



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 4

Дисциплина Анализ алгоритмов

Тема Параллельное умножение матриц

Студент Куликов Д. А.

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Волкова Л. Л.

Москва — 2020 г.

Оглавление

Введение	3
1. Аналитическая часть	4
1.1 Описание умножения матриц по Винограду	4
1.2 Многопоточность	5
2. Конструкторская часть	7
2.1 Схемы алгоритмов	7
2.2 Распараллеливание алгоритма Винограда	10
2.2.1 Распараллеливание по группам строк	11
2.2.2 Распараллеливание по строке	11
3. Технологическая часть	13
3.1 Требования к программному обеспечению	13
3.2 Средства реализации	13
3.3 Листинг кода	14
3.4 Тестирование	21
4. Экспериментальная часть	22
4.1 Примеры работы	22
4.2 Постановка эксперимента по замеру времени	23
Заключение	26
Список литературы	27

Введение

Матрицей A размера $[m * n]$ называется прямоугольная таблица чисел, функций или алгебраических выражений, содержащая m строк и n столбцов. Числа m и n определяют размер матрицы [1].

Умножение матриц — одна из основных операций над матрицами. Матрица, получаемая в результате операции умножения, называется произведением матриц. Операция умножения двух матриц выполнима только в том случае, если число столбцов в первом сомножителе равно числу строк во втором; в этом случае говорят, что матрицы согласованы. В частности, умножение всегда выполнимо, если оба сомножителя — квадратные матрицы одного и того же порядка. Таким образом, из существования произведения AB вовсе не следует существование произведения BA . Алгоритмы умножения матриц активно применяются во всех областях, применяющих линейную алгебру. Примеры использования:

- компьютерная графика;
- физика;
- экономика и так далее.

Цель работы: изучение возможности параллельных вычислений и использование такого подхода на практике.

В данной лабораторной работе рассматривается алгоритм умножения матриц по Винограду и его параллельная версия, представленная в двух вариантах.

Задачи работы:

- рассмотрение алгоритма умножения матриц по Винограду;
- проведение сравнительного анализа алгоритма умножения матриц по Винограду и двух его параллельных версий;
- определение зависимости времени работы алгоритма от числа потоков исполнения и размера матриц.

1 | Аналитическая часть

Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй.

Пусть даны две прямоугольные матрицы А и В размерности m на n и n на k соответственно:

$$\begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \dots & \dots & \dots \\ a_{m,1} & \dots & a_{m,n} \end{bmatrix}$$

$$\begin{bmatrix} b_{1,1} & \dots & b_{1,k} \\ \dots & \dots & \dots \\ b_{n,1} & \dots & b_{n,k} \end{bmatrix}$$

В результате получим матрицу С размерности m на k:

$$\begin{bmatrix} c_{1,1} & \dots & c_{1,k} \\ \dots & \dots & \dots \\ c_{m,1} & \dots & c_{m,k} \end{bmatrix}$$

$c_{i,j} = \sum_{l=1}^n a_{i,l} \cdot b_{l,j}$ называется произведением матриц А и В.

1.1. Описание умножения матриц по Винограду

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно:

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_4 \cdot w_4$$

Это равенство можно переписать в виде:

$$V \cdot W = (v_1 + w_2) \cdot (v_2 + w_1) + (v_3 + w_4) \cdot (v_4 + w_3) - v_1 \cdot v_2 - v_3 \cdot v_4 - w_1 \cdot w_2 - w_3 \cdot w_4$$

Менее очевидно, что выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки

первой матрицы и для каждого столбца второй. Это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

В случае умножения матриц, строка и столбец которых представляют собой вектора нечетного размера, схема расчета элементов результирующей матрицы сохраняется. После чего, к каждому элементу c_{ij} результирующей матрицы прибавляется число $v_{im} \cdot w_{mj}$, где v_{im} - последний элемент i -той строки первой матрицы, w_{mj} - последний элемент j -того столбца второй матрицы.

1.2. Многопоточность

Поток выполнения — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени).

На одном процессоре многопоточность обычно происходит путём временного мультиплексирования (как и в случае многозадачности): процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

Потоки возникли в операционных системах как средство распараллеливания вычислений.

Параллельное выполнение нескольких работ в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещать набор нового текста с такими продолжительными по времени операциями, как переформатирование значительной части текста, печать документа или его сохранение на локальном или удаленном диске. Еще одним примером необходимости распараллеливания является сетевой сервер баз данных. В этом случае параллелизм желателен как для обслуживания различных запросов к базе данных, так и для более быстрого выполнения отдельного запроса за счет одновременного просмотра различных записей базы. Именно для этих целей современные ОС предлагают механизм многопоточной обработки (multithreading). Понятию «поток» соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками.

