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»

Отчёт

по лабораторной работе № 4

Название: Параллельное умножение матриц

Дисциплина: Анализ алгоритмов

Студент	ИУ7-55Б		Д.В. Сусликов
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподователь			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Содержание

Введение

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

В ходе лабораторной работы требуется:

1) выбрать алгоритм для рассмотрения;
2) описать стандартную версию;
3) реализовать 2 параллельные версии;
4) запустить эксперименты на каждой при различном числе потоков 1,2,4,8 ...4M, где M - количество логических ядер на компьютере;
5) проверить, всегда ли при росте потоков, время работы снижается;

В данной лабораторной работе был выбран алгоритм Винограда. Необходимо сравнить зависимость времени работы алгоритма от числа параллельных потоков и размера матриц, провести сравнение стандартного и параллельного алгоритмов.

1 Аналитический раздел

В данном разделе представлены математические описания стандартного и параллельного алгоритмов Винограда.

1.1 Общая информация

Матрица - математический объект, эквивалентный двумерному массиву. Числа располагаются в матрице по строкам и столбцам. Две матрицы одинакового размера можно поэлементно сложить или вычесть друг из друга. 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.

Умножение матриц — одна из основных операций над матрицами. Матрица, получаемая в результате операции умножения, называется произведением матриц.

Пусть даны две прямоугольные матрицы A и B размеров $[m * n]$ и $[n * k]$ соответственно. В результате произведение матриц A и B получим матрицу C размера $[m * k]$.

$$c_{i,j} = \sum_{r=1}^{m} a_{ir} b_{rj} \qquad (i = 1, 2, ...l; j = 1, 2, ...n) \tag{1}$$

Операция умножения двух матриц выполнима только в том случае, если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить.

1.2 Алгоритм Винограда

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. [1]

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:

$$V * W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4 \tag{2}$$

Это равенство можно переписать в виде:

$$V * W = (v_1 + w_2)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4 \tag{3}$$

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. На практике это означает, что над предварительно обработанными элементами придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения[4].

1.3 Параллельные вычисления

Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций. Теория параллельных вычислений составляет раздел прикладной теории алгоритмов[5].

Вывод

По итогу, были разобраны общая информация о матрицах и их умножении, алгоритм Винограда и суть параллельных вычислений.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов, а так же описано, каким образом будет распараллелен алгоритм Винограда.

2.1 Схемы алгоритмов

Ниже на Рисунке 1 представлена схема стандартного алгоритма умножения Винограда.

