Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №4

по курсу «Анализ алгоритмов»

Распараллеливание алгоритма умножение матриц Винограда

Выполнил: Сорокин А.П., гр. ИУ7-52Б

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

Москва, 2019 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: последовательный и параллельный.
Для того чтобы добиться этой цели, были поставлены следующие задачи:

- изучить алгоритм Винограда;
- изучить методы параллельного программирования;
- применить знания программирования для реализации указанных алгоритмов;
- 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 $m \times q$

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

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 каждой строки результирующей матрицы будет отводиться под отдельный поток.

1.3 Параллельное программирование

В рамках данной работы используется следующая модель вычислений:

- операции, имеющие трудоемкость 1: $<$, $>$, $=$, $<=$, $=>$, $==$, $!=$,$+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, [ ];
- оператор условного перехода имеет трудоёмкость, равную трудоёмкости операторов тела условия;
- оператор цикла for имеет трудоемкость:

$$F_{for} = F_{init} + F_{check} + N * (F_{body} + F_{inc} + F_{check}), \tag{1.6}$$

где F_{init} – трудоёмкость инициализации, F_{check} – трудоёмкость проверки условия, F_{inc} – трудоёмкость инкремента аргумента, F_{body} – трудоёмкость операций в теле цикла, N – число повторений. [3]

2. Конструкторская часть

2.1 Схемы алгоритмов

На рисунке 2.1 представлена схема алгоритма Винограда.