Создание потоков требует от ОС меньших накладных расходов, чем процессов. В отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и

ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им достаточно использовать общую память — один поток записывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (application programming interface, API) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows Thread API. Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер [2].

2 | Конструкторская часть

В этом разделе содержатся схема оптимизированного алгоритма умножения матриц по Винограду и описание распараллеливания этого алгоритма. На вход алгоритм принимает две матрицы. На выходе выдают матрицу — результат произведения двух этих матриц.

В рамках данной лабораторной работы были предложены следующие оптимизации алгоритма умножения по Винограду:

- 1) Замена $C[i][j] = C[i][j] + \dots$ на $C[i][j] += \dots$
- 2) Замена цикла по k от 0 до $N/2$ с шагом 1 на цикл от 0 до N с шагом 2, что убрало лишние умножения на 2
- 3) Элементы $MulH$ и $MulV$ сразу высчитываются отрицательными, что убирает 1 операцию отрицания в цикле
- 4) Замена $C[i][j] = \dots$ в цикле по k на буферную переменную, что убирает 2 операции индексирования во внутреннем цикле, но добавляет $C[i][j] = tmp$ во внешний цикл
- 5) Перенос проверки четности внутрь основного цикла, что ухудшило лучший случай, но улучшило худший
- 6) Вычисление условия четности и значения $N-1$, тем самым улучшая и лучший, и худший случаи

2.1. Схемы алгоритмов

На Рис. 2.1 - 2.3 представлена схема оптимизированного алгоритма Винограда умножения матриц.