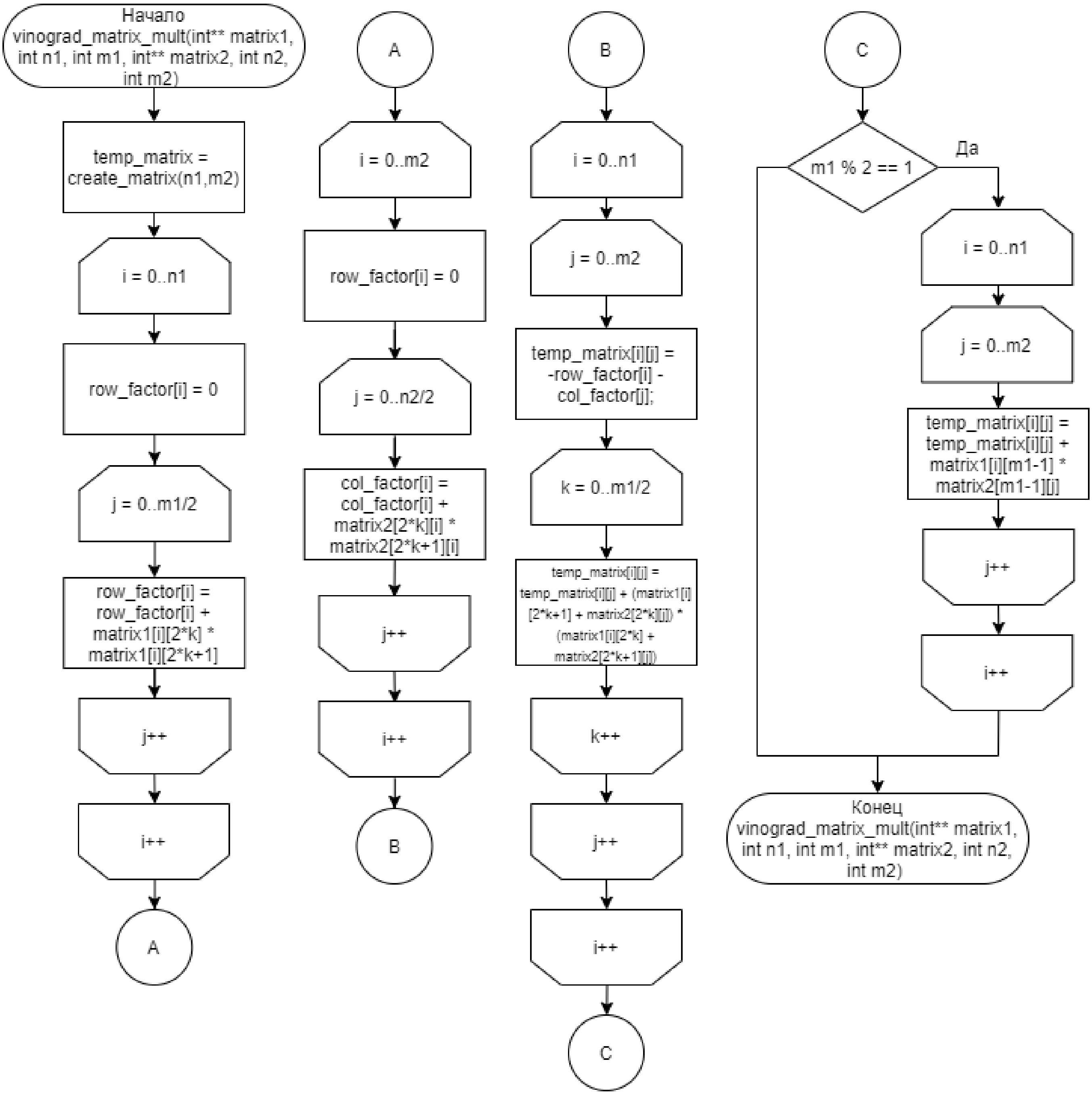

Рисунок 1 - Схема стандартного алгоритма умножения Винограда

2.2 Распараллеливание алгоритма умножения Винограда

В данной лабораторной работе будет рассмотрено две реализации алгоритма умножения Винограда:

1) распараллеливание части кода, где происходит заполнение векторов row_factor и col_factor;

2) распараллеливание части кода, где происходит заполнение результирующей матрицы.

Вывод

Таким образом, были рассмотрены схема алгоритма умножения Винограда и реализации его распараллеливания.

3 Технологическая часть

В данном разделе даны общие требования к программе, средства реализации и реализация алгоритмов.

3.1 Общие требования

Требования к вводу:

1) вводятся размеры матриц;

2) вводятся (или автоматически генерируются) матрицы.

Требования к программе:

1) при вводе неправильных размеров матриц программа не должна завершаться аварийно;

2) должно выполняться корректное умножение матриц.

3.2 Средства реализации

В лабораторной работе был использован язык C++[1], так как он известен, и на нём было написано множество предыдущих работ.

Среда разработки - Qt[2].

Для замеров процессорного времени была использована функция $clock()$[3].

3.3 Реализация алгоритмов

В Листинге 1 показана реализация стандартного алгоритма умножения Винограда.

