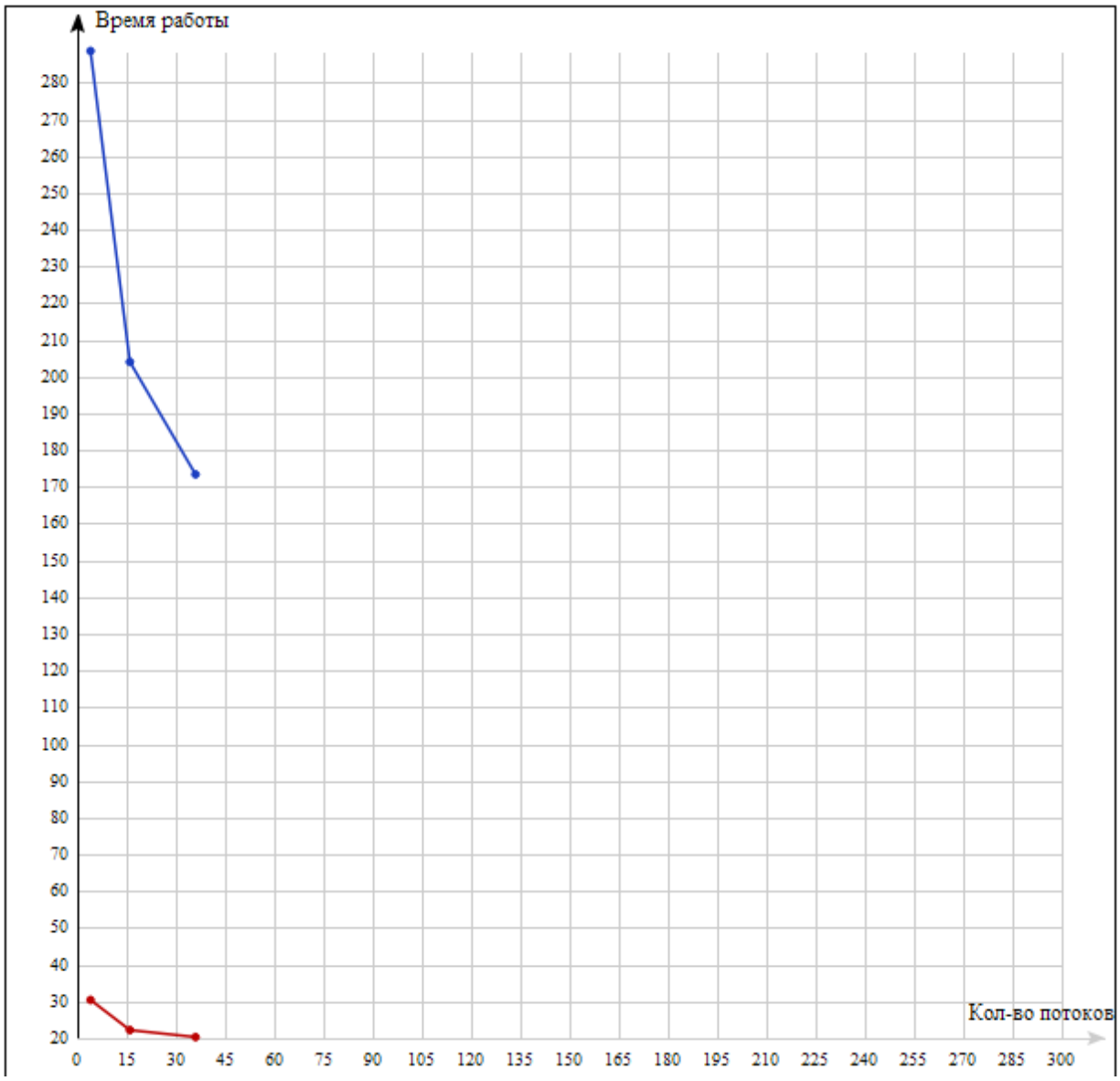
В OpenMP и TBB отсутствуют пересылки, так как используется общая память, поэтому алгоритм сводится к тому, что каждый поток перемножает свои блоки с помощью последовательного алгоритма Кэннона и записывает блок-результат в результирующую матрицу.
В TBB будут использоваться задачи (task), которые будут выполняться потоками по очереди.