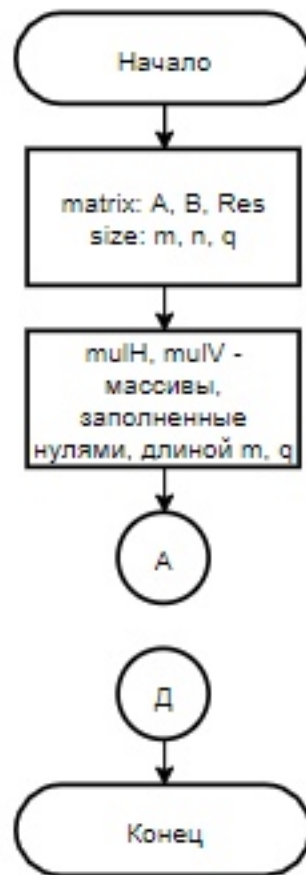


Рис. 2.1. Алгоритм умножения матриц по Винограду

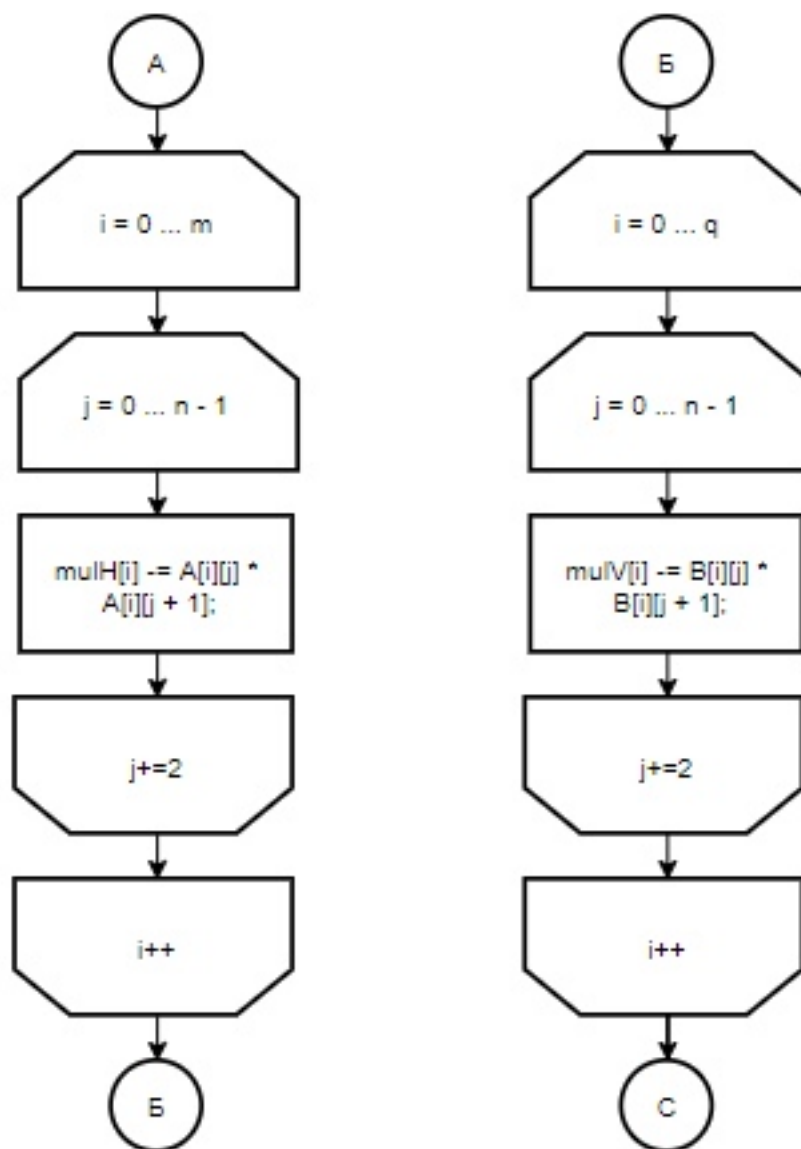


Рис. 2.2. Алгоритм умножения матриц по Винограду(продолжение 1)

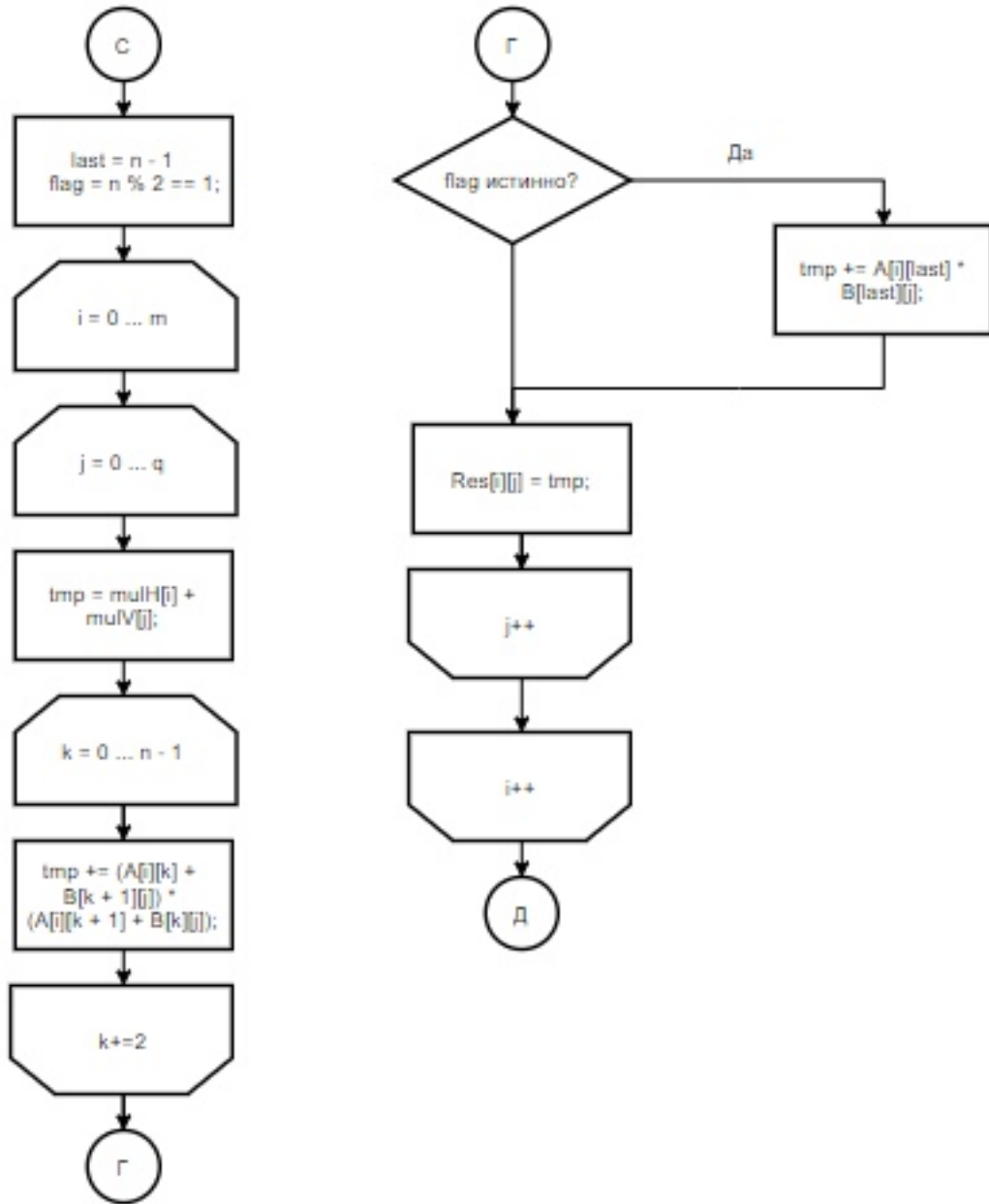


Рис. 2.3. Алгоритм умножения матриц по Винограду(продолжение 2)

2.2. Распараллеливание алгоритма Винограда

Распараллеливание программы должно ускорять время работы. Это достигается за счет реализации в узких участках (например в циклах с большим количеством независимых вычислений).

