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
по курсу "Анализ алгоритмов"

Тема Распараллеливание алгоритма умножения матриц Виноградова

Студент Мишина Е.В.

Группа ИУ7-54Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 г.

Оглавление

Введение

В огромном количестве областей научной и технической сферы деятельности человека при различных математических расчетах используют такую операцию как умножение матриц. Это довольно трудоемкий процесс даже при небольших размерах матриц, так как требуется большое количество операций умножения и сложения различных чисел. По этой причине человек озадачен проблемой оптимизации умножения матриц и ускорения процесса вычисления.
Таким образом, эффективное умножение матриц по времени и затратам ресурсов является актуальной проблемой для науки и техники.

1. Аналитическая часть

1.1 Задачи

Цель лабораторной работы - изучение двух реализаций алгоритма умножения матриц Винограда: последовательной и параллельной.
Для того чтобы добиться этой цели, были поставлены следующие задачи:

- изучить алгоритм Винограда;
- изучить методы параллельного программирования;
- применить знания программирования для реализации указанного алгоритма;
- выполнить сравнительный анализ последовательной реализации и параллельной реализации алгоритма Винограда при различном числе потоков;
- экспериментально подтвердить различия в эффективности по времени реализаций алгоритма Винограда.

1.2 Описание алгоритмов

Матрицей называют математический объект, эквивалентный двумерному массиву. Матрица является таблицей, на пересечении строк и столбцов находятся элементы матрицы. Количество строк и столбцов является размерностью матрицы.
Пусть даны две прямоугольные матрицы A и B размерности $m \times n$, $n \times q$ соответственно:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1q} \\ b_{21} & b_{22} & \dots & b_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nq} \end{bmatrix}$$

Тогда произведением матриц A и B называется матрица C размерностью $m \times q$ [1]

$$C = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mq} \end{bmatrix}, \text{ [?]} \tag{1.1}$$

в которой:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad \left(\overline{i = 1 \dots m};\ \overline{j = 1 \dots q}\right).$$

1.2.1 Алгоритм Винограда

Исходя из равенства 1.1, видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее. [2] Рассмотрим два вектора U и V:

$$U = A_i = (u_1, u_2, \dots, u_n), \tag{1.2}$$

где $U = A_i$ – i-ая строка матрицы A,
$u_k = a_{ik}, \overline{k = 1 \dots n}$ – элемент i-ой строки k-ого столбца матрицы A.

$$V = B_j = (v_1, v_2, \dots, v_n), \tag{1.3}$$

где $V = B_j$ – j-ый столбец матрицы B,
$v_k = b_{kj}, \overline{k = 1 \dots n}$ – элемент k-ой строки j-ого столбца матрицы B.

По определению их скалярное произведение равно:

$$U \cdot V = u_1 v_1 + u_2 v_2 + u_3 v_3 + u_4 v_4. \tag{1.4}$$

Равенство 1.4 можно переписать в виде:

$$U \cdot V = (u_1 + v_2)(u_2 + v_1) + (u_3 + v_4)(u_4 + v_3) - u_1 u_2 - u_3 u_4 - v_1 v_2 - v_3 v_4. \tag{1.5}$$

В равенстве 1.4 насчитывается 4 операции умножения и 3 операции сложения, в равенстве 1.5 насчитывается 6 операций умножения и 9 операций сложения. Однако выражение $-u_1 u_2 - u_3 u_4$ используются повторно при умножении i-ой строки матрицы A на каждый из столбцов матрицы B, а выражение $-v_1 v_2 - v_3 v_4$ - при умножении j-ого столбца матрицы B на строки матрицы A. Таким образом, данные выражения можно вычислить предварительно для каждых строк и столбцов матриц для сокращения повторных вычислений. В результате повторно будут выполняться лишь 2 операции умножения и 7 операций сложения (2 операции нужны для добавления предварительно посчитанных произведений).

1.2.2 Параллельная реализация алгоритма Винограда

Трудоёмкость алгоритма Винограда для матриц размеров $M \times N$ и $N \times Q$ имеет сложность $O(MNQ)$. Для улучшения работы алгоритма необходимо выполнить распараллеливание той части алгоритма, которая задаёт данную сложность - часть, содержащая 3 вложенных цикла по размерам матриц. Вычисление результата для каждой строки результирующей

матрицы не зависит от результата умножения других строк. Таким образом, можно выполнить распараллеливание той части кода, где выполняется вычисление строки. Вычисление отдельных групп строк результирующей матрицы будет отводиться под отдельный поток.