Листинг 1 - Стандартный алгоритм умножения Винограда

```
void vinograd_matrix_mult(int** matrix1, int row1, int col1,\
int** matrix2, int row2, int col2)
{
  if ((col1 != row2) || row1 == 0 || row2 == 0)
  {
    cout << "Incorrect matrixes" << endl;
    return;
  }

  int** temp_matrix = create_matrix(row1, col2);

  int row_factor[row1];
  for (int i = 0; i < row1; i++)
  {
    row_factor[i] = 0;
    for (int k = 0; k < col1 / 2; k++)
    row_factor[i] = row_factor[i] + matrix1[i][2 * k] * matrix1[
        i][2 * k + 1];
  }

  int col_factor[col2];
  for (int i = 0; i < col2; i++)
  {
    col_factor[i] = 0;
    for (int k = 0; k < row2 / 2; k++)
    col_factor[i] = col_factor[i] + matrix2[2 * k][i] * matrix2
        [2 * k + 1][i];
  }

```

```
    for (int i = 0; i < row1; i++)
    {
      for (int j = 0; j < col2; j++)
      {
        temp_matrix[i][j] = -row_factor[i] - col_factor[j];
        for (int k = 0; k < col1 / 2; k++)
        temp_matrix[i][j] = temp_matrix[i][j] + (matrix1[i][2 * k + 1] + matrix2[2 * k][j])
        * (matrix1[i][2 * k] + matrix2[2 * k + 1][j]);
      }
    }

    if (col1 % 2 == 1)
    {
      for (int i = 0; i < row1; i++)
      {
        for (int j = 0; j < col2; j++)
        temp_matrix[i][j] = temp_matrix[i][j] + matrix1[i][col1 - 1] * matrix2[col1 - 1][j];
      }
    }

    print_matrix(temp_matrix, row1, col2);
    delete_matrix(temp_matrix, col2);
  }
```

В Листинге 2 и Листинге 3 представлены параллельные реализации алгоритма Винограда.

Листинг 2 - Алгоритм Винограда с распараллеленным заполнением векторов

```
void vinograd_matrix_mult_parallel(int** matrix1, int row1, int
   col1,\
int** matrix2, int row2, int col2, int threads_amount)
{
  if ((col1 != row2) || row1 == 0 || row2 == 0)
  {
    cout << "Incorrect matrixes" << endl;
    return;
  }

  int** temp_matrix = create_matrix(row1, col2);

  int* row_factor = (int*)calloc(row1, sizeof(int));
  thread* threads = new thread[threads_amount];

  int rows_for_thread = row1 / threads_amount;
  int start_row = 0;
  for (int i = 0; i < threads_amount; i++)
  {
    int end_row = start_row + rows_for_thread;
    if (i == threads_amount - 1)
    end_row = row1;

    threads[i] = thread(thread_row_mult, matrix1, col1,
       row_factor, start_row, end_row);
    start_row = end_row;
  }

  for (int i = 0; i < threads_amount; i++)
  threads[i].join();
```

```
    int* col_factor = (int*)calloc(col2, sizeof(int));

    int columns_for_thread = col2 / threads_amount;
    int start_column = 0;
    for (int i = 0; i < threads_amount; i++)
    {
      int end_column = start_column + columns_for_thread;
      if (i == threads_amount - 1)
      end_column = col2;

      threads[i] = thread(thread_columns_mult, matrix2, row2, col_factor,\
      start_column, end_column);
      start_column = end_column;
    }

    for (int i = 0; i < threads_amount; i++)
    threads[i].join();

    for (int i = 0; i < row1; i++)
    {
      for (int j = 0; j < col2; j++)
      {
        temp_matrix[i][j] = -row_factor[i] - col_factor[j];
        for (int k = 0; k < col1 / 2; k++)
        temp_matrix[i][j] = temp_matrix[i][j] + (matrix1[i][2 * k + 1] + matrix2[2 * k][j])
        * (matrix1[i][2 * k] + matrix2[2 * k + 1][j]);
      }
    }

    if (col1 % 2 == 1)
    {
      for (int i = 0; i < row1; i++)
```

```
        {
          for (int j = 0; j < col2; j++)
          temp_matrix[i][j] = temp_matrix[i][j] + matrix1[i][col1 - 1] * matrix2[col1 - 1][j];
        }
      }

      print_matrix(temp_matrix, row1, col2);
      free(row_factor);
      free(col_factor);
      delete_matrix(temp_matrix, col2);
    }

    void thread_row_mult(int** matrix1, int columns, int* row_factor, int start_row, int end_row)
    {
      for (int i = start_row; i < end_row; i++)
      {
        for (int j = 0; j < columns / 2; j++)
        row_factor[i] += matrix1[i][2 * j] * matrix1[i][2 * j + 1];
      }
    }

    void thread_columns_mult(int** matrix2, int rows, int* col_factor,\
    int start_column, int end_column)
    {
      for (int i = start_column; i < end_column; i++)
      {
        for (int j = 0; j < rows / 2; j++)
        col_factor[i] += matrix2[2 * j][i] * matrix2[2 * j + 1][i];
      }
    }
```