В предложенном алгоритме данными участками являются тройной цикл поиска результата(участок от С до Д), цикл вычисления сумм перемноженных пар строк первой матрицы и сумм перемноженных пар столбцов второй матрицы(участок от А до С).

Данные участки программы как раз предлагается распараллелить. Задача эффективно-го распараллеливания алгоритма может быть решена по-разному. Ниже представлены два варианта распараллеливание алгоритма умножения матриц по Винограду.

2.2.1. Распараллеливание по группам строк

Первая версия параллельного алгоритма заключается в том, чтобы разбить строки первой матрицы на блоки с одинаковым количеством строк в каждой, за исключением, когда количество строк не кратно количеству потоков, последнего. Таким образом мы получим схему, в которой каждый поток выполняет вычисления для каждой строки из своего блока, как это показано на Рис. 2.4 (количество потоков равно 3). На нем видно, что матрица А разбита на блоки по две строки в каждом, кроме последнего, так как количество строк оказалось нечетным. Внутри каждого потока идет перемножение каждой строки, на каждый столбец матрицы В, после выполнения работы каждым из потоков на выходе получается готовая матрица

А

5	10	8	7
6	0	4	8
3	2	1	7
8	1	4	7
9	1	4	4

В

8	9	6
4	8	3
0	1	5
3	5	4

Рис. 2.4. Версия параллельного алгоритма с разбивкой строк первой матрицы на блоки. В матрице А каждый цвет обозначает группу строк, принадлежащих отдельному потоку

При такой схеме нет проблем с обеспечением синхронизации, так как каждый блок независим от другого, и вычисления в одном ни как не влияют на вычисления в другом.

2.2.2. Распараллеливание по строке

Данный подход заключается в том, чтобы каждый поток занимался обработкой лишь одной строки. В отличие от предыдущего варианта, он будет сложнее в реализации, в силу необходимости обеспечения синхронизации между потоками. На вход подается определенное количество потоков, которое необязательно будет равно количеству строк в таблице, следовательно, возникает следующая проблема: как поток, после обработки строки, будет понимать какую следующую строчку ему обрабатывать? Для решения этого вопроса необходимо создать очередь, в которой будут лежать индексы тех строк, которые еще не были обработаны. Каждый поток будет обращаться к этой очереди и "брать себе задачу то есть получать индекс строки, которую нужно обработать, после чего повторит эти действия. Разделение первой матрицы 2 потоками представлено на Рис 2.5. Несмотря на то, что на нем каждая

строка закрашена одним из двух цветов(так как имеется в данном примере только 2 потока), эти строки не обрабатываются сразу. Сначала 1 и 2 поток захватят индекс 1 и 2 строки соответственно, произведут над ними вычисления, а затем перейдут к строке 3 и 4 и т.д.

A

5	10	8	7
6	0	4	8
3	2	1	7
8	1	4	7
9	1	4	4

B

8	9	6
4	8	3
0	1	5
3	5	4

Рис. 2.5. Версия параллельного алгоритма с построчной обработкой матрицы каждым потоком. В матрице A каждый цвет обозначает строку, которая будет обработана конкретным потоком

Проблема заключается в захвате индекса следующей строки из очереди. Потоки работают внутри одного процесса, а следовательно разделяют его общую память. Если один поток получит значение из очереди, то другой может просто "не увидеть"этого и обработать эту же строку, что приведет к неверному ответу. Для того чтобы синхронизировать потоки, необходимо воспользоваться мьютексом. Мьютекс - примитив синхронизации, который позволяет захватить владение каким-либо объектом определенным потоком, в таком случае другие потоки не смогут каким-либо образом повлиять на данный объект, пока данный поток не вернет его в общее пользование.[3]

3 | Технологическая часть

В данном разделе будут рассмотрены требования к программному обеспечению, средства реализации, представлен листинг кода и описание тестирования.

3.1. Требования к программному обеспечению

Требования к вводу: на вход подаются две матрицы, размеры которых $m \times n$ и $n \times q$ соответственно.

Требования к программе:

- 1) на выходе необходимо получить матрицу, которая является результатом умножения двух матриц;
- 2) требуется замерить время работы каждого из алгоритмов.

3.2. Средства реализации

В качестве языка программирования был выбран C++ т.к. я знаком с данным языком, у него есть уникальный баланс между возможностями объектно-ориентированного программирования и производительностью. Он одновременно позволяет писать высокоуровневый абстрактный код, который при этом работает со скоростью близкой к машинному коду.

