МГТУ им. Баумана

Лабораторная работа №4

По курсу: "Анализ алгоритмов"

Параллельное умножение матриц

Работу выполнил: Гаврилов Дмитрий, ИУ7-56Б

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

Москва, 2019

Оглавление

Введение

Цель работы: изучение алгоритмов умножения матриц. В данной лабораторной работе рассматривается стандартный алгоритм умножения матриц, алгоритм Винограда и модифицированный алгоритм Винограда. Также требуется изучить рассчет сложности алгоритмов, получить навыки в улучшении алгоритмов. Эти алгоритмы активно применяются во всех областях, применяющих линейную алгебру, таких как:

- компьютерная графика
- физика
- экономика

В ходе лабораторной работы предстоит:

- Изучение и реализация параллельного алгоритма Винограда для умножения матриц
- Сравнить зависимость времени работы алгоритма от числа параллельных потоков исполнения и размера матриц
- Провести сравнение стандартного и параллельного алгоритма.

1 Аналитическая часть

Матрица - математический объект, эквивалентный двумерному массиву. Числа располагаются в матрице по строкам и столбцам. 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.

1.1 Алгоритм Винограда

Рассмотрим два вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$. 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.

1.2 Параллельный алгоритм Винограда

Трудоемкость алгоритма Винограда имеет сложность $O(nmk)$ для умножения матриц $n1 \times m1$ на $n2 \times m2$. Чтобы улучшить алгоритм, следу-

ет распараллелить ту часть алгоритма, которая содержит 3 вложенных цикла.

Вычисление результата для каждой строки не зависит от результата выполнения умножения для других строк. Поэтому можно распараллелить часть кода, где происходят эти действия. Каждый поток будет выполнять вычисления определенных строк результирующей матрицы.

1.3 Распараллеливание задачи

В рамках данной лабораторной работы производилось распараллеливание задачи по потокам. В CPU для данной цели используются treads.

CPU – central processing unit – это универсальный процессор, также именуемый процессором общего назначения. Он оптимизирован для достижения высокой производительности единственного потока команд. Доступ к памяти с данными и инструкциями происходит преимущественно случайным образом. Для того, чтобы повысить производительность CPU еще больше, они проектируются специально таким образом, чтобы выполнять как можно больше инструкций параллельно. Например для этого в ядрах процессора используется блок внеочередного выполнения команд. Но несмотря на это, CPU все равно не в состоянии осуществить параллельное выполнение большого числа инструкций, так как расходы на распараллеливание инструкций внутри ядра оказываются очень существенными. Именно поэтому процессоры общего назначения имеют не очень большое количество исполнительных блоков.

1.3.1 Вывод

Были рассмотрены поверхностно стандартная и параллельная реализации алгоритма Винограда.

2 | Конструкторская часть

Требования к вводу:
На вход подаются две матрицы и их размерности
Требования к программе:
Корректное умножение двух матриц

2.1 Схемы алгоритмов

На рис. 2.3 представлена схема стандартного алгоритма Винограда:

На рис. 2.3 представлена схема распараллеленого алгоритма Винограда:

На рис. 2.3 ппредставлена схема фнкции параллельного вычисления матрицы:

