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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

О Т Ч Е Т

по лабораторной работе № 4

Название: Разработка параллельных алгоритмов

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ Е.В. Брянская
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

В этой лабораторной работе будет рассматриваться разработка параллельных алгоритмов умножения матриц, в основе которых лежит алгоритм Винограда.

Параллельное программирование используется для распараллеливания обработки информации с целью ускорения вычислений и эффективного использования ресурсов ЭВМ. То есть, возникает совокупность процессов, которые полностью независимы друг от друга или связаны между собой пространственно-временными отношениями [1].

И, соответственно, **параллельный алгоритм** — алгоритм, который может быть реализован по частям с последующим объединением промежуточных результатов с целью получения итогового.

Алгоритм Винограда является последовательным, поэтому может рассматриваться как совокупность подзадач, которые могут быть реализованы независимо друг от друга.

1. Аналитическая часть

В этом разделе будут поставлены цель и основные задачи лабораторной работы, которые будут решаться в ходе её выполнения.

1.1. Цель и задачи

Цель данной работы: разработка и исследование параллельных алгоритмов умножения матриц методом Винограда.

Для достижения поставленной цели необходимо решить ряд следующих **задач**:

1) ввести понятие параллелизма;
2) описать алгоритм;
3) реализовать его;
4) сделать замеры процессорного времени работы на материале серии экспериментов;
5) провести сравнительный анализ многопоточных алгоритмов с базовым (последовательным).

Умножение осуществляется над матрицами $A[M \times N]$ и $B[N \times Q]$. Число столбцов первой матрицы должно совпадать с числом строк второй, а таком случае можно осуществлять умножение. Результатом является матрица $C[M \times Q]$, в которой число строк столько же, сколько в первой, а столбцов, столько же, сколько во второй.

1.2. Общие принципы работы в параллельном режиме

Под **потоком** подразумевается непрерывнная часть кода процесса, которая может выполняться с другими частями выполняемой программы.

Для решения задачи с задействованием нескольких рабочих потоков нужен главный — **диспетчер**. Он создаёт рабочие потоки, передаёт каждому из них **делегат** (указатель на соответствующую функцию). Далее диспетчер запускает их, в качестве аргументов могут быть переданы слудеющие величины:

- границы ответственности;
- ссылка на исходные данные;
- ссылка на память, в которую необходимо записать ответ;

- примитивы синхронизаций.

Главный поток создаёт массив сброшенных **семафоров**, каждый из которых закреплен за определённым потоком. После того, как рабочий поток выполнит поставленную задачу, сохранит или передаст свою часть решения, он устанавливает свой семафор и заканчивает работу.

1.3. Алгоритм Винограда

Обозначим строку $A_{i,*}$ как $\overrightarrow{u}$, $B_{*,j}$ как $\overrightarrow{v}$. Пусть $u = (u_1, u_2, u_3, u_4)$ и $v = (v_1, v_2, v_3, v_4)$, тогда их произведение описывается формулой 1.1.

$$u \cdot v = u_1 \cdot v_1 + u_2 \cdot v_2 + u_3 \cdot v_3 + u_4 \cdot v_4 \tag{1.1}$$

Выражение (1.1) можно преобразовать к виду 1.2.

$$u \cdot v = (u_1 + v_2) \cdot (u_2 + v_1) + (u_3 + v_1) \cdot (u_4 + v_3) - u_1 \cdot u_2 - u_3 \cdot u_4 - v_1 \cdot v_2 - v_3 \cdot v_4 \tag{1.2}$$

Причём, при нечётном значении N нужно отдельно учесть ещё одно слагаемое $u_5 \cdot v_5$.

Алгоритм Винограда основывается на раздельной работе со слагаемыми из выражения (1.2).

1.4. Параллельный алгоритм Винограда

В силу того, что вычисление результата для каждой строки/стобца не зависит от результатов других строк/столбцов, можно распараллелить эти действия.

В алгоритме Винограда формирование конечной матрицы занимает большую часть времени работы всего алгоритма, поэтому в целях уменьшения временных затрат стоит распараллелить эту часть метода.

Во второй главе будут описаны два параллельных алгоритма.

Вывод

Был рассмотрен общий принцип работы алгоритма Винограда, а также возможные способы его распараллеливания.

2. Конструкторская часть

Рассмотрим и оценим работу алгоритмов на матрицах $A[M \times N]$ и $B[N \times Q]$.

2.1. Алгоритм Винограда

Основная задача данного алгоритма — сократить долю умножений в самом тяжёлом, затратном участке кода. Для этого используется формула (1.2).

Некоторые из слагаемых можно вычислить заранее и использовать повторно для каждой строки первой матрицы и для каждого столбца второй. Таким образом, трудоёмкость алгоритма уменьшается за счёт сокращения количества производимых операций.

В этом алгоритме важно учитывать, что при нечётном значении N, необходимо вычислять дополнительное слагаемое $u_N \cdot v_N$.

Схема алгоритма представлена на Рис.2.1.