Среда разработки — Visual Studio, которая предоставляет умную проверку кода, быстрое выявление ошибок и оперативное исправление, вкуче с автоматическим рефакторингом кода, и богатыми возможностями в навигации.

Время работы алгоритмов было замерено с помощью функции `steady_clock()` из библиотеки `chrono` [4].

Для тестирования использовался компьютер на базе процессора Intel(R) Core(TM) i5-4200U, 2 ядра, 4 логических процессоров.

Многопоточное программирование было реализовано с помощью библиотеки `thread` [5].

3.3. Листинг кода

В Листинге 3.1 показана реализация оптимизированного алгоритма умножения матриц по Винограду.

Листинг 3.1. Оптимизированный алгоритм умножения матриц по Винограду

```
1 int vinogradOPT(const vector<vector<int>>& A, const vector<vector<int>>& B,  
2 const int m, const int n, const int q, vector<vector<int>>& C)  
3 {  
4     vector<int> mulH(m, 0);  
5     for (int i = 0; i < m; i++)  
6     {  
7         for (int j = 0; j < n - 1; j += 2)  
8         {  
9             mulH[i] -= A[i][j] * A[i][j + 1];  
10        }  
11    }  
12  
13    vector<int> mulV(q, 0);  
14    for (int i = 0; i < q; i++)  
15    {  
16        for (int j = 0; j < n - 1; j += 2)  
17        {  
18            mulV[i] -= B[j][i] * B[j + 1][i];  
19        }  
20    }  
21    int last = n - 1;  
22    bool flag = n % 2 == 1;  
23    C = vector<vector<int>>(m, vector<int>(q, 0));  
24    for (int i = 0; i < m; i++)  
25    {  
26        for (int j = 0; j < q; j++)  
27        {  
28            int tmp = mulH[i] + mulV[j];  
29            for (int k = 0; k < n - 1; k += 2)  
30            {  
31                tmp += (A[i][k] + B[k + 1][j]) * (A[i][k + 1] + B[k][j]);  
32            }  
33            if (flag)  
34            {  
35                tmp += A[i][last] * B[last][j];  
36            }  
37            C[i][j] = tmp;  
38        }  
39    }  
40    return OK;  
41 }
```

В Листингах 3.2 - 3.5 показана реализация распараллеленного по группам строк оптимизированного алгоритма умножения матриц по Винограду.

Листинг 3.2. Многопоточного оптимизированный алгоритм умножения матриц по Винограду

```
1 int threadedVinogradOPT1(const vector<vector<int>>& A, const vector<vector<
  int>>& B,
2 const int m, const int n, const int q, vector<vector<int>>& C,
3 const int& nThreads)
4 {
5     vector<thread> threads;
6     vector<int> mulH(m, 0);
7     double start = 0;
8     double del = m / static_cast<double>(nThreads);
9     for (int i = 0; i < nThreads; i++)
10    {
11        threads.push_back(thread(computeMulH, ref(mulH), A, round(start),
12                                round(start + del)));
13        start += del;
14    }
15    for (auto& thread : threads)
16    {
17        thread.join();
18    }
19    start = 0;
20    del = q / static_cast<double>(nThreads);
21    vector<int> mulV(q, 0);
22    for (int i = 0; i < nThreads; i++)
23    {
24        threads[i] = thread(computeMulV, ref(mulV), B, round(start),
25                            round(start + del));
26        start += del;
27    }
28    for (auto& thread : threads)
29    {
30        thread.join();
31    }
32
33    C = vector<vector<int>>(m, vector<int>(q, 0));
34    start = 0;
35    del = m / static_cast<double>(nThreads);
36    for (int i = 0; i < nThreads; i++)
37    {
38        threads[i] = thread(computeResult, ref(C), A, B,
39                            mulH, mulV, round(start), round(start + del));
40        start += del;
41    }
42    for (auto& thread : threads)
43    {
```

```

44     thread.join();
45 }
46 return OK;
47 }

```

Листинг 3.3. Функция вычисления сумм строк первой матрицы

```

1 void computeMulH(vector<int>& mulH, vector<vector<int>> A, int startRow, int
  endRow)
2 {
3     int n = A[0].size();
4     for (int i = startRow; i < endRow; i++)
5     {
6         for (int j = 0; j < n - 1; j += 2)
7         {
8             mulH[i] -= A[i][j] * A[i][j + 1];
9         }
10    }
11 }

```

Листинг 3.4. Функция вычисления сумм столбцов второй матрицы

```

1 void computeMulV(vector<int>& mulV, vector<vector<int>> B, int startCol, int
  endCol)
2 {
3     int n = B.size();
4     for (int i = startCol; i < endCol; i++)
5     {
6         for (int j = 0; j < n - 1; j += 2)
7         {
8             mulV[i] -= B[j][i] * B[j + 1][i];
9         }
10    }
11 }