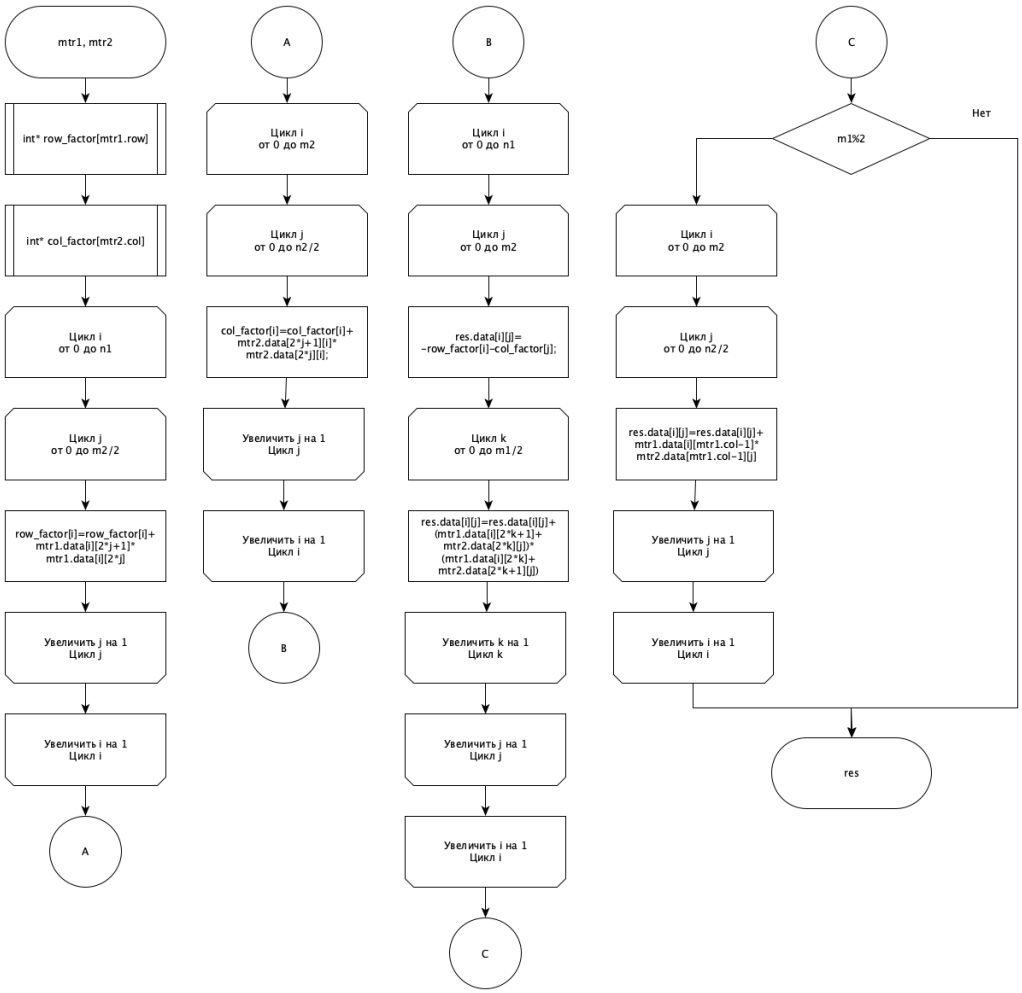

Рис. 2.1: Схема стандартного алгоритма Винограда

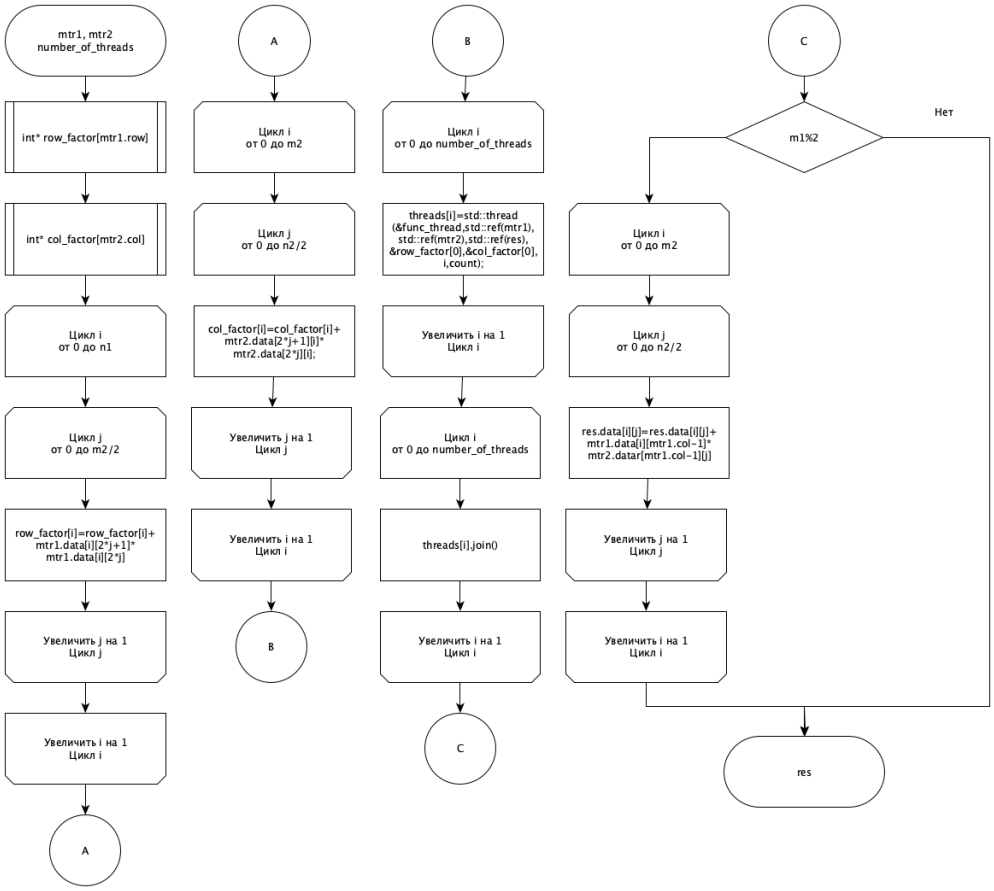

Рис. 2.2: Схема распараллеленого алгоритма Винограда

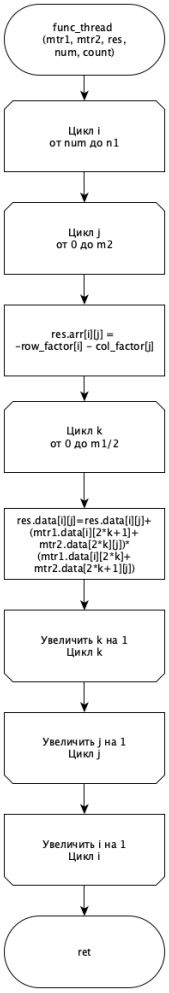

Рис. 2.3: Схема распараллеленого алгоритма Винограда

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрал язык программирования Java, так имею большой опыт работы с ним. Среда разработки - IntelliJ IDEA.

Для работы с потоками используется библиотека поддержки потоков java.util.concurrent.
Для замера времени используется системный вызов System.currentTimeMillis, возвращающий количество миллисекунд.

3.2 Реализация алгоритма

Листинг 3.1: Стандартный алгоритм Винограда

```
1  public class MultiplyVinograd implements MatrixMultiplier {
2      @Override
3      public int[][] multiply(int[][] firstMatrix, int[][]
           secondMatrix) {
4          int firstRowCount = firstMatrix.length;
5          int secondRowCount = secondMatrix.length;
6
7          if (firstRowCount == 0 || secondRowCount == 0)
8              return null;
9
10         int firstColumnCount = firstMatrix[0].length;
11         int secondColumnCount = secondMatrix[0].length;
12
```

```
13          if (firstColumnCount != secondRowCount)
14              return null;
15
16          int[] rowFactors = new int[firstRowCount];
17          int[] columnFactors = new int[secondColumnCount];
18
19          int[][] result = new int[firstRowCount][
                secondColumnCount];
20
21          for (int i = 0; i < firstRowCount; i++) {
22              for (int j = 0; j < firstColumnCount / 2; j++)
                    {
23                  rowFactors[i] += firstMatrix[i][j * 2] *
                        firstMatrix[i][j * 2 + 1];
24              }
25          }