Листинг 3 - Распараллеленное заполнение результирующей матрицы.

```
void vinograd_matrix_mult_parallel2(int** matrix1, int row1, int
    col1,\
int** matrix2, int row2, int col2, int threads_amount)
{
  if ((col1 != row2) || row1 == 0 || row2 == 0)
  {
    cout << "Incorrect matrixes" << endl;
    return;
  }

  int** temp_matrix = create_matrix(row1, col2);

  int* row_factor = (int*)calloc(row1, sizeof(int));
  for (int i = 0; i < row1; i++)
  {
    row_factor[i] = 0;
    for (int k = 0; k < col1 / 2; k++)
    row_factor[i] = row_factor[i] + matrix1[i][2 * k] * matrix1[
       i][2 * k + 1];
  }

  int* col_factor = (int*)calloc(col2, sizeof(int));
  for (int i = 0; i < col2; i++)
  {
    col_factor[i] = 0;
    for (int k = 0; k < row2 / 2; k++)
    col_factor[i] = col_factor[i] + matrix2[2 * k][i] * matrix2
       [2 * k + 1][i];
  }


  thread* threads = new thread[threads_amount];

  int rows_for_thread = row1 / threads_amount;
```

```
    int start_row = 0;
    for (int i = 0; i < threads_amount; i++)
    {
      int end_row = start_row + rows_for_thread;
      if (i == threads_amount - 1)
      end_row = row1;

      threads[i] = thread(thread_cycle, matrix1, col1, matrix2,
         col2,\
      temp_matrix, row_factor, col_factor, start_row, end_row);
      start_row = end_row;
    }

    for (int i = 0; i < threads_amount; i++)
    threads[i].join();

    if (col1 % 2 == 1)
    {
      for (int i = 0; i < row1; i++)
      {
        for (int j = 0; j < col2; j++)
        temp_matrix[i][j] = temp_matrix[i][j] + matrix1[i][col1 -
           1] * matrix2[col1 - 1][j];
      }
    }

    print_matrix(temp_matrix, row1, col2);
    free(row_factor);
    free(col_factor);
    delete_matrix(temp_matrix, col2);
  }

  void thread_cycle(int** matrix1, int col1, int** matrix2, int
     col2,\
  int** temp_matrix, int* row_factor, int* col_factor, int
```

```
        start_row, int end_row)
    {
        for (int i = start_row; i < end_row; i++)
        {
            for (int j = 0; j < col2; j++)
            {
                temp_matrix[i][j] = -row_factor[i] - col_factor[j];
                for (int k = 0; k < col1 / 2; k++)
                temp_matrix[i][j] = temp_matrix[i][j] + (matrix1[i][2 * k
                    + 1] + matrix2[2 * k][j])
                * (matrix1[i][2 * k] + matrix2[2 * k + 1][j]);
            }
        }
    }
```

Вывод

Таким образом, были разобраны требования у программе, описаны средства реализации, реализованы стандартный и 2 распараллеленые алгоритмы умножения Винограда.

4 Экспериментальный раздел

В данном разделе представлены результаты работы программы и приведен анализ времени работы каждого из алгоритмов.

4.1 Примеры работы программы

На Рисунке 2 представлены меню и ввод матриц.