Результаты экспериментов

Результаты проведенных экспериментов с использованием TBB.

Время работы алгоритма при различном количестве потоков.

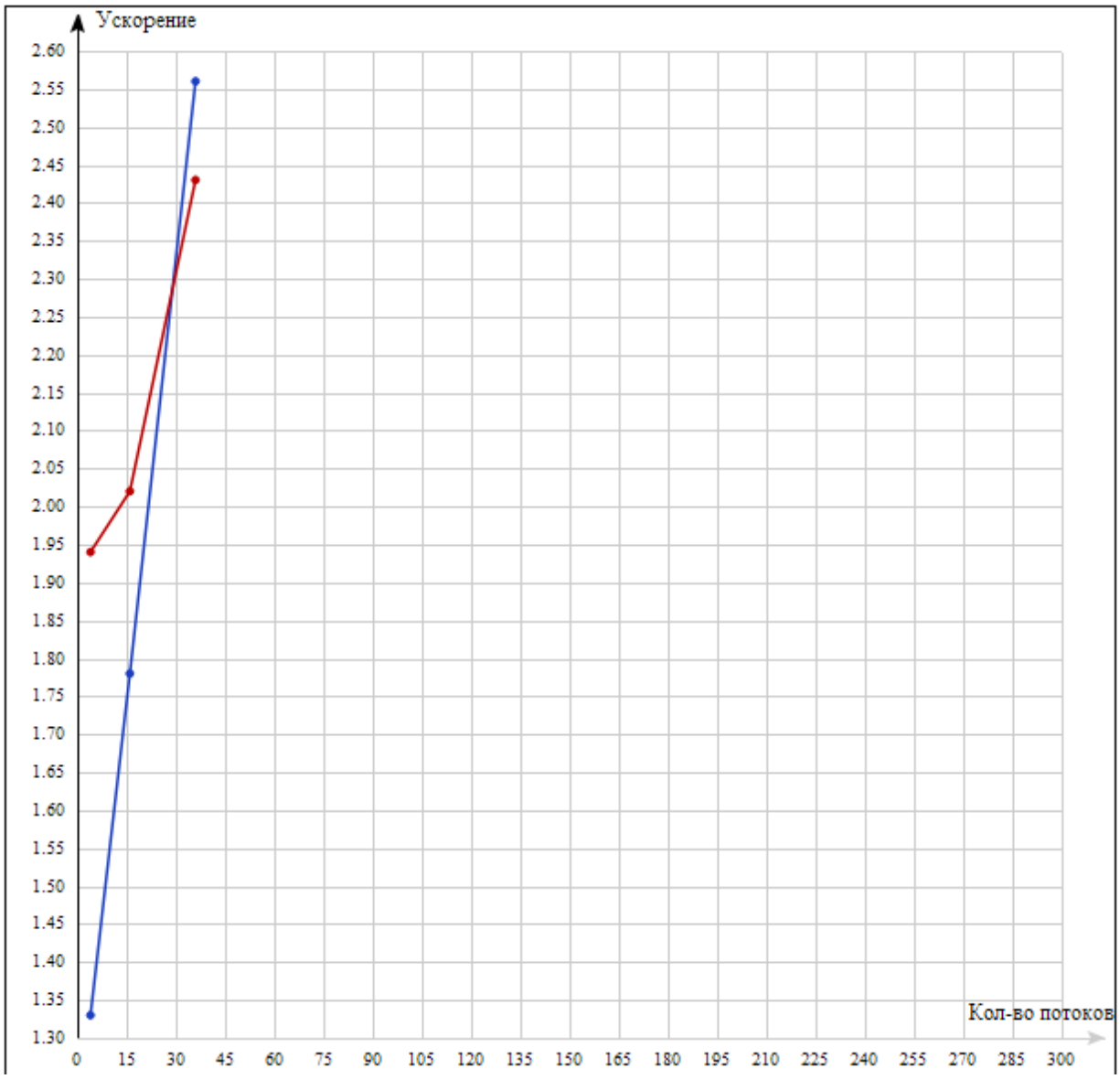
Размер матрицы	Время работы (секунд)		
	4 потока	16 потоков	36 потоков
1296	288.6	204	173.4
658	30.4	22.2	20.3