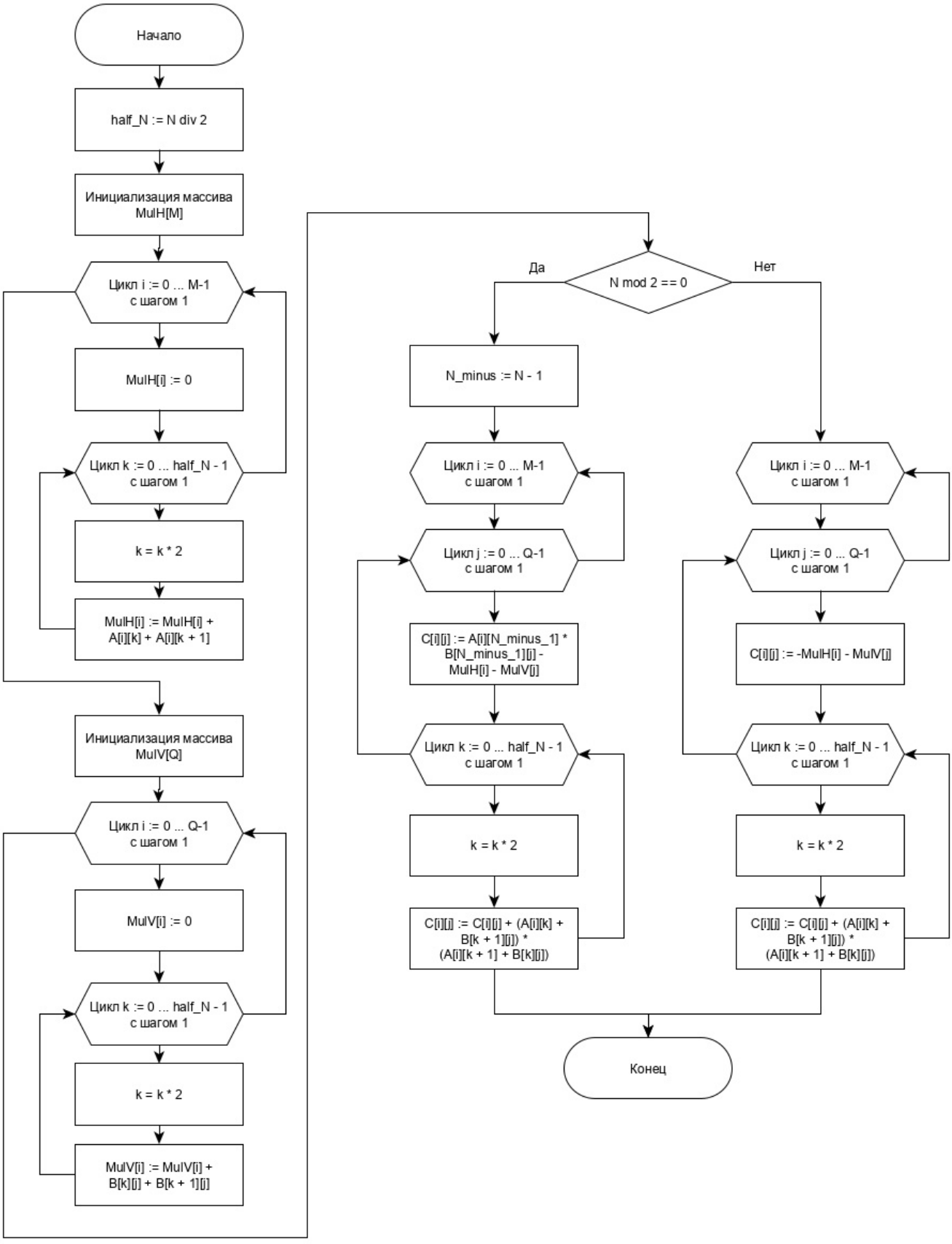

Рис. 2.1: Алгоритм Винограда

2.2 Распараллеливание алгоритма

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.

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

Листинг 3.2: Структура аргументов

```
1  typedef struct
2  {
3    Matrix &A;
4    Matrix &B;
5    Matrix &C;
6    int *MulH;
7    int *MulV;
```

```
  unsigned half_N;
  unsigned N_minus_1;

  unsigned i;
  unsigned start_i;
  unsigned end_i;
  unsigned step;
  unsigned left;
  unsigned amount;
} MultArgs;

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

Листинг 3.3: 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)
  {
```

```
35      printf("Can't join thread thr_MulH, status = %d\n", status);
36      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    MultArgs args = {
46      A, B, C, MulH, MulV, half_N, N - 1, thread_amount, M / thread_amount,
47      M % thread_amount, 0, 0, M / thread_amount
48    };
49    void* (*thread_func)(void*) = multiply_odd;
50    if (N % 2)
51      thread_func = multiply_even;
52
53    for (unsigned i = 0; i < thread_amount; i++)
54    {
55      status = pthread_create(&(threads[i]), NULL, thread_func,
56      (void*) &args);
57      if (status)
58      {
59        printf("Can't create thread %u, status = %d\n", i, status);
60        exit(ERROR_CREATE_THREAD);
61      }
62    }
63
64    for (unsigned i = 0; i < thread_amount; i++)
65    {
66      status = pthread_join(threads[i], (void**)&status_addr);
67      if (status)
68      {
69        printf("Can't join thread %u, status = %d\n", i, status);
70        exit(ERROR_JOIN_THREAD);
71      }
72    }
73
74    delete [] MulH;
75    delete [] MulV;
76    delete [] threads;
77
78    return C;
79  }
```

Листинг 3.4: Функции потоков инициализации векторов

```
1  void *init_MultH(void *_args)
2  {
3    InitVectorArgs *args = (InitVectorArgs*) _args;
4
5    const unsigned M = args->A.get_rows();
6    *(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 3.5: Функция потока вычисления строки при чётных размерах матрицы

```
void *multiply_even(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]) *
                (args->A[i][k + 1] + args->B[k][j]);
      }
    }

  args->i += 1;
  args->start_i = args->end_i;
```

```
23   if (args->i + 1 != args->amount)
24     args->end_i += args->step;
25   else
26     args->end_i += args->step + args->left;
27
28   return NULL;
29 }
```

Листинг 3.6: Функция потока вычисления строки при нечётных размерах матрицы