1.3 Параллельное программирование

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (symmetric multiprocessors, SMP).

Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время многоядерные процессоры, в которых каждое ядро представляет практически независимо функциони рующее вычислительное устройство.

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки, расширенные некоторым набором операторов для параллельных вычислений.

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 [3].

2. Конструкторская часть

2.1 Схема алгоритма

На рисунке 2.1 представлена схема оптимизированного алгоритма Винограда.

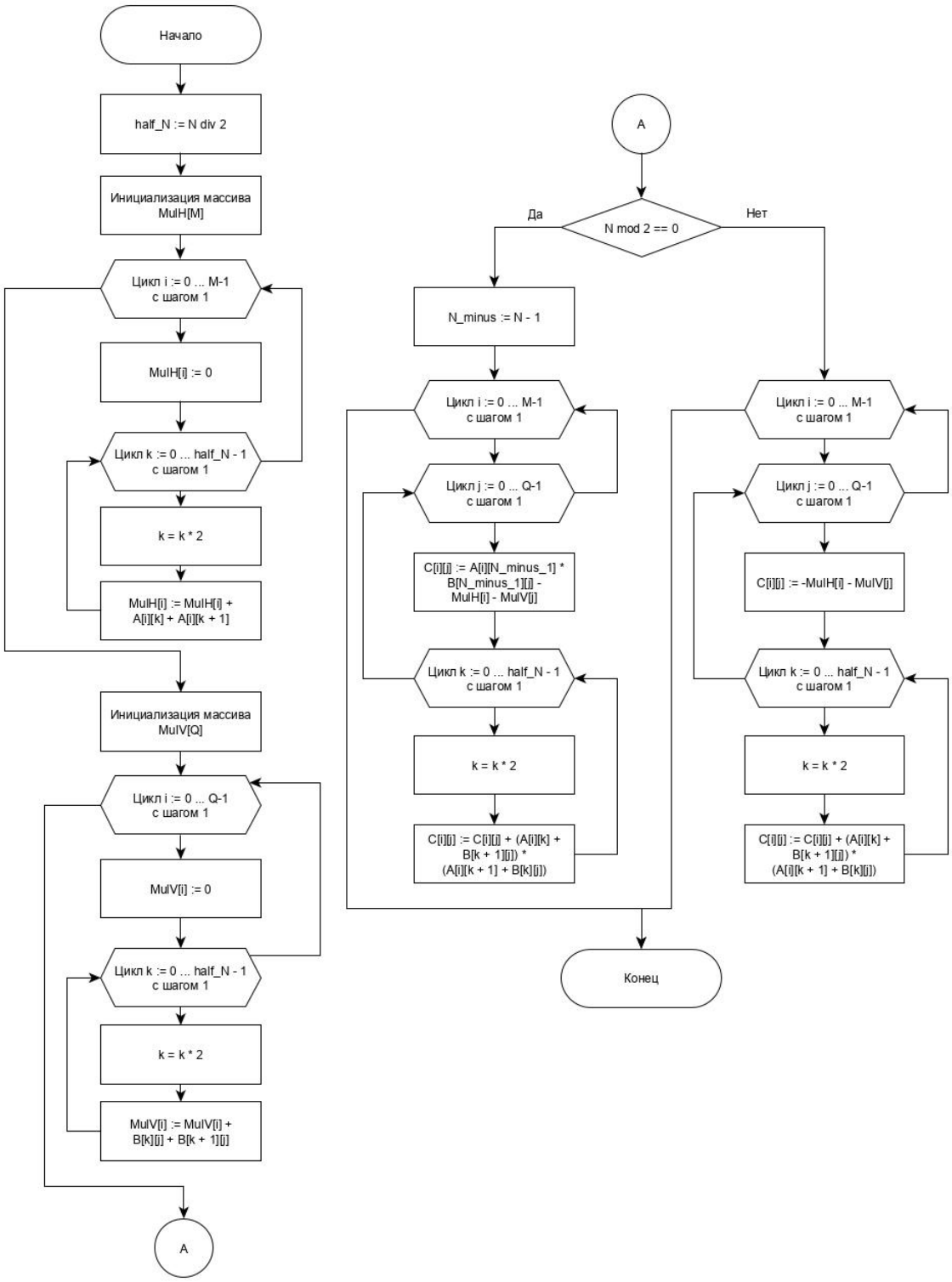

Рис. 2.1: Алгоритм Винограда

2.2 Распараллеливание алгоритма

Распараллеливание алгоритма должно выполняться над теми данными, которые могут быть использованы только одним потоком. Это условие необходимо для обеспечения защиты данных от случайных изменений разными потоками одновременно. В данном алгоритме распараллеливание происходит в циклах, которые обозначены на схеме алгоритма участком A. В данных циклах каждая строка матрицы вычисляется независимо от других строк. Таким образом, многопоточность можно реализовать именно на данных циклах: каждый поток отвечает за определённые строки матрицы.