Ри.с 1. (Синий цвет – 1296x1296, красный – 658x658)

Ускорение алгоритма при различном количестве потоков

Размер матрицы	Ускорение		
	4 потока	16 потоков	36 потоков
1296	1.33	1.78	2.56
658	1.94	2.02	2.43

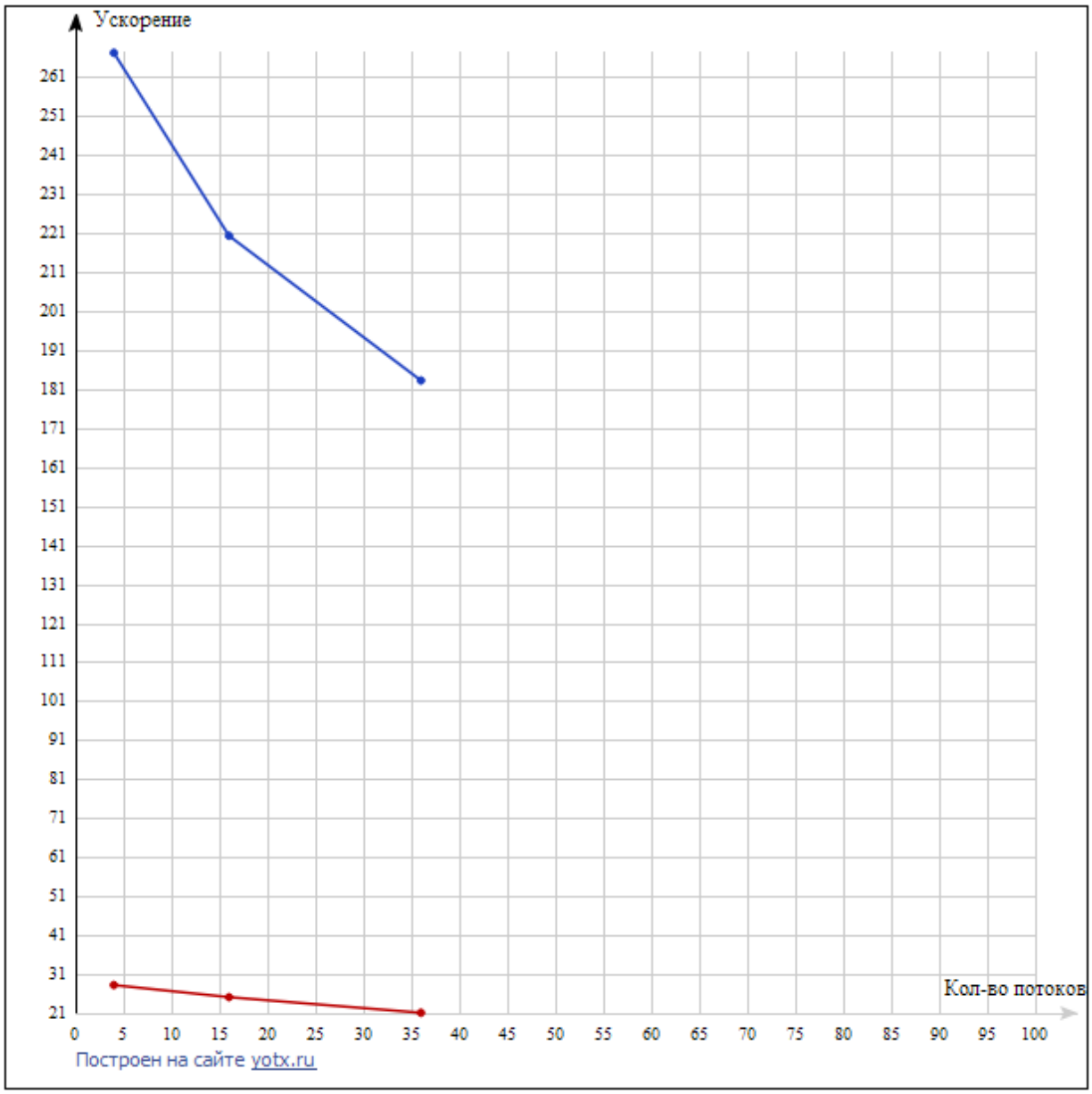


Ри.с 2. (Синий цвет – 1296x1296, красный – 658x658)