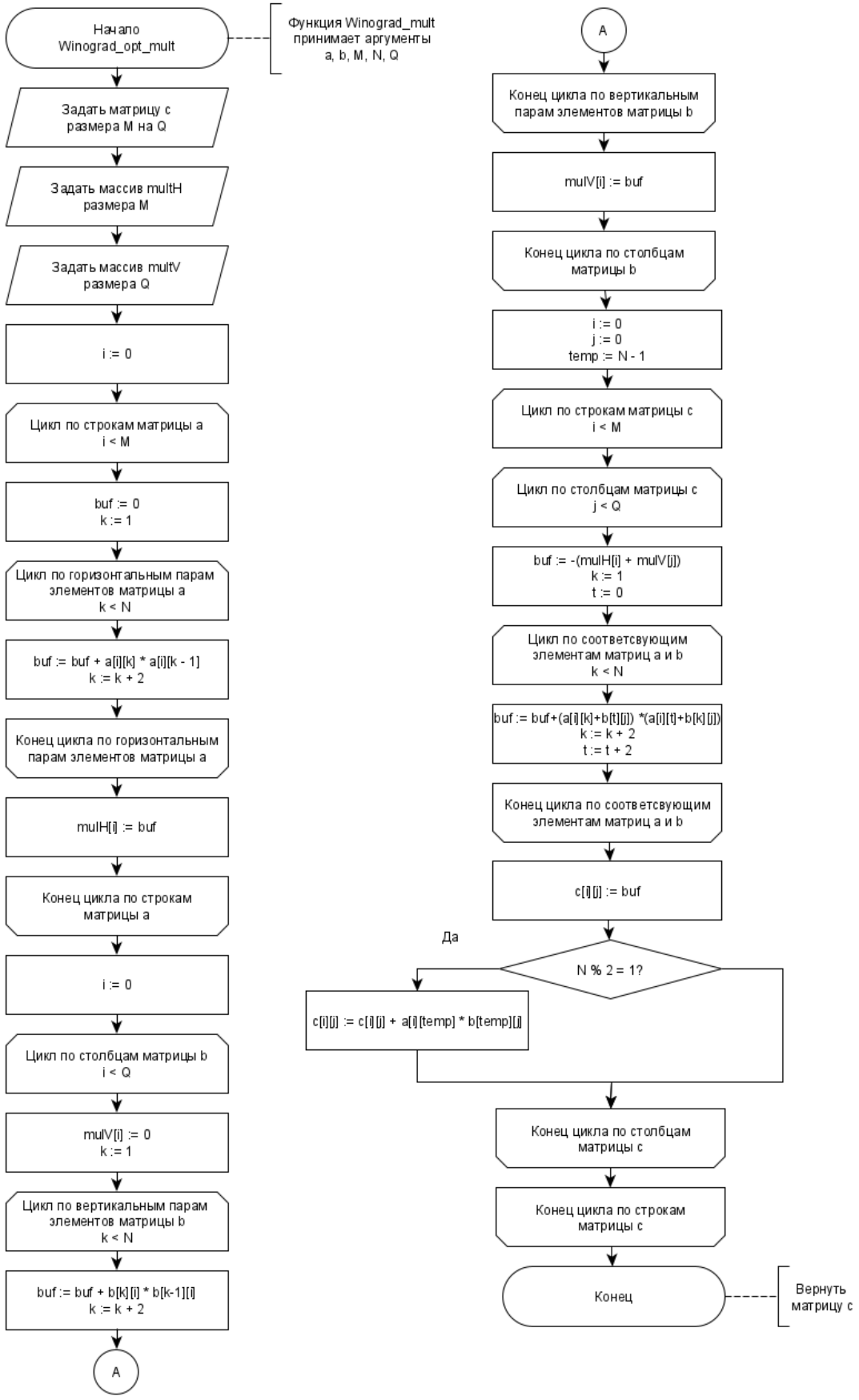

Рис. 2.1 — Алгоритм Винограда

2.2. Распараллеленный алгоритм Винограда (1 вариант)

В этом алгоритме параллельно выполняются вычисления по строкам матрицы C. Каждому потоку выделяется строки, начиная с i, где i - порядковый номер потока, с шагом $step$ - количество потоков.

Схема алгоритма представлена на Рис.2.2 (главный поток) и на Рис. 2.3 (рабочий поток).