26
27          for (int i = 0; i < secondColumnCount; i++) {
28              for (int j = 0; j < secondRowCount / 2; j++) {
29                  columnFactors[i] += secondMatrix[j * 2][i]
                        * secondMatrix[j * 2 + 1][i];
30              }
31          }
32
33          for (int i = 0; i < firstRowCount; i++) {
34              for (int j = 0; j < secondColumnCount; j++) {
35                  result[i][j] = -rowFactors[i] -
                        columnFactors[j];
36                  for (int k = 0; k < firstColumnCount / 2; k
                        ++) {
37                      result[i][j] += (firstMatrix[i][2 * k +
                            1] + secondMatrix[2 * k][j]) * (
                            firstMatrix[i][2 * k] + secondMatrix
                            [2 * k + 1][j]);
38                  }
39              }
40          }
41
42          if (firstColumnCount \% 2 == 1) {
43              for (int i = 0; i < firstRowCount; i++) {
```

```
44                  for (int j = 0; j < secondColumnCount; j++)
                        {
45                      result[i][j] += firstMatrix[i][
                            firstColumnCount - 1] * secondMatrix
                            [firstColumnCount - 1][j];
46                  }
47              }
48          }
49
50          return result;
51      }
52  }
```

Можно заметить, что вычисление результата для каждой строки происходит независимо от результата выполнения умножения для других строк. Поэтому возможно распараллелить участок кода, соответстующий строкам 33-40 листинга 3.1. Каждый поток будет выполнять вычисление некоторых строк результирующей матрицы. Это сделано потому, что проход по строкам матрицы является более эффективным с точки зрения организации данных в памяти.