Результаты проведенных экспериментов с использованием OpenMP.

Время работы алгоритма при различном количестве потоков.

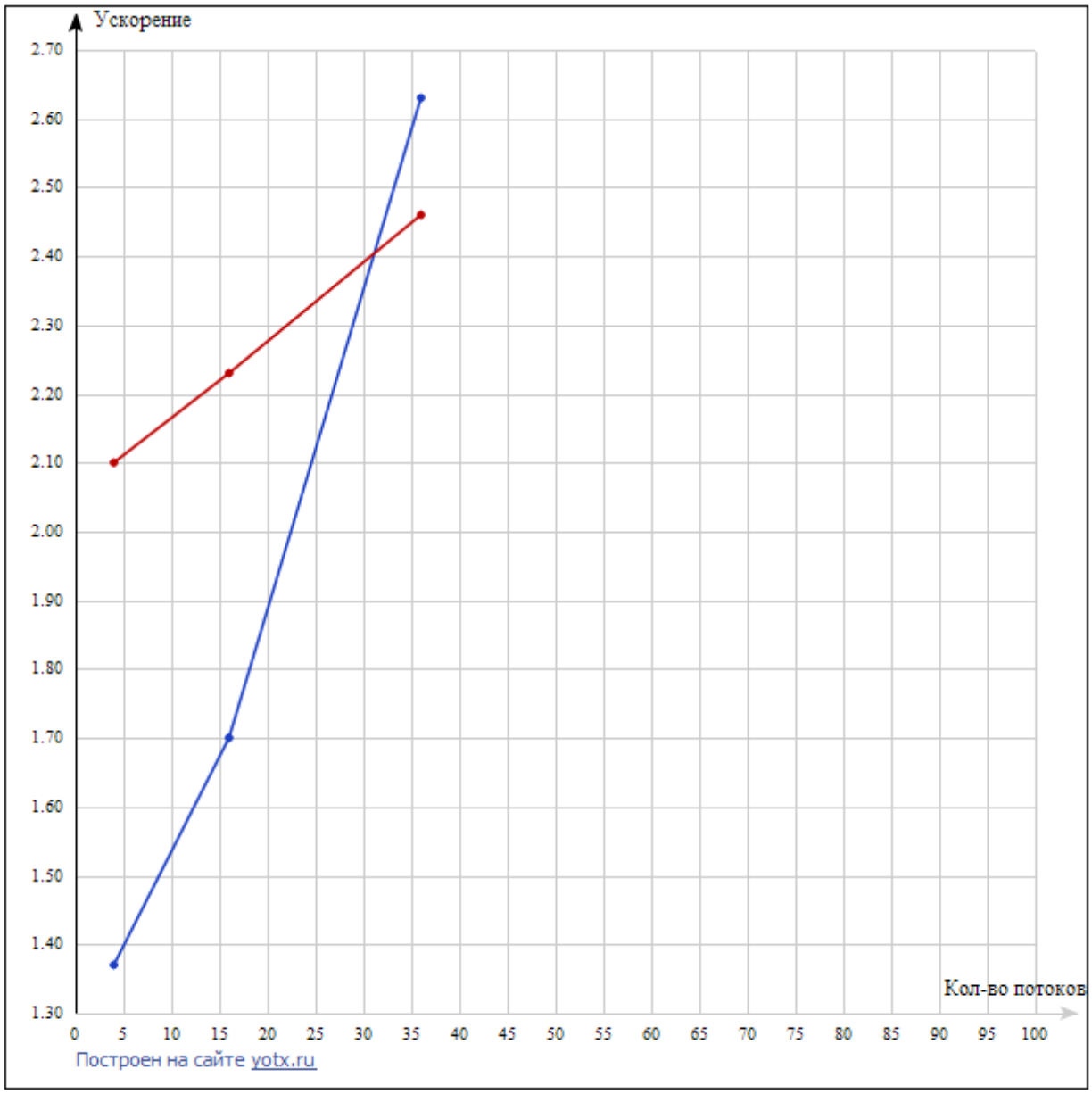
Размер матрицы	Время работы (секунд)		
	4 потока	16 потоков	36 потоков
1296	267.2	220.3	183.2
658	28.2	25.1	21.1