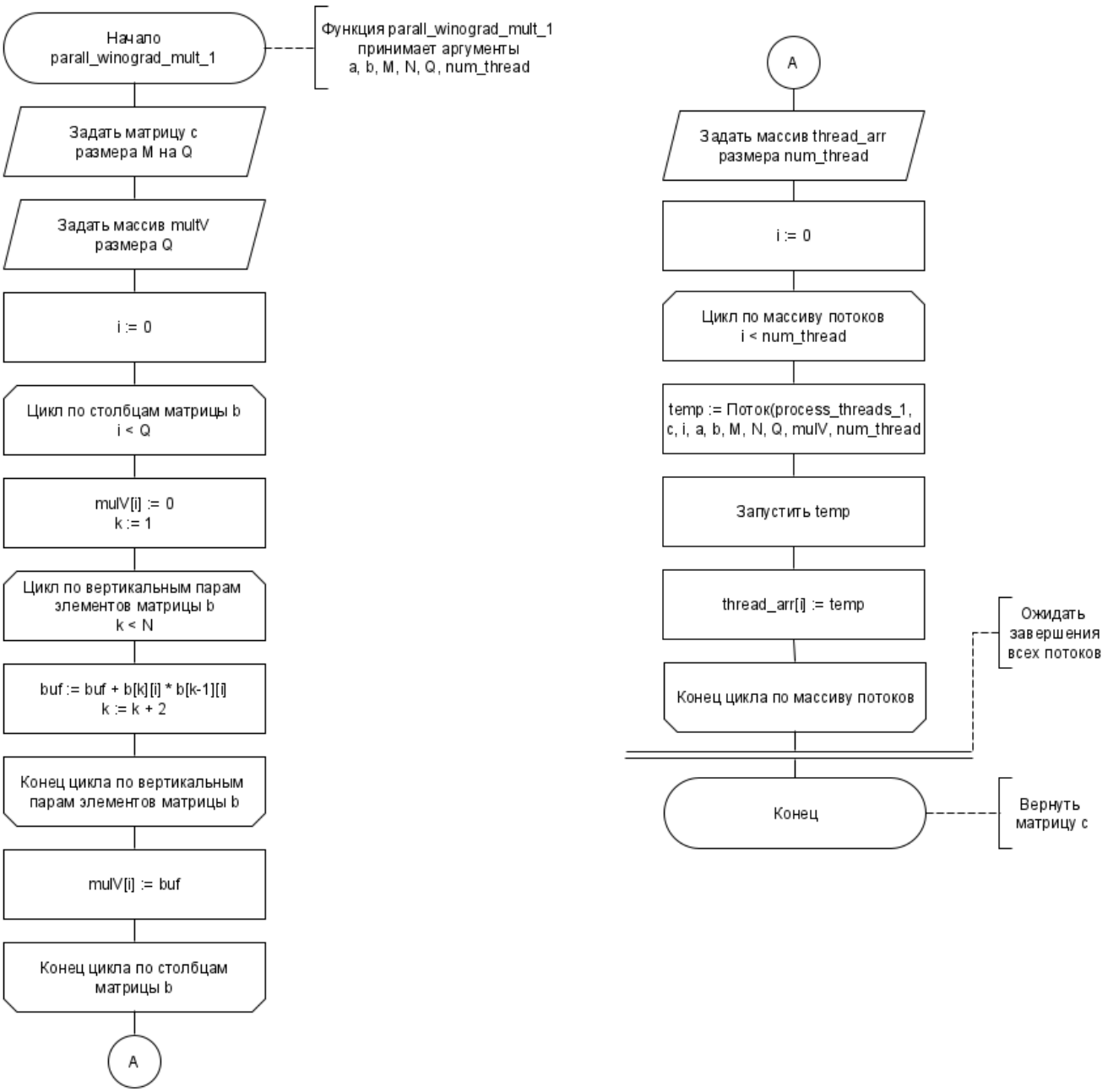

Рис. 2.2 — Распараллеленный алгоритм Винограда (1 вариант). Главный поток

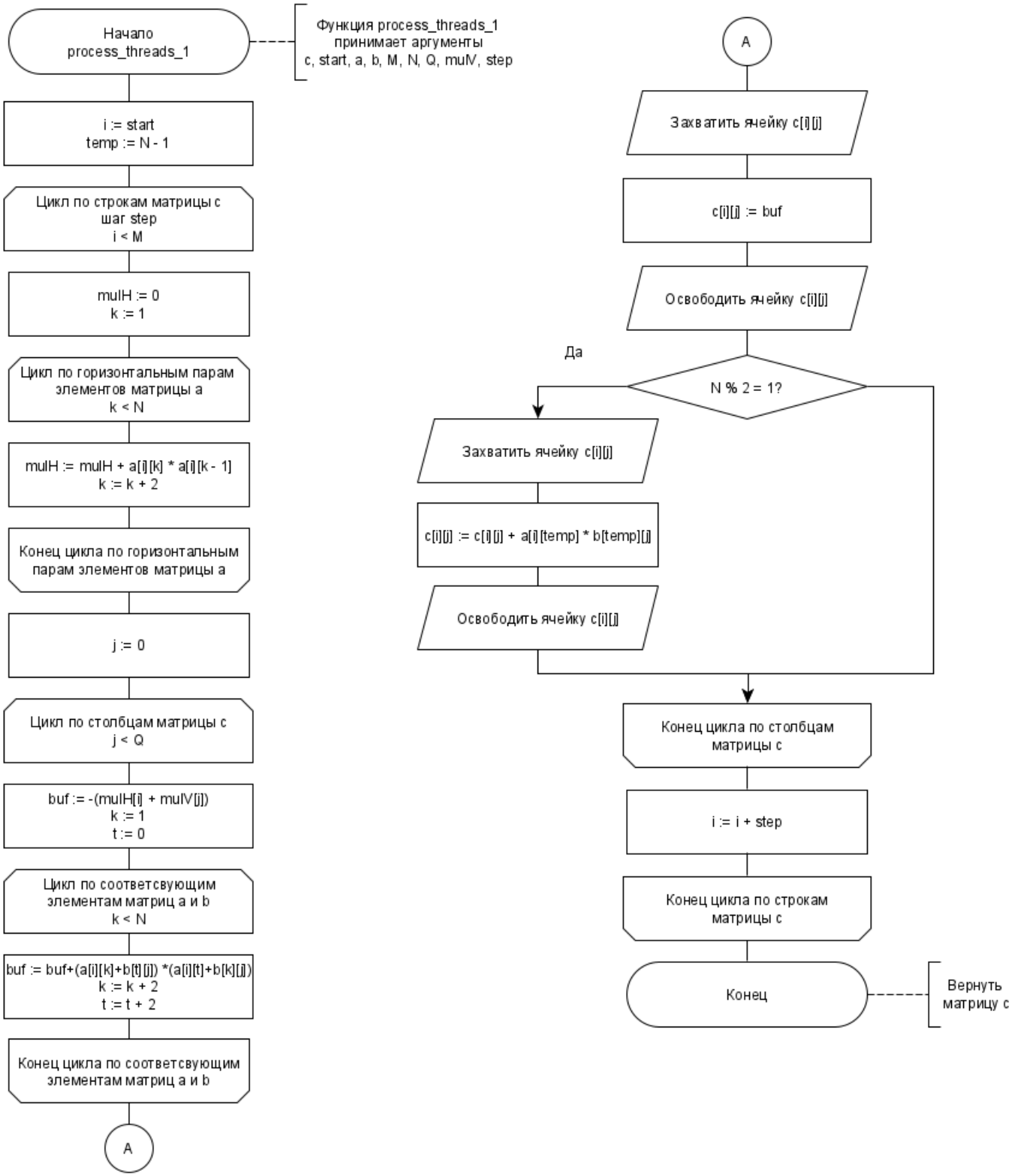

Рис. 2.3 — Распараллеленный алгоритм Винограда (1 вариант). Рабочий поток

2.3. Распараллеленный алгоритм Винограда (2 вариант)

В отличие от предыдущего алгоритма, в этом алгоритме параллельно выполняются вычисления по столбцам. Каждому потоку выделяется столбцы, начиная с i, где i - порядковый номер потока, с шагом $step$ - количество потоков.

Схема алгоритма представлена на Рис.2.4 (главный поток) и на Рис. 2.5 (рабочий поток).