Также можно выполнить параллельную инициализацию векторов MulV и MulH, так как инициализация их значений не зависят друг от друга.

3. Технологическая часть

3.1 Требования к программному обеспечению

На вход подаются размеры двух матриц, затем число потоков для распараллеливания алгоритма. Матрицы генерируются случайным образом. На выход программа выдаёт сгенерированные исходные матрицы и результирующую матрицу.

3.2 Средства реализации

Для реализации программы был использован язык C++ [4]. Для замера процессорного времени была использована функция rdtsc() из библиотеки stdrin.h. Потоки реализовывались с использованием библиотеки pthreads.h.

3.3 Реализации алгоритмов

В листинге 3.1 представлен код последовательной реализации алгоритма Винограда.

Листинг 3.1: Последовательная реализация алгоритма Винограда

```
1  void multiply_vinograd_nothread(Matrix &A, Matrix &B)
2  {
3    const unsigned M = A.get_rows();
4    const unsigned N = A.get_cols();
5    const unsigned Q = B.get_cols();
6
7    Matrix C(M, Q);
8
9    unsigned half_N = N >> 1;
10
11   int *MulH = new int[M];
12   for (unsigned i = 0; i < M; i++)
13   {
14     MulH[i] = 0;
15     for (unsigned k = 0; k < half_N; k++)
16     {
17       k <<= 1;
18       MulH[i] += A[i][k] * A[i][k + 1];
19     }
20   }
21
22   int *MulV = new int[Q];
23   for (unsigned i = 0; i < Q; i++)
24   {
```

```
25      MulV[i] = 0;
26      for (unsigned k = 0; k < half_N; k++)
27      {
28        k <<= 1;
29        MulV[i] += B[k][i] * B[k + 1][i];
30      }
31    }
32
33    if (N % 2)
34    {
35      unsigned N_minus_1 = N - 1;
36      for (unsigned i = 0; i < M; i++)
37        for (unsigned j = 0; j < Q; j++)
38        {
39          C[i][j] = A[i][N_minus_1] * B[N_minus_1][j] - MulH[i] - MulV[j];
40          for (unsigned k = 0; k < half_N; k++)
41          {
42            k <<= 1;
43            C[i][j] += (A[i][k] + B[k + 1][j]) * (A[i][k + 1] + B[k][j]);
44          }
45        }
46    }
47    else
48    {
49      for (unsigned i = 0; i < M; i++)
50        for (unsigned j = 0; j < Q; j++)
51        {
52          C[i][j] = -MulH[i] - MulV[j];
53          for (unsigned k = 0; k < half_N; k++)
54          {
55            k <<= 1;
56            C[i][j] += (A[i][k] + B[k + 1][j]) * (A[i][k + 1] + B[k][j]);
57          }
58        }
59    }
60
61    delete [] MulH;
62    delete [] MulV;
63  }
```

В листинге 3.4 представлена параллельная реализация алгоритма Винограда. Функции потоков вычисления строк при чётных и нечётных размерах матриц представлены в листингах 3.6 и 3.7 соответственно. Функции потоков инициализации рабочих векторов описаны в листинге 3.5. Структура аргументов для функции потока описана в листинге 3.3. Структура класса "Матрица"описана в листинге 3.2.

Листинг 3.2: Класс Матрица

```
1  class Matrix;
2
3  class Array
4  {
5  public:
6    Array();
```

```
7   explicit Array(unsigned size);
8   Array(const Array& other);
9
10  ~Array();
11
12  void alloc(unsigned size);
13
14  void read(std::istream& stream);
15  void write(std::ostream& stream);
16
17  int& operator[](unsigned i);
18
19  friend Matrix;
20
21 private:
22  int *ptr;
23  unsigned _size;
24 };
25
26 class Matrix
27 {
28 public:
29  Matrix(unsigned rows, unsigned cols);
30  explicit Matrix(std::istream& stream);
31  Matrix(const Matrix &other);
32  ~Matrix();
33
34  void read(std::istream& stream);
35  void write(std::ostream& stream);
36
37  void randomize(int min, int max);
38
39  Matrix& operator=(Matrix &other);
40
41  bool operator==(const Matrix &other);
42  bool operator!=(const Matrix &other);
43
44  unsigned get_rows();
45  unsigned get_cols();
46  Array& operator[] (unsigned i);
47
48 private:
49  Array *ptr;
50  unsigned _rows;
51  unsigned _cols;
52 };
```