В листинге 3.2 представлен параллельный алгоритм Винограда:

Листинг 3.2: Параллельный алгоритм Винограда

```
1  public class MultiplyVinogradParralel{
2      public int[][] multiplyParralel(int[][] firstMatrix,
           int[][] secondMatrix, int threadsCount) throws
           InterruptedException {
3          int firstRowCount = firstMatrix.length;
4          int secondRowCount = secondMatrix.length;
5
6          if (firstRowCount == 0 || secondRowCount == 0) {
7              return null;
8          }
9
10         int firstColumnCount = firstMatrix[0].length;
11         int secondColumnCount = secondMatrix[0].length;
12
13         if (firstColumnCount != secondRowCount) {
14             return null;
```

```
15          }
16
17          int[] rowFactors = new int[firstRowCount];
18          int[] columnFactors = new int[secondColumnCount];
19
20          int[][] result = new int[firstRowCount][
                secondColumnCount];
21
22          for (int i = 0; i < firstRowCount; i++) {
23              for (int j = 0; j < firstColumnCount / 2; j++)
                    {
24                  rowFactors[i] += firstMatrix[i][j * 2] *
                        firstMatrix[i][j * 2 + 1];
25              }
26          }
27
28          for (int i = 0; i < secondColumnCount; i++) {
29              for (int j = 0; j < secondRowCount / 2; j++) {
30                  columnFactors[i] += secondMatrix[j * 2][i]
                        * secondMatrix[j * 2 + 1][i];
31              }
32          }
33
34          ExecutorService service = Executors.
                newFixedThreadPool(threadsCount);
35
36
37          for (int i = 0; i < threadsCount; i++) {
38              service.execute(new LoopExecutor(
39                      firstMatrix,
40                      secondMatrix,
41                      result,
42                      rowFactors,
43                      columnFactors,
44                      i,
45                      threadsCount));
46          }
47          service.shutdown();
48
49          while (!service.awaitTermination(24L, TimeUnit.
```

```
            HOURS)) {
50              System.out.println("Not yet. Still waiting for
                    termination");
51          }
52
53
54          if (firstColumnCount \% 2 == 1) {
55              for (int i = 0; i < firstRowCount; i++) {
56                  for (int j = 0; j < secondColumnCount; j++)
                        {
57                      result[i][j] += firstMatrix[i][
                            firstColumnCount - 1] * secondMatrix
                            [firstColumnCount - 1][j];
58                  }
59              }
60          }
61
62          return result;
63      }
64  }
```

В листинге 3.3 представлено распараллеленное вычисление тройного цикла:

Листинг 3.3: Распараллеленный тройной цикл

```
1   class LoopExecutor implements Runnable {
2           private int[][] firstMatrix;
3           private int[][] secondMatrix;
4           private int[][] result;
5           private int[] rowFactors;
6           private int[] colFactors;
7           private int startIndex;
8           private int step;
9
10          public LoopExecutor(int[][] firstMatrix, int[][]
                secondMatrix, int[][] result, int[] rowFactors,
                int[] colFactors, int startIndex, int step) {
11              this.firstMatrix = firstMatrix;
12              this.secondMatrix = secondMatrix;
13              this.result = result;
14              this.rowFactors = rowFactors;
```

```
15              this.colFactors = colFactors;
16              this.startIndex = startIndex;
17              this.step = step;
18          }
19
20          @Override
21          public void run() {
22              for (int i = startIndex; i < firstMatrix.length
                    ; i+=step) {
23                  for (int j = 0; j < secondMatrix[0].length;
                        j++) {
24                      result[i][j] = -rowFactors[i] -
                            colFactors[j];
25                      for (int k = 0; k < firstMatrix[0].
                            length / 2; k++) {
26                          result[i][j] += (firstMatrix[i][2 *
                                k + 1] + secondMatrix[2 * k][j
                                ]) * (firstMatrix[i][2 * k] +
                                secondMatrix[2 * k + 1][j]);
27                      }
28                  }
29              }
30          }
31      }
```

4 | Исследовательская часть

4.1 Постановка эксперимента

Были проведены исследования зависимости времени работы трех алгоритмов от размеров перемножаемых матриц и количества использованных потоков. Замеры времени проводились для матриц четной размерности размером от 100 до 1000 с шагом 100 и матриц нечетной размерности размером от 101 до 1001 с шагом 100. Количество потоков - от 1 до 16, т.к. компьютер, на котором проводились вычисления, содержит 4 логических ядра.