```


Листинг 3.5. Функция вычисления результирующей матрицы

```
1 void computeResult1(vector<vector<int>>& C, vector<vector<int>> A,  
2 vector<vector<int>> B, vector<int> mulH,  
3 vector<int> mulV, int startRow, int endRow)  
4 {  
5     int n = B.size();  
6     int q = B[0].size();  
7     int last = n - 1;  
8     bool flag = n % 2 == 1;  
9     for (int i = startRow; i < endRow; i++)  
10    {  
11        for (int j = 0; j < q; j++)  
12        {  
13            int tmp = mulH[i] + mulV[j];  
14            for (int k = 0; k < n - 1; k += 2)  
15            {  
16                tmp += (A[i][k] + B[k + 1][j]) * (A[i][k + 1] + B[k][j]);  
17            }  
18            if (flag)  
19            {  
20                tmp += A[i][last] * B[last][j];  
21            }  
22            C[i][j] = tmp;  
23        }  
24    }  
25 }
```

В Листингах 3.6 - 3.7 показана реализация распараллеленного по строке оптимизированного алгоритма умножения матриц по Винограду.

Листинг 3.6. Функция вычисления результирующей матрицы

```
1 int threadedVinogradOPT2(const vector<vector<int>>& A, const vector<vector<
  int>>& B,
2 const int m, const int n, const int q, vector<vector<int>>& C,
3 const int& nThreads)
4 {
5     vector<thread> threads;
6     vector<int> mulH(m, 0);
7     double start = 0;
8     double del = m / static_cast<double>(nThreads);
9     for (int i = 0; i < nThreads; i++)
10    {
11        threads.push_back(thread(computeMulH, ref(mulH), A, round(start),
12                                round(start + del)));
13        start += del;
14    }
15    for (auto& thread : threads)
16    {
17        thread.join();
18    }
19    start = 0;
20    del = q / static_cast<double>(nThreads);
21    vector<int> mulV(q, 0);
22    for (int i = 0; i < nThreads; i++)
23    {
24        threads[i] = thread(computeMulV, ref(mulV), B, round(start),
25                            round(start + del));
26        start += del;
27    }
28    for (auto& thread : threads)
29    {
30        thread.join();
31    }
32
33    C = vector<vector<int>>(m, vector<int>(q, 0));
34    queue<int> que;
35
36    for (int i = 0; i < m; i++)
37        que.push(i);
38    for (int i = 0; i < nThreads; i++)
39    {
40        threads[i] = thread(computeResult1, ref(C), A, B,
41                            mulH, mulV, ref(que));
42    }
43    for (auto& thread : threads)
```

```
44 {  
45     thread.join();  
46 }  
47 return OK;  
48 }
```

Листинг 3.7. Функция вычисления результирующей матрицы

```

1 void computeResult1(vector<vector<int>>& C, vector<vector<int>> A,
2 vector<vector<int>> B, vector<int> mulH,
3 vector<int> mulV, queue<int> &que)
4 {
5     int n = B.size();
6     int q = B[0].size();
7     int last = n - 1;
8     bool flag = n % 2 == 1;
9     mutex m;
10    while (true)
11    {
12        m.lock();
13        if (que.empty())
14        {
15            m.unlock();
16            break;
17        }
18        int i = que.front();
19        que.pop();
20        m.unlock();
21        for (int j = 0; j < q; j++)
22        {
23            int tmp = mulH[i] + mulV[j];
24            for (int k = 0; k < n - 1; k += 2)
25            {
26                tmp += (A[i][k] + B[k + 1][j]) * (A[i][k + 1] + B[k][j]);
27            }
28            if (flag)
29            {
30                tmp += A[i][last] * B[last][j];
31            }
32            C[i][j] = tmp;
33        }
34    }
35 }

```

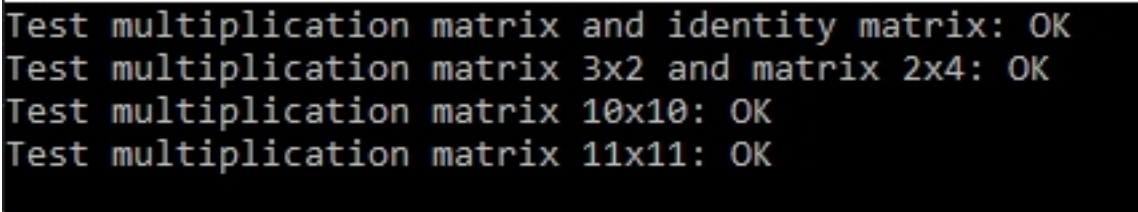
3.4. Тестирование

Реализовано функциональное тестирование отдельным файлом test.cpp. Полученные результаты функций сравниваются с контрольными значениями.