Листинг 3.3: Структура аргументов

```
1 typedef struct MultArgs_t
2 {
3   Matrix &A;
4   Matrix &B;
5   Matrix &C;
```

```
  int *MulH;
  int *MulV;
  unsigned half_N;
  unsigned N_minus_1;

  unsigned start_i;
  unsigned end_i;
} MultArgs;

typedef struct
{
  int **MVector;
  Matrix &A;
  unsigned half_N;
} InitVectorArgs;
```

Листинг 3.4: Параллельная реализация алгоритма Винограда

```
Matrix multiply_vinograd_thread(Matrix &A, Matrix &B, unsigned thread_amount)
{
  int status, status_addr;
  pthread_t thr_MulH, thr_MulV;
  pthread_t *threads = new pthread_t[thread_amount];

  const unsigned M = A.get_rows();
  const unsigned N = A.get_cols();
  const unsigned Q = B.get_cols();

  Matrix C(M, Q);

  unsigned half_N = N >> 1;

  int *MulH = nullptr, *MulV = nullptr;
  InitVectorArgs argsH = { &MulH, A, half_N };
  InitVectorArgs argsV = { &MulV, B, half_N };

  status = pthread_create(&thr_MulH, NULL, init_MultH, (void*) &argsH);
  if (status)
  {
    printf("Can't create thread for MulH, status = %d\n", status);
    exit(ERROR_CREATE_THREAD);
  }
  status = pthread_create(&thr_MulV, NULL, init_MultV, (void*) &argsV);
  if (status)
  {
    printf("Can't create thread for MulV, status = %d\n", status);
    exit(ERROR_CREATE_THREAD);
  }

  status = pthread_join(thr_MulH, (void**)&status_addr);
  if (status)
  {
    printf("Can't join thread thr_MulH, status = %d\n", status);
    exit(ERROR_JOIN_THREAD);
```

```
37    }
38    status = pthread_join(thr_MulV, (void**)&status_addr);
39    if (status)
40    {
41      printf("Can't join thread thr_MulV, status = %d\n", status);
42      exit(ERROR_JOIN_THREAD);
43    }
44
45    float step_i = M / float(thread_amount), value_i = step_i;
46    unsigned start_i = 0, end_i = unsigned(value_i);
47
48    std::vector<MultArgs> args;
49    for (unsigned i = 0; i < thread_amount; i++)
50    {
51      args.push_back({A, B, C, MulH, MulV, half_N, N - 1, start_i, end_i});
52      start_i = end_i;
53      value_i += step_i;
54      end_i = unsigned(value_i);
55    }
56
57    void* (*thread_func)(void*) = multiply_even;
58    if (N % 2)
59      thread_func = multiply_odd;
60
61    for (unsigned i = 0; i < thread_amount; i++)
62    {
63      status = pthread_create(&(threads[i]), NULL, thread_func,
64                  (void*) &args[i]);
65      if (status)
66      {
67        printf("Can't create thread %u, status = %d\n", i, status);
68        exit(ERROR_CREATE_THREAD);
69      }
70    }
71
72    for (unsigned i = 0; i < thread_amount; i++)
73    {
74      status = pthread_join(threads[i], (void**)&status_addr);
75      if (status)
76      {
77        printf("Can't join thread %u, status = %d\n", i, status);
78        exit(ERROR_JOIN_THREAD);
79      }
80    }
81
82    delete [] MulH;
83    delete [] MulV;
84    delete [] threads;
85
86    return C;
87  }
```

Листинг 3.5: Функции потоков инициализации векторов

```
void *init_MultH(void *_args)
{
  InitVectorArgs *args = (InitVectorArgs*) _args;

  const unsigned M = args->A.get_rows();
  *(args->MVector) = new int[M];
  for (unsigned i = 0; i < M; i++)
  {
    (*(args->MVector))[i] = 0;
    for (unsigned k = 0; k < args->half_N; k++)
    {
      k <<= 1;
      (*(args->MVector))[i] += args->A[i][k] * args->A[i][k + 1];
    }
  }
  return NULL;
}

void *init_MultV(void *_args)
{
  InitVectorArgs *args = (InitVectorArgs*) _args;

  const unsigned Q = args->A.get_cols();
  *(args->MVector) = new int[Q];
  for (unsigned j = 0; j < Q; j++)
  {
    (*(args->MVector))[j] = 0;
    for (unsigned k = 0; k < args->half_N; k++)
    {
      k <<= 1;
      (*(args->MVector))[j] += args->A[k][j] * args->A[k + 1][j];
    }
  }
  return NULL;
}
```