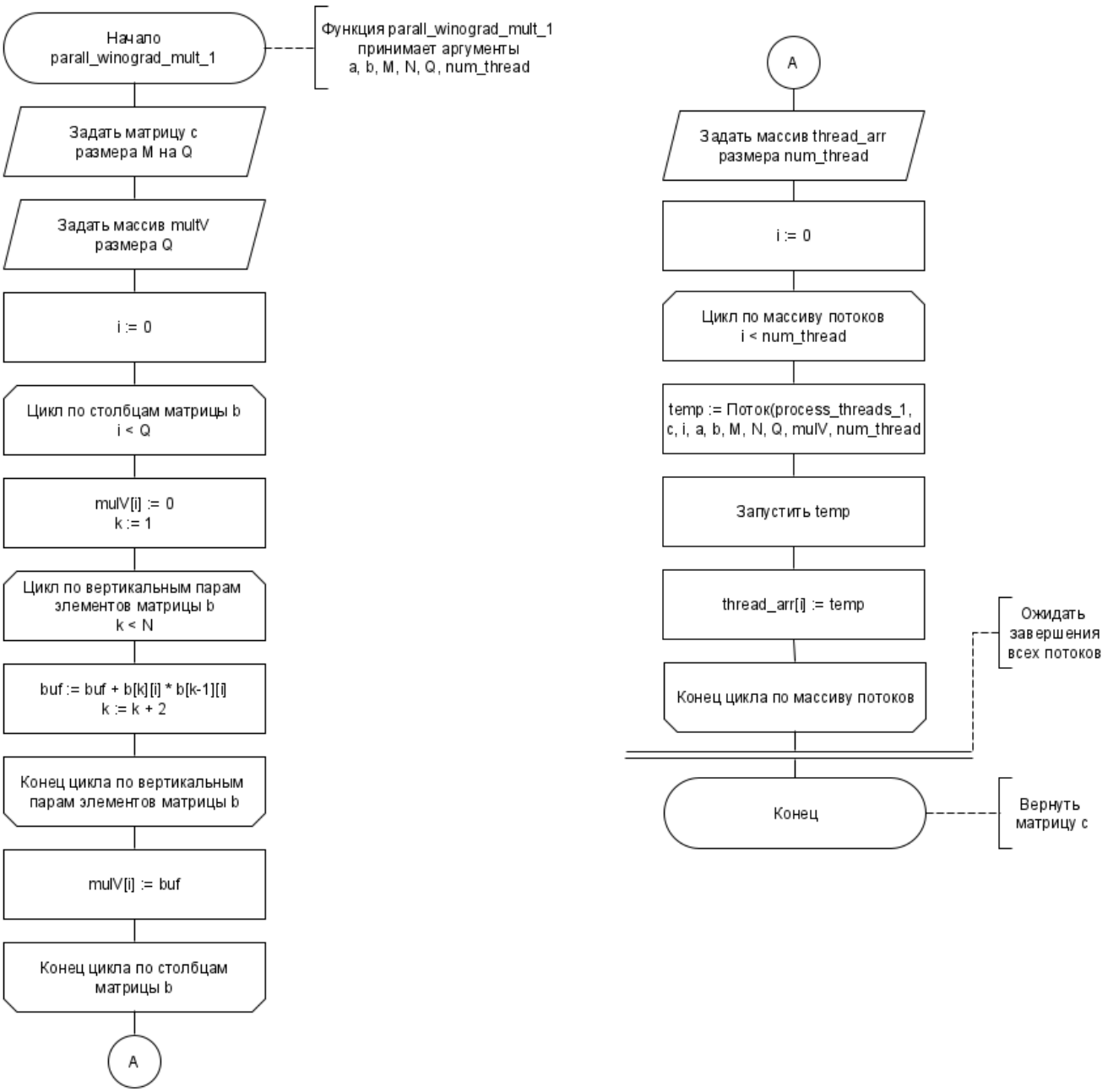

Рис. 2.4 — Распараллеленный алгоритм Винограда (2 вариант). Главный поток

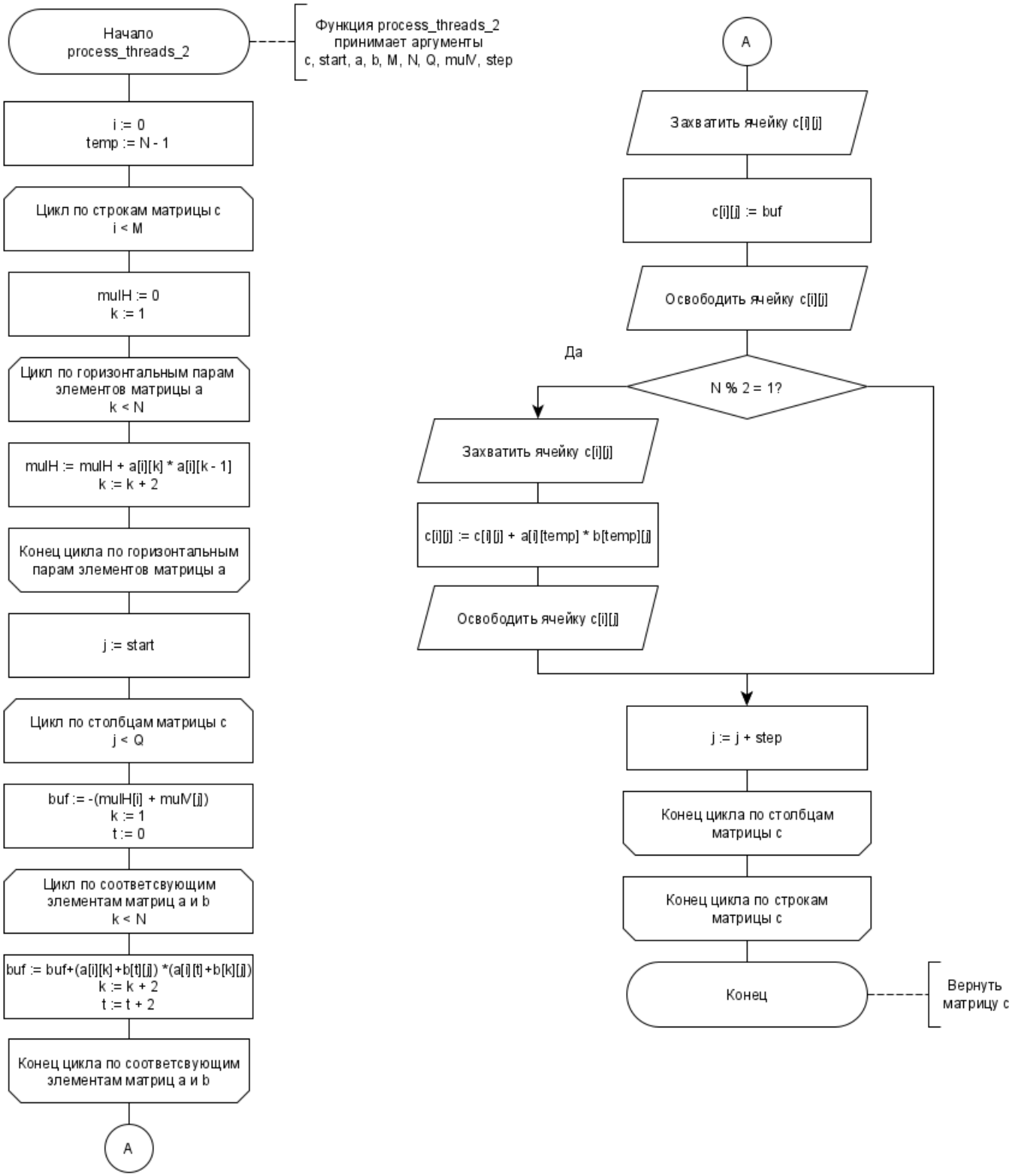

Рис. 2.5 — Распараллеленный алгоритм Винограда (2 вариант). Рабочий поток

2.4. Требования к ПО

Для корректной работы алгоритмов и проведения тестов необходимо выполнить следующее.