Временные замеры проводятся путём многократного проведения эксперимента и деления результирующего времени на количество итераций эксперимента.

4.2 Сравнительный анализ на основе замеров времени работы алгоритмов

На рис. 4.1 представлен график времени работы алгоритма на матрицах четной размерности:

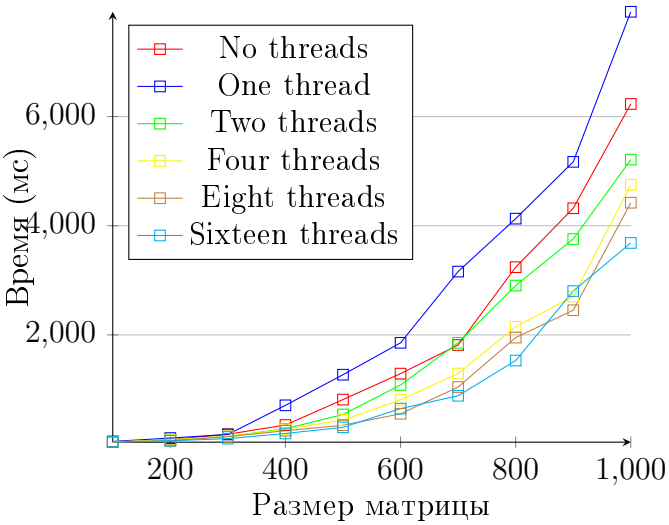

Рис. 4.1: Сравнение времени на матрицах чётной размерности

На рис. 4.2 представлен график времени работы алгоритма на матрицах четной размерности:

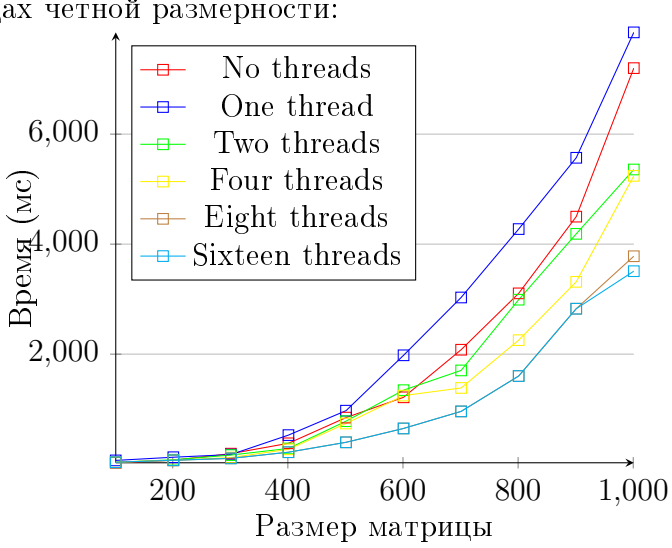

Рис. 4.2: Сравнение времени на матрицах нечётной размерности

4.3 Вывод

Эксперименты замера времени показали, что при последовательной и параллельной (с одним рабочим потоком) реализациях оптимизированного алгоритма Винограда совсем немного выигрывает последовательная реализация (в ней не тратится время на выделение рабочего потока).

При сравнении замеров времени для параллельной реализации алгоритма с 1, 2, 4, 8 и 16 рабочими потоками выяснилось, что максимальная производительность достигается на 8-ми и 16-и рабочих потоках. Выполнение алгоритма на 8-ми рабочих потоках быстрее в 3,82 раз, по сравнению с выполнением на 1-м потоке для матриц размера 1000×1000.

Заключение

В ходе работы был изучен параллельный алгоритм умножения матриц: алгоритм Винограда. Выполнено сравнение зависимости всех рассматриваемых алгоритмов от числа параллельных потоков и размера матриц. В ходе исследования было установлено, что многопоточный алгоритм Винограда выполняется быстрее, чем стандартный алгоритм.