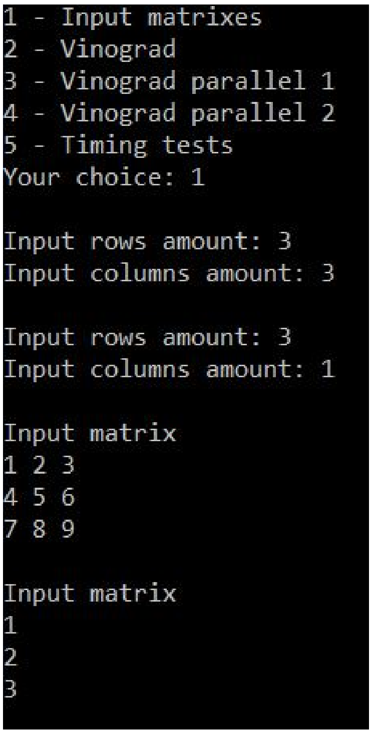

Рисунок 2 - Меню выбора и ввод матриц

На Рисунке 3 можно увидеть пример работы всех алгоритмов

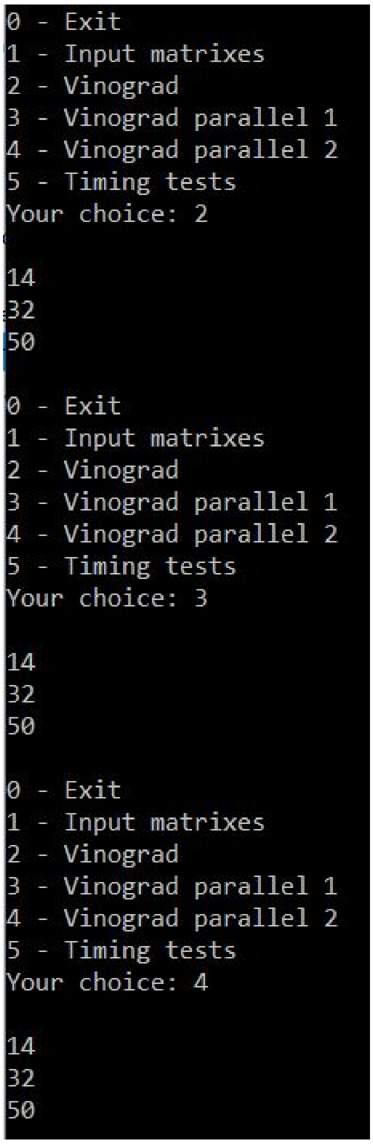

Рисунок 3 - Пример работы всех алгоритмов

4.2 Технические характеристики ПК

Характеристики:

1) операционная система - Windows 10 (64-bit);
2) оперативная память - 16 Гб;
3) процессор Intel® Core™ i7-6700K 4ГГц;
4) количество ядер - 4 (логических - 8);

4.3 Анализ времени работы алгоритмов

Эксперименты проводятся на матрицах разных размеров и при разном количестве поток. Элементы матрицы заполняются произвольно.

В первом случае берутся квадратные матрицы с размерами 100x100, 101x101, 200x200, 201x201, 300x300, 301x301, 400x400, 401x401, 500x500, 501x501. Количество потоков неизменно и равно 4. Графики зависимости времени работы алгоритмов от размеров матрицы изображены на Рисунке 4.