Тестирование происходит по следующим данным:

- 1) проверка работы умножения на единичную матрицу;
- 2) проверка работы умножения матрицы 3x2 на матрицу 2x4;
- 3) проверка работы умножения матриц 10x10;
- 4) проверка работы умножения матриц 11x11;

Программа успешно прошла все тестовые случаи, см. Рис. 3.1.



```
Test multiplication matrix and identity matrix: OK
Test multiplication matrix 3x2 and matrix 2x4: OK
Test multiplication matrix 10x10: OK
Test multiplication matrix 11x11: OK
```

Рис. 3.1. Тестирование программы

4 | Экспериментальная часть

В данном разделе приведены примеры работы программы и сравнительный анализ алгоритмов на основе экспериментальных данных.

4.1. Примеры работы

На Рис. 4.1 - 4.2 приведены примеры работы программы.

```
Matrix A
23 81
26 62

Matrix B
88 36
66 90

Result matrix OPT Vinograd:
7370 8118
6380 6516

Result matrix OPT Vinograd thread 2 schema 1:
7370 8118
6380 6516

Result matrix OPT Vinograd thread 2 schema 2:
7370 8118
6380 6516
```

Рис. 4.1. Пример работы программы 2 потока

```

Matrix A
64 42
47 4

Matrix B
22 39
31 13

Result matrix OPT Vinograd:
2710 3042
1158 1885

Result matrix OPT Vinograd thread 4 schema 1:
2710 3042
1158 1885

Result matrix OPT Vinograd thread 4 schema 2:
2710 3042
1158 1885

```

Рис. 4.2. Пример работы программы 4 потока

4.2. Постановка эксперимента по замеру времени

Для произведения замеров времени выполнения реализации алгоритмов будет использована формула:

$$t = \frac{T}{N} \quad (4.1)$$

где t — среднее время выполнения алгоритма, N — количество замеров, T — время выполнения N замеров. Неоднократное измерение времени необходимо для получения более точного результата.

Все эксперименты производятся на квадратных матрицах размером от 100×100 до 1000×1000 с шагом 100. Время измеряется в микросекундах.

На рисунке 4.1 изображены графики зависимости времени выполнения программы от размера входных квадратных матриц и количество потоков для 1 параллельной версии классического алгоритма :

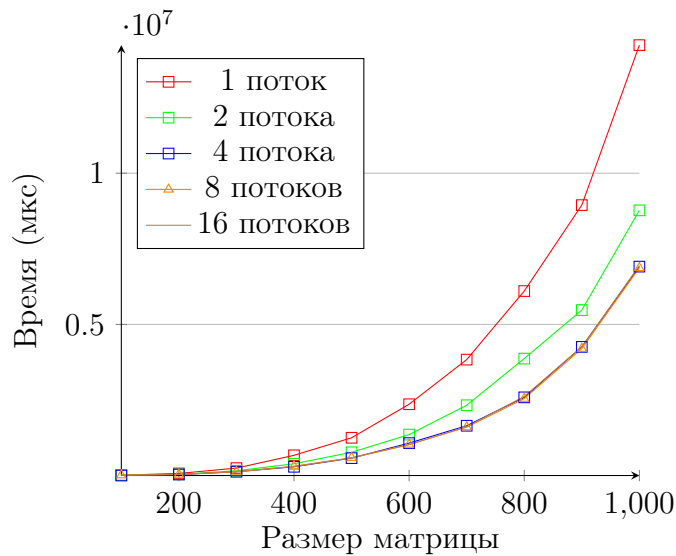


Рис. 4.1: Сравнение времени работы первой параллельной версии алгоритма при различном количестве потоков

Как видно из рисунка, начиная с 4 потоков увеличение числа потоков не дало сильного прироста в скорости, это связано с тем, что число потоков, которые работают параллельно равно числу логических процессоров, которых на экспериментальном процессоре, как было сказано в Технологической части 4.