- Обеспечить возможность ввода двух матриц через консоль.
- В случае ввода некорректных данных вывести соответствующее сообщение. Программа не должна аварийно завершаться.
- Обеспечить возможность консольного ввода предела количества используемых потоков.
- Реализовать функцию замера процессорного времени, которое выбранный метод затрачивает на вычисление результатов. Вывести результаты замеров на экран.

2.5. Заготовки тестов

При проверке на корректность работы реализованных функций необходимо провести следующие тесты:

- один поток;
- умножение матриц размером 1×1;
- число потоков меньше, чем M, N, Q;
- число потоков больше, чем M, N, Q.

Вывод

В этом разделе разобраны основные принципы выбранных алгоритмов, построены схемы их работы. Также описаны требования к программному обеспечению и приведены заготовки тестов, которые будут использоваться в дальнейшем.

3. Технологическая часть

В данном разделе будут приведены листинги функций разрабатываемых алгоритмов и проведено их тестирование.

3.1. Выбранный язык программирования

Для выполнения этой лабораторной работы был выбран язык программирования C++, так как есть большой навык работы с ним и с подключаемыми библиотеками, которые также использовались для проведения тестирования и замеров. Для реализации параллельных алгоритмов использовались библиотеки *thread* [3], *mutex* [4].

Использованная среда разработки - Visual Studio [5].

3.2. Листинг кода

Ниже представлены Листиги 3.1, 3.2, 3.3 функций, реализующих последовательный алгоритм Винограда и его модификации в виде распараллеленных алгоритмов.

```
#include "winograd_mult.h"

matrix_t winograd_mult(matrix_t a, matrix_t b, int m, int n, int q)
{
  arr_t mulV = create_array(q);
  int temp = n - 1;
  double mulH, buf;

  matrix_t c = create_matrix(m, q);

  for (int i = 0; i < q; i++)
  {
    buf = 0;
    for (int k = 1; k < n; k += 2)
    buf += b[k][i] * b[k - 1][i];
    mulV[i] = buf;
  }

  for (int i = 0; i < m; i++)
  {
    mulH = 0;
    for (int k = 1; k < n; k += 2)
```

```
23        mulH += a[i][k] * a[i][k - 1];
24
25      for (int j = 0; j < q; j++)
26      {
27        buf = -(mulH + mulV[j]);
28        for (int k = 1, t = 0; k < n; k += 2, t += 2)
29          buf += (a[i][k] + b[t][j]) * (a[i][t] + b[k][j]);
30        c[i][j] = buf;
31
32        if (n % 2)
33          c[i][j] += a[i][temp] * b[temp][j];
34      }
35    }
36
37    free_array(&mulV);
38
39    return c;
40  }
```

Листинг 3.1 — Последовательный алгоритм Винограда

```
1  #include "parall_winograd.h"
2
3  mutex set_mutex;
4
5
6  void process_threads_1(matrix_t& c, int start, matrix_t& a, matrix_t& b,
       int m, int n, int q, arr_t mulV, int step)
7  {
8    double mulH, buf;
9    int temp = n - 1;
10
11   for (int i = start; i < m; i += step)
12   {
13     mulH = 0;
14     for (int k = 1; k < n; k += 2)
15       mulH += a[i][k] * a[i][k - 1];
16
17     for (int j = 0; j < q; j++)
18     {
19       buf = -(mulH + mulV[j]);
```

```
20        for (int k = 1, t = 0; k < n; k += 2, t += 2)
21          buf += (a[i][k] + b[t][j]) * (a[i][t] + b[k][j]);
22
23        set_mutex.lock();
24        c[i][j] = buf;
25        set_mutex.unlock();
26
27        if (n % 2)
28        {
29          set_mutex.lock();
30          c[i][j] += a[i][temp] * b[temp][j];
31          set_mutex.unlock();
32        }
33      }
34    }
35  }
36
37  matrix_t parall_winograd_mult_1(matrix_t a, matrix_t b, int m, int n,
        int q, int num_thread)
38  {
39    arr_t mulV = create_array(q);
40    double buf;
41
42    matrix_t c = create_matrix(m, q);
43    vector<thread> thread_arr;
44
45    for (int i = 0; i < q; i++)
46    {
47      buf = 0;
48      for (int k = 1; k < n; k += 2)
49        buf += b[k][i] * b[k - 1][i];
50      mulV[i] = buf;
51    }
52
53    for (int i = 0; i < num_thread; i++)
54      thread_arr.push_back(thread(process_threads_1, ref(c), i, ref(a),
        ref(b), m, n, q, mulV, num_thread));
55
56    for (int i = 0; i < num_thread; i++)
57      thread_arr[i].join();
```

```
58
59    free_array(&mulV);
60
61    return c;
62  }
```