Листинг 3.6: Функция потока вычисления строки при нечётных размерах матрицы

```
void *multiply_odd(void *_args)
{
  MultArgs *args = (MultArgs*) _args;

  const unsigned Q = args->B.get_cols();

  for (unsigned i = args->start_i; i < args->end_i; i++)
    for (unsigned j = 0; j < Q; j++)
    {
      args->C[i][j] = args->A[i][args->N_minus_1] *
      args->B[args->N_minus_1][j]
      - args->MulH[i] - args->MulV[j];
      for (unsigned k = 0; k < args->half_N; k++)
      {
        k <<= 1;
        args->C[i][j] += (args->A[i][k] + args->B[k + 1][j]) *
```

```
17                  (args->A[i][k + 1] + args->B[k][j]);
18        }
19      }
20
21    return NULL;
22  }
```

Листинг 3.7: Функция потока вычисления строки при чётных размерах матрицы

```
1  void *multiply_even(void *_args)
2  {
3    MultArgs *args = (MultArgs*) _args;
4
5    const unsigned Q = args->B.get_cols();
6
7    for (unsigned i = args->start_i; i < args->end_i; i++)
8      for (unsigned j = 0; j < Q; j++)
9      {
10       args->C[i][j] = -args->MulH[i] - args->MulV[j];
11       for (unsigned k = 0; k < args->half_N; k++)
12       {
13         k <<= 1;
14         args->C[i][j] += (args->A[i][k] + args->B[k + 1][j]) *
15                  (args->A[i][k + 1] + args->B[k][j]);
16       }
17     }
18
19   return NULL;
20 }
```

3.4 Тесты

Для проверки корректности работы были подготовлены функциональные тесты, представленные в таблице 3.1. Входные данные удовлетворяют условиям, необходимым для умножения матриц, так как проверка на соответствие их размеров возложена на другую функцию.

Таблица 3.1: Функциональные тесты

Матрица 1	**Матрица 2**	**Ожидание**
$\begin{bmatrix} 5 \end{bmatrix}$	$\begin{bmatrix} -8 \end{bmatrix}$	$\begin{bmatrix} -40 \end{bmatrix}$
$\begin{bmatrix} 2 & 1 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ -1 \\ 5 \end{bmatrix}$	$\begin{bmatrix} 6 \end{bmatrix}$
$\begin{bmatrix} 5 & 1 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 3 & -5 \\ 10 & 0 \end{bmatrix}$	$\begin{bmatrix} -10 & 25 \\ -10 & 0 \end{bmatrix}$
$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 \\ 3 & 0 \\ 0 & -2 \end{bmatrix}$	$\begin{bmatrix} 7 & 2 \\ 3 & 8 \end{bmatrix}$
$\begin{bmatrix} 1 & 1 & -1 \\ 5 & -3 & -4 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 3 \\ -2 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 3 \\ -2 & 1 \end{bmatrix}$

В результате проверки последовательная реализация алгоритма Винограда и его параллельная реализация при различном числе потоков прошли все поставленные функциональные тесты.

4. Экспериментальная часть

4.1 Примеры работы

На рисунке 4.1 представлен пример работы программы, демонстрирующий корректную работу алгоритмов.