Ри.с 3. (Синий цвет – 1296x1296, красный – 658x658)

Ускорение алгоритма при различном количестве потоков

Размер матрицы	Ускорение		
	4 потока	16 потоков	36 потоков
1296	1.37	1.70	2.63
658	2.1	2.23	2.46



Ри.с 4. (Синий цвет – 1296x1296, красный – 658x658)

Заключение

В ходе выполнения лабораторной работы мы ознакомились с алгоритмами перемножения матриц. В частности, изучили алгоритм Кэннона. Как последовательный, так и параллельный.

По результатам проведенных экспериментов можно сделать следующие выводы:

1) Алгоритм Кэннона на больших матрицах, в среднем, показывает ускорение, равное числу участвующих процессов (что соответствует теоретическим оценкам сложности алгоритма).
2) Алгоритм Кэннона может работать как в несколько раз медленнее, так и в несколько раз быстрее теоретической оценки. На размере матриц 1000x1000 было достигнуто сверхлинейное ускорение. Это явление можно объяснить удачным расположением блоков матриц в кэше процессора, что ускоряет работу с памятью при работе отдельно взятого процесса. Для выполнения последовательного алгоритма необходимы матрицы в полном объеме, а их полностью в кэше разместить нельзя, в результате чего большое количество времени тратится на обращение к оперативной памяти.
3) На маленьких матрицах Алгоритм Кэннона показывает ускорение ниже теоритического, так как большое время тратится на пересылку данных между процессами, а не на сами вычисления.
4) Результаты экспериментов на кластере хорошо демонстрируют пункт 2) выводов. При большом количестве доступных процессов матрицы делятся на небольшие, по отношению к исходному размеру матриц, блоки, которые могут быть закешированы процессором, в результате чего операции над ними выполняются быстро. Для получения одного элемента результирующей матрицы последовательному алгоритму необходима одна строчка первой матрицы и один столбец второй, элементы которого находятся далеко друг от друга в оперативной памяти. В результате, во время выполнения последовательного алгоритма тратится большое количество времени на обмен данными с оперативной памятью.
5) Так же стоит отметить, что OpenMP и TBB отрабатывают почти за одинаковое время, но всё же в ряде примеров TBB проигрывает OpenMP во времени, возможно, потому что в нём нет поддержки привязки потоков к ядрам (affinity) и статического распределения нагрузки.

В результате все поставленные задачи были выполнены, цели достигнуты. Мы приобрели важные навыки в применении параллельного программирования с помощью OpenMP и TBB.