Листинг 3.2 — Распараллеленный алгоритм Винограда (1 вариант)

```
1  #include "parall_winograd.h"
2
3  mutex set_mutex;
4
5
6  void process_threads_2(matrix_t& c, int start, matrix_t& a, matrix_t& b,
       int m, int n, int q, arr_t mulV, int step)
7  {
8    double mulH, buf;
9    int temp = n - 1;
10
11   for (int i = 0; i < m; i++)
12   {
13     mulH = 0;
14     for (int k = 1; k < n; k += 2)
15       mulH += a[i][k] * a[i][k - 1];
16
17     for (int j = start; j < q; j += step)
18     {
19       buf = -(mulH + mulV[j]);
20       for (int k = 1, t = 0; k < n; k += 2, t += 2)
21         buf += (a[i][k] + b[t][j]) * (a[i][t] + b[k][j]);
22
23       set_mutex.lock();
24       c[i][j] = buf;
25       set_mutex.unlock();
26
27       if (n % 2)
28       {
29         set_mutex.lock();
30         c[i][j] += a[i][temp] * b[temp][j];
31         set_mutex.unlock();
32       }
```

```
33      }
34    }
35  }
36
37  matrix_t parall_winograd_mult_2(matrix_t a, matrix_t b, int m, int n,
        int q, int num_thread)
38  {
39    arr_t mulV = create_array(q);
40    double buf;
41
42    matrix_t c = create_matrix(m, q);
43    vector<thread> thread_arr;
44
45    for (int i = 0; i < q; i++)
46    {
47      buf = 0;
48      for (int k = 1; k < n; k += 2)
49        buf += b[k][i] * b[k - 1][i];
50      mulV[i] = buf;
51    }
52
53    for (int i = 0; i < num_thread; i++)
54      thread_arr.push_back(thread(process_threads_2, ref(c), i, ref(a),
       ref(b), m, n, q, mulV, num_thread));
55
56    for (int i = 0; i < num_thread; i++)
57      thread_arr[i].join();
58
59    free_array(&mulV);
60
61    return c;
62  }
```

Листинг 3.3 — Распараллеленный алгоритм Винограда (2 вариант)

3.3. Результаты тестов

Для тестирования были написаны функции, проверяющие, согласно заготовкам выше, случаи. Выводы о корректности работы делаются на основе сравнения результатов.

Все тесты пройдены успешно. На Листинге 3.4 представлены сами тесты.

```
#include "tests.h"

double PCFreq = 0.0;
__int64 CounterStart = 0;

bool mult_cmp(matrix_t a, matrix_t b, int m, int n, int q, int
    num_thread = 5)
{
  matrix_t c1 = winograd_mult(a, b, m, n, q);
  matrix_t c2 = parall_winograd_mult_1(a, b, m, n, q, num_thread);

  bool res = cmp_matrix(c1, c2, m, q);

  free_matrix(&c1, m, q);
  free_matrix(&c2, m, q);

  return res;
}

// Матрицы размером 1 x 1
void test_size_1_1()
{
  int n = 1;
  matrix_t a = create_matrix(n, n);
  matrix_t b = create_matrix(n, n);

  a[0][0] = 15;
  b[0][0] = -7;

  if (!mult_cmp(a, b, n, n, n))
  {
    cout << endl << __FUNCTION__ << " FAILED" << endl;
    free_matrix(&a, n, n);
    free_matrix(&b, n, n);
    return;
  }

  free_matrix(&a, n, n);
```

```
38    free_matrix(&b, n, n);
39
40    cout << endl << __FUNCTION__ << " OK" << endl;
41  }
42
43  // Один поток
44  void test_one_thread()
45  {
46    int m[] = { 2, 6, 10 };
47    int n[] = { 1, 4, 7 };
48    int q[] = { 3, 4, 8 };
49    int num_thread = 1;
50
51    for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
52    {
53      matrix_t a = random_fill_matrix(m[i], n[i]);
54      matrix_t b = random_fill_matrix(n[i], q[i]);
55
56      if (!mult_cmp(a, b, m[i], n[i], q[i], num_thread))
57      {
58        cout << endl << __FUNCTION__ << " FAILED" << endl;
59        free_matrix(&a, m[i], n[i]);
60        free_matrix(&b, n[i], q[i]);
61        return;
62      }
63      free_matrix(&a, m[i], n[i]);
64      free_matrix(&b, n[i], q[i]);
65
66      cout << endl << __FUNCTION__ << " OK" << endl;
67    }
68  }
69
70  // Потоков меньше, чем значения M, N, Q
71  void test_less_mnq()
72  {
73    int m[] = { 4, 8, 10 };
74    int n[] = { 3, 5, 12 };
75    int q[] = { 3, 6, 8 };
76    int num_thread[] = { 2, 4, 2 };
77
```

```
78    for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
79    {
80      matrix_t a = random_fill_matrix(m[i], n[i]);
81      matrix_t b = random_fill_matrix(n[i], q[i]);
82
83      if (!mult_cmp(a, b, m[i], n[i], q[i], num_thread[i]))
84      {
85        cout << endl << __FUNCTION__ << " FAILED" << endl;
86        free_matrix(&a, m[i], n[i]);
87        free_matrix(&b, n[i], q[i]);
88        return;
89      }
90      free_matrix(&a, m[i], n[i]);
91      free_matrix(&b, n[i], q[i]);
92
93      cout << endl << __FUNCTION__ << " OK" << endl;
94    }
95  }
96
97  // Потоков больше, чем значения M, N, Q
98  void test_more_mnq()
99  {
100   int m[] = { 4, 8, 10 };
101   int n[] = { 3, 5, 12 };
102   int q[] = { 3, 6, 8 };
103   int num_thread[] = { 7, 10, 15 };
104
105   for (int i = 0; i < sizeof(n) / sizeof(n[0]); i++)
106   {
107     matrix_t a = random_fill_matrix(m[i], n[i]);
108     matrix_t b = random_fill_matrix(n[i], q[i]);
109
110     if (!mult_cmp(a, b, m[i], n[i], q[i], num_thread[i]))
111     {
112       cout << endl << __FUNCTION__ << " FAILED" << endl;
113       free_matrix(&a, m[i], n[i]);
114       free_matrix(&b, n[i], q[i]);
115       return;
116     }
117     free_matrix(&a, m[i], n[i]);
```

```
118     free_matrix(&b, n[i], q[i]);
119
120     cout << endl << __FUNCTION__ << " OK" << endl;
121   }
122 }
123
124 // Умножение одних и тех же матриц на разном числе потоков
125 void test_dif_num_threads()
126 {
127   int m = 20, n = 17, q = 23;
128   int num_thread[] = { 1, 2, 5, 8, 10, 14, 16, 19, 20, 24, 26, 30 };
129
130   for (int i = 0; i < sizeof(num_thread) / sizeof(num_thread[0]); i++)
131   {
132     matrix_t a = random_fill_matrix(m, n);
133     matrix_t b = random_fill_matrix(n, q);
134
135     if (!mult_cmp(a, b, m, n, q, num_thread[i]))
136     {
137       cout << endl << __FUNCTION__ << " FAILED" << endl;
138       free_matrix(&a, m, n);
139       free_matrix(&b, n, q);
140       return;
141     }
142     free_matrix(&a, m, n);
143     free_matrix(&b, n, q);
144
145     cout << endl << __FUNCTION__ << " OK" << endl;
146   }
147 }
148
149 void run_tests()
150 {
151   test_size_1_1();
152   test_one_thread();
153   test_less_mnq();
154   test_more_mnq();
155   test_dif_num_threads();
156 }
```