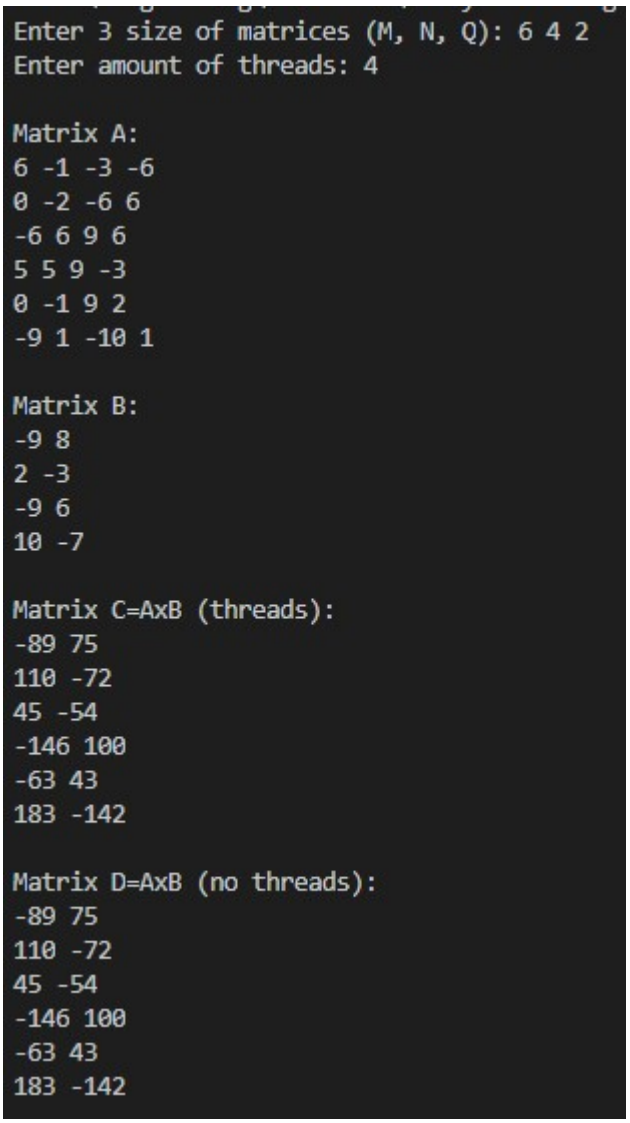

Рис. 4.1: Пример работы программы

4.2 Сравнение работы алгоритмов при чётных размерах матрицы

Для сравнения времени работы алгоритмов умножения матриц были использованы квадратные матрицы размером от 100 до 1000 с шагом 100. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов. Эксперимент проводился на компьютере с 4 логическими процессорами. Результаты измерений показаны в таблице 4.1 и на рисунках 4.2 и 4.3.

Таблица 4.1: Время работы алгоритма при чётных размерах матриц

Размер матриц	Время в тиках при				Время в тиках без потоков
	2 потоках	4 потоках	6 потоках	8 потоках	
100	4370944	1712815	2530937	2325854	3345123
200	10602024	5506377	7759844	7776970	15618673
300	24016614	13613351	17180251	15420683	39259460
400	42269146	22936189	29471685	25318631	69448845
500	64470705	35236340	45405323	37972543	107663545
600	101734972	56470438	69611791	64452769	170605044
700	138204087	82358603	86578899	85281182	233676686
800	180100902	105274512	102682254	104575791	303756098
900	227552320	129232416	136451108	132694553	389185156
1000	282442316	166749445	162272478	163284877	477772819

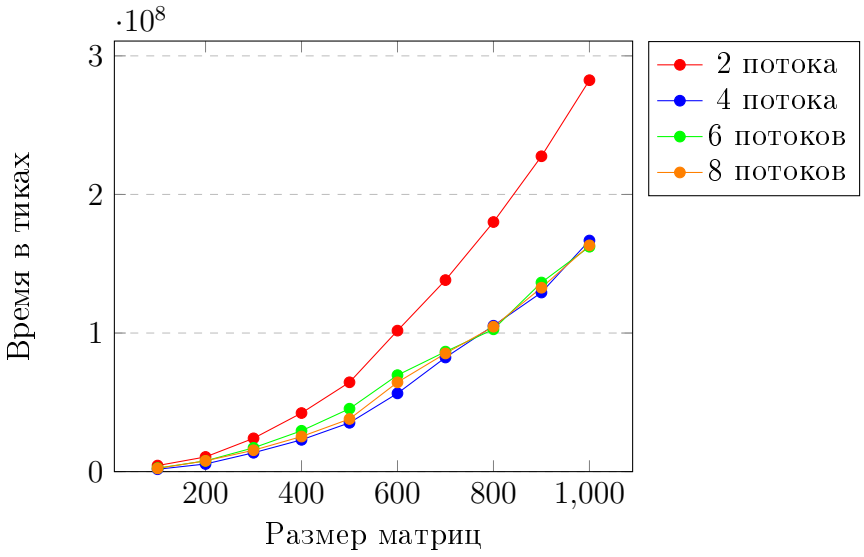

Рис. 4.2: График времени работы алгоритмов при чётных размерах матриц

Можно заметить, что с ростом числа используемых потоков, уменьшается и время работы алгоритма: время работы при 2 потоках в 2 раза больше времени работы при 4 потоках. Однако, время работы при 6 и 8 потоках приблизительно одинаково, при этом оно выше времени работы при 4 потоках: время работы при 6-8 потоках выше в среднем на 40% по сравнению с временем при двух, причём время при восьми меньше, чем время

при шести. Стоит заметить, что при размерах матриц, равным 1000, уже время при 6 потоках является самым минимальным: оно меньше времени при 4 потоках на 2,6%. При этом время работы алгоритма при 8 потоках также меньше времени при двух (на 2,0%).