Приложение

OpenMP версия:

```
//
Copyright
2019
Kasmazyuk
Nikita
```

```
#include <omp.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <string>
double* CreateMatrix(int N) {
    double *matrix = new double[N*N];
    return matrix;
}
void PrintMatrix(double* matrix, int N) {
for (int i = 0; i < N*N; i += N) {
        for (int j = 0; j< N; j++)
            std::cout << matrix[i + j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
void RandMatrix(double* matrix1, double* matrix2, int N) {
    for (int i = 0; i < N*N; ++i) {
            matrix1[i] = (std::rand() % 10000) / 1000.0f;
            matrix2[i] = (std::rand() % 10000) / 1000.0f;
        }
}
void ClearMatrix(double *C, int N) {
    for (int i = 0; i < N*N; ++i) {
            C[i] = 0;
    }
}
void MultMatrix(double* A, double* B, double* C, int blockSize, int N) {
    for (int i = 0; i < blockSize; ++i)
        for (int j = 0; j < blockSize; ++j)
            for (int k = 0; k < blockSize; ++k) {
                C[i * N + j] += A[i * N + k] * B[k * N + j];
            }
```

```
}
void Canon(double *A, double *B, double* C, int n, int q) {
    int blockSize = n / q;
    for (int i = 0; i < q; ++i) {
        for (int j = 0; j < q; ++j) {
            for (int k = 0; k < q; ++k) {
                MultMatrix(&A[(i*n + (j+i+k)%q)*blockSize], &B[(((i+j+k)%q)*n +
j)*blockSize],
                    &C[(i*n + j)*blockSize], blockSize, n);
            }
        }
    }
}
void Canon_Omp(double* pAMatrix, double* pBMatrix, double *pCMatrix, int q, int
Size) {
    omp_set_num_threads(q);
    int GridSize = static_cast <int>(sqrt(q));
    int BlockSize = Size / GridSize;
    #pragma omp parallel
    {
        int ThreadID = omp_get_thread_num();
        int RowIndex = ThreadID / GridSize;
        int ColIndex = ThreadID%GridSize;
        for (int iter = 0; iter < GridSize; iter++) {
            for (int i = RowIndex*BlockSize; i < (RowIndex + 1)*BlockSize; i++)
                for (int j = ColIndex*BlockSize; j < (ColIndex + 1)*BlockSize;
j++)
                    for (int k = iter*BlockSize; k < (iter + 1)*BlockSize; k++) {
                        pCMatrix[i*Size + j] += pAMatrix[i*Size + k] *
pBMatrix[k*Size + j];
            }
        }
    }
}
int main(int argc, char** argv) {
    std::cout << "Chislo potokov (q) - polniy kvadrat!" << std::endl;
    int size = 4;
    int q = 2;
    int proverka = 0;
    double *A, *B, *C, *SS, *S, *C1;
    double time_par = 0;
    double time_pos = 0;
    double time_izi = 0;
    std::cout << "omp_get_max_threads() = " << omp_get_max_threads() << std::endl;
    if (argc > 2) {
    size = atoi(argv[1]);
    q = atoi(argv[2]);
    }
```

```
A = CreateMatrix(size);
SS = CreateMatrix(size);
B = CreateMatrix(size);
C = CreateMatrix(size);
S = CreateMatrix(size);
C1 = CreateMatrix(size);
ClearMatrix(C, size);
ClearMatrix(SS, size);
ClearMatrix(S, size);
ClearMatrix(C1, size);
RandMatrix(A, B, size);
if (size < 5) {
PrintMatrix(A, size);
PrintMatrix(B, size);
}
time_izi = omp_get_wtime();
for (int i = 0; i < size; ++i)
    for (int j = 0; j < size; ++j)
        for (int k = 0; k < size; ++k) {
           SS[i * size + j] += A[i * size + k] * B[k * size + j];
        }
time_izi = omp_get_wtime() - time_izi;
time_pos = omp_get_wtime();
Canon(A, B, C, size, q);
time_pos = omp_get_wtime() - time_pos;
time_par = omp_get_wtime();
Canon_Omp(A, B, S, q, size);
time_par = omp_get_wtime() - time_par;
if (size < 5) {
PrintMatrix(SS, size);
PrintMatrix(C, size);
PrintMatrix(S, size);
}
for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++) {
        if (fabs(S[i * size + j] - C[i * size + j]) < 0.1)
            proverka++;
        else
            proverka = 0;
    }
std::cout << "Time izi version is " << time_izi << std::endl;
std::cout << "Time posl version is " << time_pos << std::endl;
std::cout << "Time parallel version is " << time_par << std::endl;
std::cout << "Boost is " << time_izi / time_par << std::endl;
if (proverka == 0)
    std::cout << "DANGER: Result not right" << std::endl;
else
    std::cout << "Bingo! Result right!" << std::endl;
```

```
    return 0;
}
```