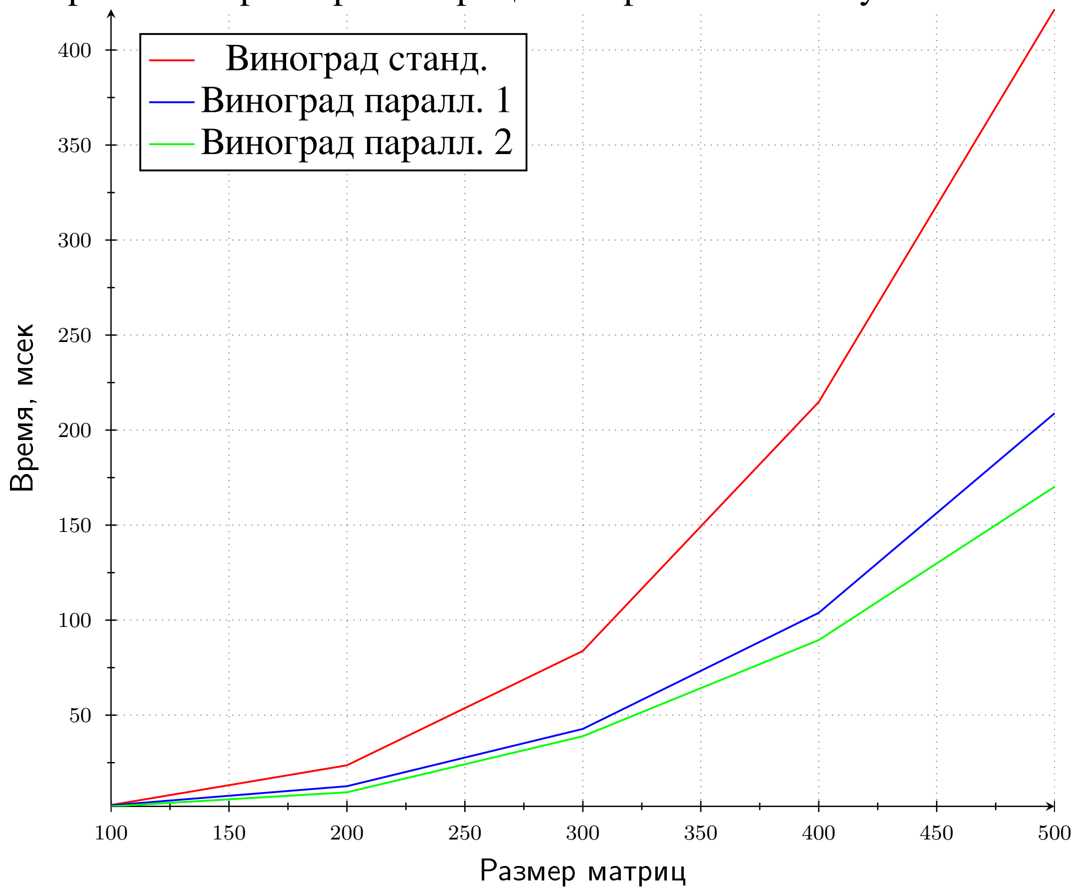

Рисунок 4 - Графики зависимости времени работы алгоритмов от размеров матриц

Во втором случае берутся матрицы одного размера 100x100, но при разном количестве потоков. Графики зависимости времени работы алгоритмов от количества потоков изображены на Рисунке 5.