```
1  void *multiply_odd(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   args->i += 1;
20   args->start_i = args->end_i;
21   if (args->i + 1 != args->amount)
22     args->end_i += args->step;
23   else
24     args->end_i += args->step + args->left;
25
26   return NULL;
27 }
```

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. Результаты измерений показаны в таблице 4.2 и на рисунке 4.2.

Таблица 4.1: Время работы алгоритма при чётных размерах матриц

Размер матриц	Время в тиках при			
	2 потоках	4 потоках	6 потоках	8 потоках
100	1079621	1336500	1429639	1719100
200	1868640	2118858	2696051	2867224
300	4801348	3382294	3673955	4149612
400	4280742	5421391	5855953	7389256
500	8600721	7667042	7736335	6953143
600	6605085	7658315	7205055	7808372
700	8685191	8946213	9252224	10249792
800	10322456	10779327	11839892	12299641
900	11318034	11642143	11896053	12503616
1000	12767219	13568252	13403517	14974970

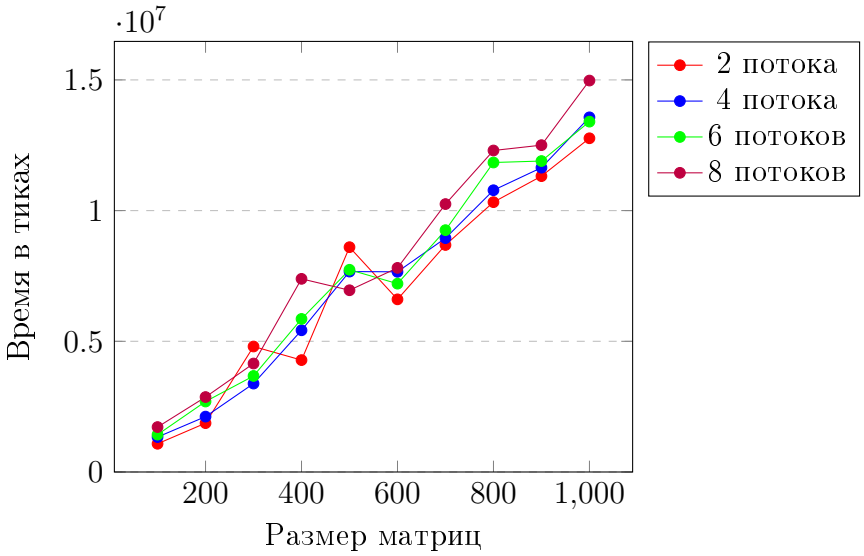

Рис. 4.2: График времени работы алгоритмов при чётных размерах матриц

...

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. Результаты измерений показаны в таблице **??** и на рисунке 4.3.

Таблица 4.2: Время работы алгоритма при нечётных размерах матриц

Размер матриц	Время в тиках при			
	2 потоках	4 потоках	6 потоках	8 потоках
101	1103200	1287671	1425753	1716967
201	1822043	2195514	2509995	2814434
301	2696725	2909712	3381560	4780288
401	3541121	4622769	5165189	4283757
501	4201023	4842153	5092641	5552390
601	4836585	5219435	5470350	5868582
701	6515748	6366694	6722863	6873247
801	6855674	7103331	7572905	8032646
901	9265517	9535723	10160023	10399878
1001	10725876	10200885	10670903	10456370

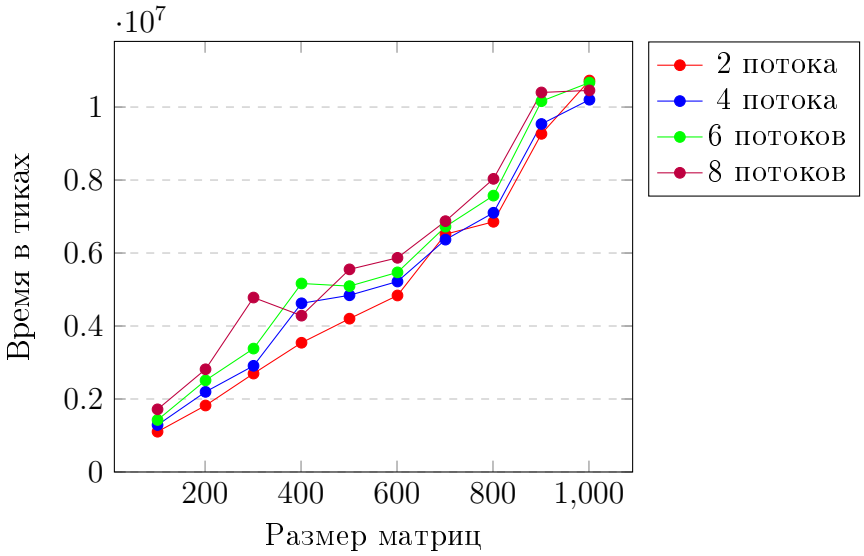

Рис. 4.3: График времени работы алгоритмов при нечётных размерах матриц

...

Заключение

...

Литература

[1] Бахвалов, Н.С. Численные методы / Н.С. Бахвалов, Н.П. Жидков, Г.М. Кобельков – М.: Наука, 1987.

[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] Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р.М. Ривест: – МЦНТО, 1999.

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