TBB версия:

```
//
Copyright
2019
Kasmazyuk
Nikita
#include <tbb/tbb.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <string>
double* CreateMatrix(int N) {
    double *matrix = new double[N*N];
    return matrix;
}
void PrintMatrix(double* matrix, int N) {
for (int i = 0; i < N*N; i += N) {
        for (int j = 0; j< N; j++)
            std::cout << matrix[i + j] << " ";
        std::cout << std::endl;
    }
    std::cout << std::endl;
}
void RandMatrix(double* matrix1, double* matrix2, int N) {
    for (int i = 0; i < N*N; ++i) {
            matrix1[i] = (std::rand() % 10000) / 1000.0f;
            matrix2[i] = (std::rand() % 10000) / 1000.0f;
        }
}
void ClearMatrix(double *pCMatrix, int N) {
    for (int i = 0; i < N*N; ++i) {
            pCMatrix[i] = 0;
    }
}
void MultMatrix(double* pAMatrix, double* pBMatrix, double* pCMatrix, int
blockSize, int N) {
    for (int i = 0; i < blockSize; ++i)
        for (int j = 0; j < blockSize; ++j)
```

```
            for (int k = 0; k < blockSize; ++k) {
                pCMatrix[i * N + j] += pAMatrix[i * N + k] * pBMatrix[k * N + j];
            }
}
// void Canon_Omp(double* pAMatrix, double* pBMatrix, double *pCMatrix, int q, int
Size) {
//    omp_set_num_threads(q);
//    int GridSize = static_cast <int>(sqrt(q));
//    int BlockSize = Size / GridSize;
//    #pragma omp parallel
//    {
//        int ThreadID = omp_get_thread_num();
//        int RowIndex = ThreadID / GridSize;
//        int ColIndex = ThreadID%GridSize;
//        for (int iter = 0; iter < GridSize; iter++) {
//            for (int i = RowIndex*BlockSize; i < (RowIndex + 1)*BlockSize; i++)
//                for (int j = ColIndex*BlockSize; j < (ColIndex + 1)*BlockSize;
j++)
//                    for (int k = iter*BlockSize; k < (iter + 1)*BlockSize; k++)
{
//                        pCMatrix[i*Size + j] += pAMatrix[i*Size + k] *
pBMatrix[k*Size + j];
//            }
//        }
//    }
//}
void Canon(double *pAMatrix, double *pBMatrix, double *pCMatrix, int n, int q) {
    int blockSize = n / q;
    for (int i = 0; i < q; ++i) {
        for (int j = 0; j < q; ++j) {
            for (int k = 0; k < q; ++k) {
                MultMatrix(&pAMatrix[(i*n + (j+i+k)%q)*blockSize],
                    &pBMatrix[(((i+j+k)%q)*n + j)*blockSize],
                    &pCMatrix[(i*n + j)*blockSize], blockSize, n);
            }
        }
    }
}
class TBB {
    int ThreadNum_;
    double *pAMatrix_;
    double *pBMatrix_;
    double *pCMatrix_;
    int RowIndex_;
    int ColIndex_;
    int Size_;
 public:
    TBB(int ThreadNum, double *pAMatrix, double *pBMatrix, double *pCMatrix, int
```

```
RowIndex, int
    ColIndex, int Size) : ThreadNum_(ThreadNum), pAMatrix_(pAMatrix),
pBMatrix_(pBMatrix),
    pCMatrix_(pCMatrix), RowIndex_(RowIndex), ColIndex_(ColIndex), Size_(Size) {}
    void operator()() const {
        int GridSize = static_cast<int>(sqrt(static_cast<int>(ThreadNum_)));