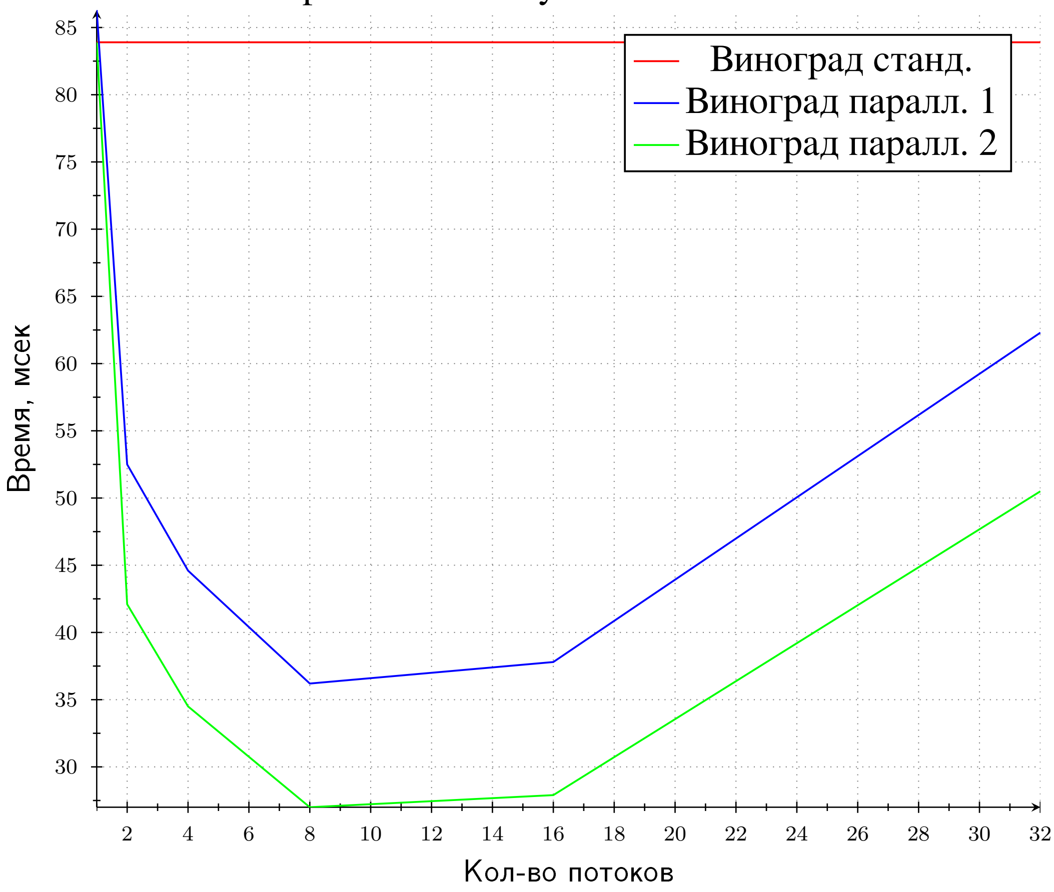

Рисунок 5 - Графики зависимости времени работы алгоритмов от количества потоков

Вывод

Результаты экспериментов показывают, что с увеличением числа потоков до количества логических ядер уменьшается время работы распараллеленных алгоритмов. Если число потоков становится больше числа логических ядер, то скорость работы замедляется. Линейная реализация работает с одинаковым результатом, так как не зависит от количества потоков. При выделении лишь одного потока распараллеленные алгоритмы работают медленнее, так как на создание и объединение потока тратится лишнее время.

Учитывая все ранее написанное, можно сделать вывод, что алгоритм с распараллеленным циклом подсчета результирующей матриц является наиболее эффективным.

Заключение

В ходе выполнения лабораторной работы были изучены возможности параллельных вычислений и применены на примере алгоритма умножения матриц Винограда. Были описаны схемы алгоритмов. Также было произведено сравнение по времени работы алгоритмов, в результате которого стало известно, что самой эффективной по времени реализацией оказалась та, в которой было произведено распараллеливание цикла подсчета результирующей матрицы. Данная реализация быстрее линейной в 2.4 раза и быстрее реализации с распараллеленным подсчетом векторов в 1.2 раза.

Цель работы достигнута, все поставленные задачи выполнены.

Литература

1) Бьерн Страуструп. Язык программирования C++. -URL:

   https://codernet.ru/books/c_plus/bern_straustrup_yazyk_programmirovaniya_c_specialnoe_izdanie/

   (дата обращения: 01.10.2020). Текст: электронный.

2) Qt. -URL:

   https://www.qt.io/ (дата обращения: 01.10.2020). Текст: электронный.

3) Функция $clock$. -URL:

   https://docs.microsoft.com/ru-ru/cpp/c-runtime-library/reference/clock?view=vs-2019 (дата обращения: 01.10.2020). Текст: электронный.

4) Дж. Макконнелл. Основы современных алгоритмов.
   2-е дополненное издание
   Москва: Техносфера, 2004. - 368с. ISBN 5-94836-005-9
   с. 130 - 133

5) Параллельные вычисления -URL:

   https://ru.bmstu.wiki/ (дата обращения: 13.11.2020). Текст: электронный.