На рисунке 4.2 изображены графики зависимости времени выполнения программы от размера входных квадратных матриц и количество потоков для 2 параллельной версии классического алгоритма :

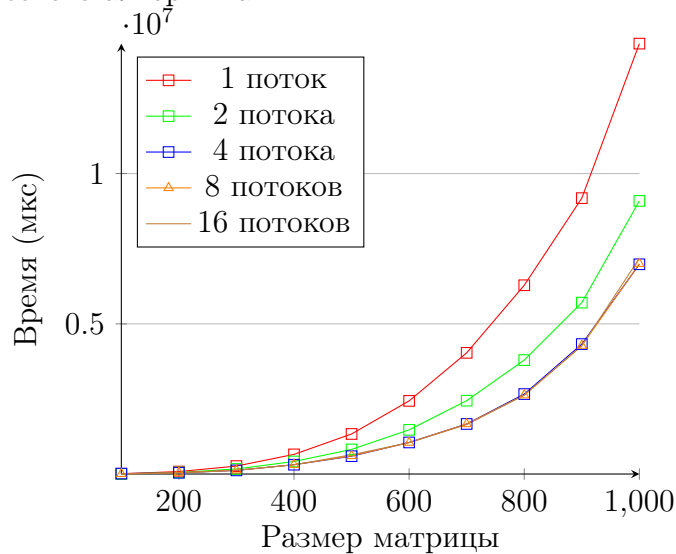


Рис. 4.2: Сравнение времени работы второй параллельной версии алгоритма при различном количестве потоков

Видно, что ситуация повторилась как и в случае с 1 версией.

Так как максимальную производительность в обоих случаях удастся достичь при 4 потоках, то классический метод стоит сравнивать с параллельными версиями именно при нем.

На рисунке 4.3 видно, что обе параллельные версии работают приблизительно одинаково и существенно выигрывают по времени в сравнении с классическим алгоритмом, особенно при увеличении размерности матрицы.

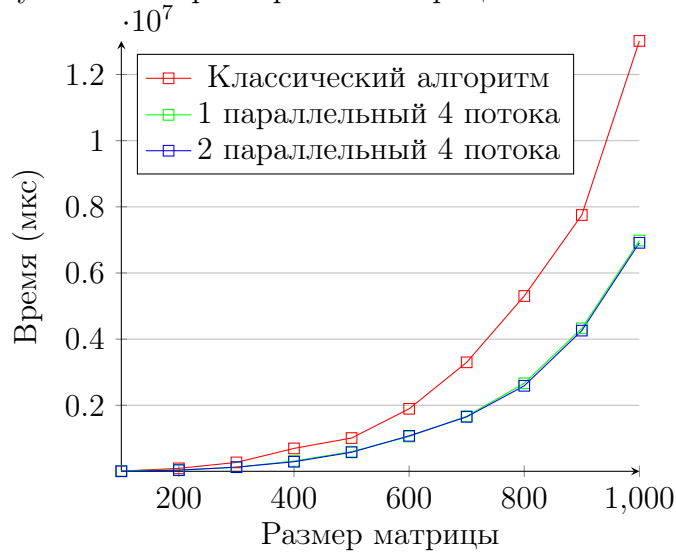


Рис. 4.3: Сравнение времени работы классического алгоритма и двух его параллельных версий при количестве потоков равных 4

Вывод

По результатам исследования получилось, что обе параллельные версии алгоритма работают приблизительно за равное время, но каждая из них быстрее классического алгоритма, причем с увеличением размерности матрицы выигрыш становится все более ощутимым. Также установлено, что увеличение количества потоков имеет смысл, пока не будет достигнуто число, равное количеству логических процессоров в системе, причем самой быстрой версией параллельного алгоритма (любой из его версий) является та, где число потоков равно числу логических процессоров.

Заключение

В ходе лабораторной работы были изучены возможности параллельных вычислений и использованы на практике. Был реализован алгоритм умножения матриц по Винограду с помощью параллельных вычислений. Было произведено сравнение работы обычного алгоритма Винограда и параллельной реализации при увеличении количества потоков. Экспериментально было установлено, что параллельные версии быстрее классического алгоритма, причем чем больше размерность матрицы, тем больше выигрыш. Было установлено, что увеличение потоков имеет смысл, пока не будет достигнуто число логических процессоров в системе, причем максимальная скорость работы достигается именно при нем.

Цель работы достигнута. Получены практические навыки использования параллельных вычислений, а также проведена исследовательская работа по временной эффективности такого подхода.

Список литературы

1. Курош А. Г. Курс высшей алгебры. — 9-е изд. — М.: Наука, 1968. — 432 с
2. Константин Баркалов, Владимир Воеводин, Виктор Гергель. Intel Parallel Programming [электронный ресурс]. Режим доступа: <https://www.intuit.ru/studies/courses/4447/983/lecture/14925>, свободный (Дата обращения: 15.10.20)
3. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб: БХВ-Петербург, 2002. — 608 с.
4. Официальный сайт Microsoft, документация [электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/cpp/standard-library/chrono?view=vs-2017>, свободный (Дата обращения: 15.10.20)
5. Официальный сайт Microsoft, документация [электронный ресурс]. Режим доступа: <https://docs.microsoft.com/ru-ru/cpp/standard-library/thread-class?view=vs-2019>, свободный (Дата обращения: 15.10.20)