        int BlockSize = Size_ / GridSize;
        for (int iter = 0; iter < GridSize; iter++) {
            for (int i = RowIndex_*BlockSize; i < (RowIndex_ + 1)*BlockSize; i++)
                for (int j = ColIndex_*BlockSize; j < (ColIndex_ + 1)*BlockSize;
j++)
                    for (int k = iter*BlockSize; k < (iter + 1)*BlockSize; k++) {
                        pCMatrix_[i*Size_ + j] += pAMatrix_[i*Size_ + k] *
                        pBMatrix_[k*Size_ + j];
            }
        }
    }
};
int main(int argc, char** argv) {
    std::cout << "Chislo potokov (q) - polniy kvadrat!" << std::endl;
    int size = 16;
    int q = 4;
    int proverka = 0;
    double *A, *B, *C, *SS, *S, *C1, *C3;
    if (argc > 2) {
    size = atoi(argv[1]);
    q = atoi(argv[2]);
    }
    A = CreateMatrix(size);
    SS = CreateMatrix(size);
    B = CreateMatrix(size);
    C = CreateMatrix(size);
    S = CreateMatrix(size);
    C1 = CreateMatrix(size);
    C3 = CreateMatrix(size);
    ClearMatrix(C, size);
    ClearMatrix(SS, size);
    ClearMatrix(S, size);
    ClearMatrix(C1, size);
    ClearMatrix(C3, size);
    RandMatrix(A, B, size);
    if (size < 17) {
    PrintMatrix(A, size);
    PrintMatrix(B, size);
    }
    tbb::tick_count time = tbb::tick_count::now();
    for (int i = 0; i < size; ++i)
        for (int j = 0; j < size; ++j)
```

```
            for (int k = 0; k < size; ++k) {
                SS[i * size + j] += A[i * size + k] * B[k * size + j];
            }
    double time_izi = (tbb::tick_count::now() - time).seconds();
    time = tbb::tick_count::now();
    Canon(A, B, C, size, q);
    double time_pos = (tbb::tick_count::now() - time).seconds();
    tbb::task_scheduler_init init(q);
    time = tbb::tick_count::now();
    tbb::task_group tg;
    for (int k = 0; k < sqrt(static_cast<int>(q)); k++)
    for (int j = 0; j < sqrt(static_cast<int>(q)); j++)
    tg.run(TBB(q, A, B, C3, k, j, size));
    tg.wait();
    double time_TBB = (tbb::tick_count::now() - time).seconds();
    if (size < 17) {
    PrintMatrix(SS, size);
    PrintMatrix(C, size);
    PrintMatrix(C3, size);
    }
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            if (fabs(SS[i * size + j] - C3[i * size + j]) < 0.1)
                proverka++;
            else
                proverka = 0;
        }
    std::cout << "Time izi version is " << time_izi << std::endl;
    std::cout << "Time posl version is " << time_pos << std::endl;
    std::cout << "Time parallel_TBB version is = " << time_TBB << std::endl;
    std::cout << "Boost is " << time_izi / time_TBB << std::endl;
    if (proverka == 0)
        std::cout << "DANGER: Result not right" << std::endl;
    else
        std::cout << "Bingo! Result right!" << std::endl;
    delete[] A;
    delete[] B;
    delete[] C;
    delete[] S;
    delete[] SS;
    delete[] C3;
    delete[] C1;
    return 0;
}
```