Листинг 3.4 — Тесты

3.4. Оценка времени

Процессорное время измеряется с помощью функции QueryPerformanceCounter библиотеки windows.h [6]. Осуществление замеров показано ниже (Листинг 3.5).

```
1  void test_range(int n, vector<int>& num_threads)
2  {
3    for (int key : num_threads)
4    {
5      cout << endl << endl << "Размер тестируемых матриц: " << n << "x" << n
       << endl;
6      cout << "Количество потоков: " << key;
7
8      cout << endl << "-----Winograd(improved)-----" << endl;
9      test_time_cons(winograd_mult, n);
10     cout << endl << "-----Parallel Winograd(1)-----" << endl;
11     test_time_parall(parall_winograd_mult_1, n, key);
12     cout << endl << "-----Parallel Winograd(2)-----" << endl;
13     test_time_parall(parall_winograd_mult_2, n, key);
14   }
15 }
16
17 void start_measuring()
18 {
19   LARGE_INTEGER li;
20   QueryPerformanceFrequency(&li);
21
22   PCFreq = double(li.QuadPart) / 1000;
23
24   QueryPerformanceCounter(&li);
25   CounterStart = li.QuadPart;
26 }
27
28 double get_measured()
29 {
30   LARGE_INTEGER li;
31   QueryPerformanceCounter(&li);
32
33   return double(li.QuadPart - CounterStart) / PCFreq;
34 }
```

```
35
36 // Замеры процессорного времени
37 void test_time_parall(matrix_t(*f)(matrix_t, matrix_t, int, int, int,
       int), int n, int num_threads)
38 {
39   matrix_t a = random_fill_matrix(n, n);
40   matrix_t b = random_fill_matrix(n, n);
41   matrix_t c;
42
43   int num = 0;
44   start_measuring();
45
46   while (get_measured() < 3 * 1000)
47   {
48     c = f(a, b, n, n, n, num_threads);
49     free_matrix(&c, n, n);
50     num++;
51   }
52
53   double t = get_measured() / 1000;
54   cout << "Выполнено " << num << " операций за " << t << " секунд" << endl;
55   cout << "Время: " << t / num << endl;
56
57   free_matrix(&a, n, n);
58   free_matrix(&b, n, n);
59 }
60
61 void test_time_cons(matrix_t(*f)(matrix_t, matrix_t, int, int, int), int
       n)
62 {
63   matrix_t a = random_fill_matrix(n, n);
64   matrix_t b = random_fill_matrix(n, n);
65   matrix_t c;
66
67   int num = 0;
68   start_measuring();
69
70   while (get_measured() < 3 * 1000)
71   {
72     c = f(a, b, n, n, n);
```

```
73      free_matrix(&c, n, n);
74      num++;
75    }
76
77    double t = get_measured() / 1000;
78    cout << "Выполнено " << num << " операций за " << t << " секунд" << endl;
79    cout << "Время: " << t / num << endl;
80
81    free_matrix(&a, n, n);
82    free_matrix(&b, n, n);
83  }
```

Листинг 3.5 — Замеры процессорного времени

Вывод

Были разработаны функции, реализующие текущие алгоритмы, приведены листинги кода каждой из них, а также листинги тестовых функций, направленных на проверку корректности их работы, и замеров процессорного времени.

4. Исследовательская часть

Проведём замеры процессорного времени, которое затрачивается каждым алгоритмом на умножение матриц различного размера, и сравним полученные результаты.

4.1. Характеристики ПК

При проведении замеров времени использовался компьютер, имеющий следующие характеристики:

- ОС - Windows 10 Pro;
- процессор - Inter Core i7 10510U (1800 МГц);
- объём ОЗУ - 16 Гб;
- число логических ядер - 8.

4.2. Измерения

Для проведения замеров процессорного времени использовались квадратные матрицы размера $N \times N$, где $N \in \{100, 200, 400, 600, 800, 1000\}$. Их содержимое генерируется случайным образом.

На графике 4.1 представлены результаты замеров процессорного времени работы реализаций алгоритмов (в секундах). В процессе измерения варьировался не только размер умножаемых матриц, но и количество потоков K в параллельных алгоритмах, где $K \in \{1, 2, 4, 8, ...4M\}$, M - число логических ядер.