Наилучший результат показывает многопоточная реализация алгоритма, когда работает на 4 потоках. При этом стоит заметить, что число потоков 4 равно количеству логических процессоров используемого компьютера. Исходя из этого можно сделать вывод о том, что наилучшего времени работы многопоточной реализации алгоритма можно добиться при числе потоков, равному количеству логических процессоров.

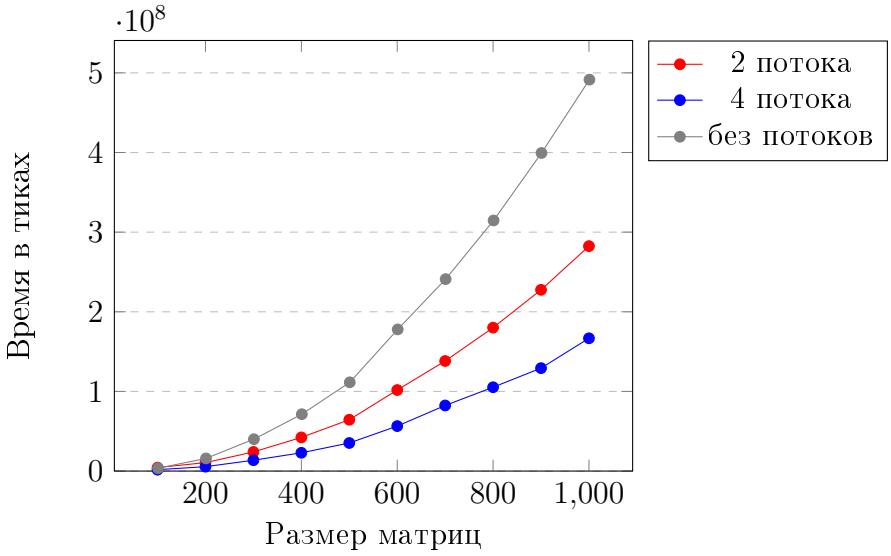

Рис. 4.3: График времени работы двух реализаций при чётных размерах матриц

Легко заметить, что многопоточная реализация даже в худшем случае (при 2 потоках) значительно выигрывает последовательную реализацию: последовательная реализация проигрывает в 1,6 раз многопоточной реализации при работе на 2 потоках, в 3 раза - при работе на 4 потоках.

4.3 Сравнение работы алгоритмов при нечётных размерах матрицы

Для сравнения времени работы алгоритмов умножения матриц были использованы квадратные матрицы размером от 101 до 1001 с шагом 100. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов. Эксперимент проводился на компьютере с 4 логическими процессорами. Результаты измерений показаны в таблице 4.2 и на рисунках 4.4 и 4.5.

При нечётных размерах матриц разница во времени при работе алгоритма при разном числе потоков и в сравнении с последовательной реализацией та же, что и при чётных размерах матриц.

Таблица 4.2: Время работы алгоритма при нечётных размерах матриц

Размер матриц	Время в тиках при 2 потоках	4 потоках	6 потоках	8 потоках	Время в тиках без потоков
101	2857625	1622257	2938607	2514622	3577121
201	10616133	5925311	8578170	8035874	15890971
301	25436636	14236155	18511799	15678003	40031469
401	44256623	25755756	31545421	26279056	71417156
501	68997243	38087065	48514126	39713556	111436166
601	109690871	60416734	73334372	67490652	177868631
701	146843917	85502036	92593251	88600072	241029004
801	193502841	109445844	114245438	113225832	314659781
901	242437841	136339359	143922119	139687196	399413056
1001	298614993	168754709	175676203	169479302	491501628