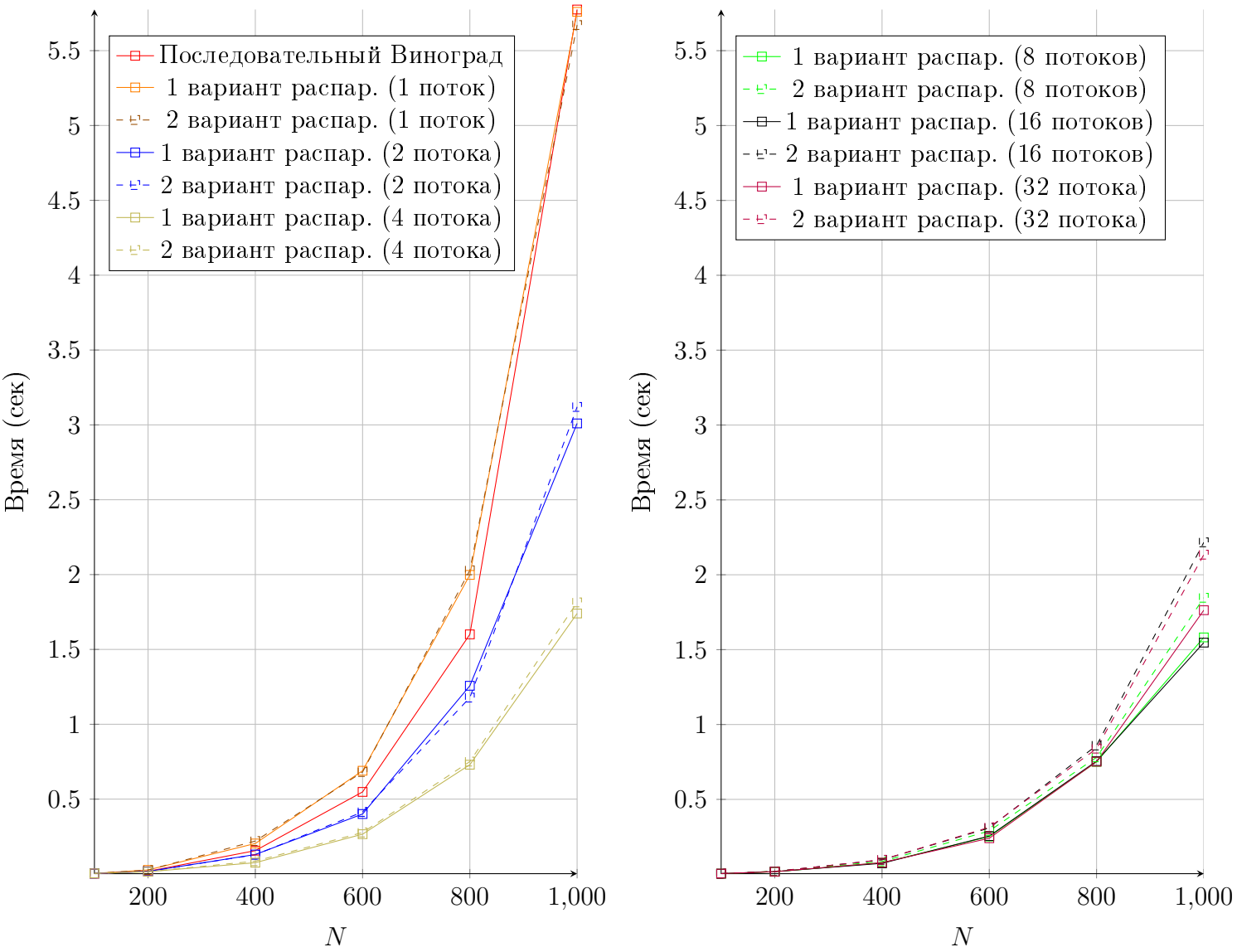

Рис. 4.1 — Сравнение времени работы последовательного Винограда и двух вариантов распараллеливания алгоритма

Согласно полученным данным можно сделать следующие **выводы**:

- обе версии параллельных алгоритмов на одном потоке показывают результаты хуже (примерно на 20%), чем последовательный алгоритм Винограда, в силу того, что организация и работа с потоками требуют дополнительных временных затрат;
- начиная с двух потоков обе версии параллельного алгоритма обрабатывают матрицы эффективнее, чем алгоритм Винограда;
- 1 вариант распараллеливания (по строкам) работает быстрее 2 варианта (распараллеливание по столбцам), до четырёх потоков разница составляет меньше 3%, а с восьми до тридцати двух потоков разница может варьироваться от 15% до 30%

(на шестнадцати потоках);

- самым быстродейственным по времени оказался 1 вариант параллельного алгоритма (расспараллеливание по строкам) на шестнадцати потоках, по сравнению с последовательным алгоритмом Винограда он лучше на 53% на матрицах размером меньше 1000×1000, а на больших преимущество превышает 70%;
- увеличение количества потоков до тридцати двух не оправдано, затрачиваемое время значительно больше того, что было при меньшем количестве потоков.

Вывод

Проведены замеры процессорного времени, и на основе полученных данных были построены графики, описывающие время, которое каждый из алгоритмов затрачивает на умножение матриц конкретного размера, а в случае многопоточных алгоритмов ещё и зависимость времени от количества выделенных потоков. В результате анализа получившихся графиков были сделаны выводы, приведённые выше.

Заключение

В ходе лабораторной работы была достигнута поставленная цель, а именно, разработаны и исследованы параллельные алгоритмы умножения матриц методом Винограда.

В процессе выполнения были решены все задачи. Описаны все рассматриваемые алгоритмы. Все проработанные алгоритмы реализованы, кроме того, были проведены замеры процессорного времени работы на материале серии экспериментов и проведён сравнительный анализ, сделаны выводы.

По результатам замеров процессорного времени сделаны следующие заключения.

- Использование параллельных алгоритмов на одном потоке неэффективно по времени, так как присутствуют значительные временные затраты на организацию многопоточности, и результат по времени примерно на 20% уступает последовательному алгоритмому.
- Но уже с двух потоков многопоточные алгоритмы оказываются эффективнее.
- Метод, распараллеливающий алгоритм Винограда по строкам, затрачивает меньше времени на умножение матриц, чем алгоритм, работающий по столбцам, до четырёх потоков разница примерно 3%, но с восьми принимает значения от 15% до 30%.
- Самым быстродейственным по времени среди разработанных алгоритмов оказался 1 алгоритм (распараллеливание по строкам). Примерно на 53% он лучше последовательного алгоритма Винограда на матрицах размером меньше 1000×1000 и далее его выигрыш растёт ещё больше (на 1000×1000 он составляет больше 70%).
- Увеличив число рабочих потоков до тридцати двух, не удалось получить выигрыша по времени по сравнению с предыдущими показателями, наоборот, затрачиваемое время увеличилось.

Список литературы

1. Иванов, К. К. Принципы разработки параллельных методов / К. К. Иванов, С. А. Раздобудько, Р. И. Ковалев. — Текст : непосредственный // Молодой ученый. — 2017. — № 3 (137). — С. 30-32. — URL: https://moluch.ru/archive/137/38412/ (дата обращения: 21.10.2020).

2. Кормен, Томас Х. и др Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс 2018. - 1328 с. : ил. - Парал. тит. англ. - ISBN 978-5-8459-2016-4 (рус.).

3. Документация по Стандартной библиотекн языка C++ thread [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/standard-library/thread?view=vs-2019, свободный (дата обращения 22.10.2020)

4. Документация по Стандартной библиотекн языка C++ mutex [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/standard-library/mutex?view=vs-2019, свободный (дата обращения 22.10.2020)

5. Документация по Visual Studio 2019 [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/visualstudio/windows/?view=vs-2019, свободный (дата обращения: 21.10.2020)

6. QueryPerformanceCounter function [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter, свободный (дата обращения: 22.10.2020).