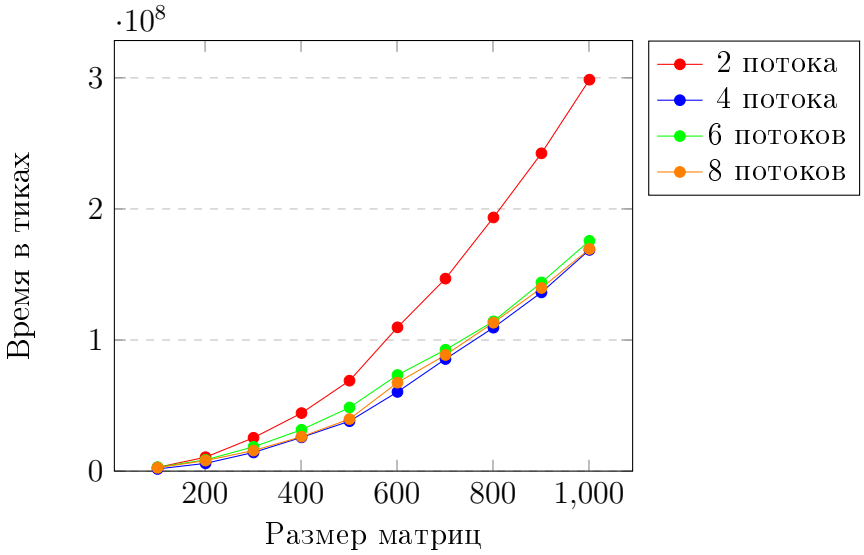

Рис. 4.4: График времени работы многопоточной реализации при нечётных размерах матриц

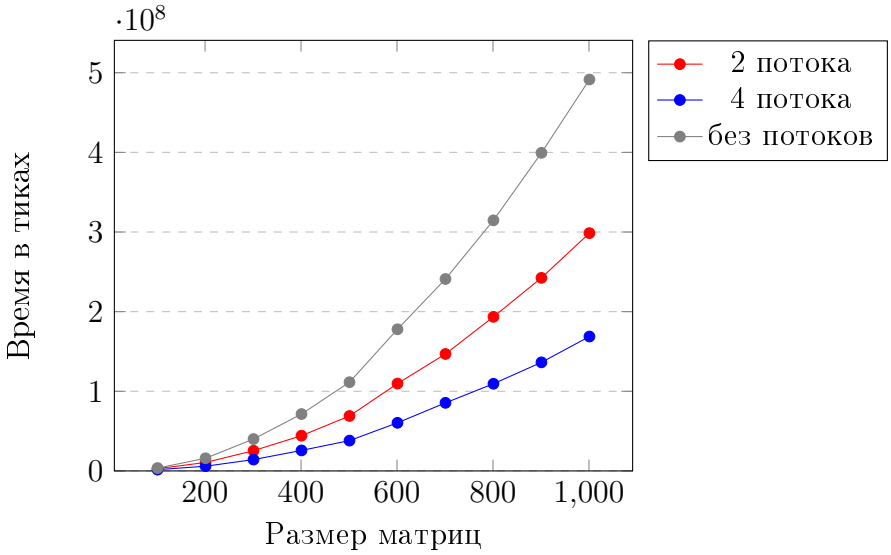

Рис. 4.5: График времени работы двух реализаций при нечётных размерах матриц

Заключение

В ходе лабораторной работы был рассмотрен алгоритм Винограда и возможность его оптимизации путём распараллеливания алгоритм с помощью потоков.

Была рассмотрена технология параллельного программирования и организация взаимодействия параллельных потоков. Потоки исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие данные могут изменяться несколькими потоками, необходимы специальные усилия для организации правильного взаимодействия. В данной лабораторной работе в качестве данных для чтения использовались исходные матрицы, которые требуется перемножить, а в качестве данных для записи - результирующая матрица, причём потоки обрабатывал только свои диапазоны строк, которые не пересекались.

В ходе работы экспериментально была подтверждена эффективность многопоточной реализацией над последовательной, а также был проведён сравнительный анализ времени работы при различном числе потоков. Был сделан вывод о том, что наиболее оптимальное количество потоков в программе должно соответствовать количеству логических процессоров используемого компьютера.

Литература

[1] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16

[2] Jelfimova L. A new fast systolic array for modified Winograd algorithm // Proc. Sevens Int. Workshop on Parallel Processing by Cellular Automata and Array, PARCELLA-96 (Berlin, Germany, Sept. 1996). — Berlin: Akad. Verlag. — 1996.

[3] Константин Баркалов, Владимир Воеводин, Виктор Гергель. Intel Parallel Programming [Электронный ресурс], - режим доступа

[4] https://cppreference.com/ [Электронный ресурс]