Министерство образования Российской Федерации
Московский Государственный Технический
Университет им. Н.Э. Баумана

Отчет по лабораторной работе №4
По курсу «Анализ алгоритмов»

Тема: «Многопоточное умножение матриц»

Студент: **Бадалян Д.А.**
Группа: **ИУ7-51**

Преподаватель: **Волкова Л.Л.**

Москва, 2017

Постановка задачи

В ходе лабораторной работы предстоит:

1. реализовать алгоритм умножения матриц средствами N потоков;
2. сравнить алгоритм многопоточного умножения матриц с однпоточным;
3. провести эксперименты с замерами времени на матрицах разных размерностей;
4. сделать выводы об эффективности многопоточного умножения матриц.

Алгоритмы

Последовательный базовый алгоритм

Листинг 1: Последовательный базовый алгоритм умножения матриц

```
1  private static int[][] multiplyStd(int[][] a, int[][] b)
2  {
3      int rowCount = a.length;
4      int colCount = b[0].length;
5      int sumLength = b.length;
6      int[][] result = new int[rowCount][colCount];
7
8      for (int i = 0; i < rowCount; ++i) {
9          for (int j = 0; j < colCount; ++j) {
10             int sum = 0;
11             for (int k = 0; k < sumLength; k++)
12                 sum += a[i][k] * b[k][j];
13             result[i][j] = sum;
14         }
15     }
16     return result;
17 }
```

Входные данные: a - первая матрица, b - вторая матрица
Выходные данные: $result$ - результирующая матрица

Последовательный алгоритм Винограда

Листинг 2: Последовательный алгоритм Винограда умножения матриц

```
1  private static int[][] multiplyVinogradStd(int[][] a, int
       [][] b) {
2          int d = a.length/2;
3          boolean flag = a.length % 2 == 1;
4          int m = a.length;
5          int n = b[0].length;
6          int[] rowFactor = new int[m];
7          int[] columnFactor = new int[n];
8          int[][] res = new int[m][n];
9          int buffer;
10
11         for (int i = 0; i < m; i++)
12         {
```

```
13              rowFactor[i] = a[i][0] * a[i][1];
14              for (int j = 1; j < d; j++)
15                  rowFactor[i] += a[i][2*j]*a[i][2*j+1];
16          }
17
18          for (int i = 0; i < n; i++)
19          {
20              columnFactor[i] = b[0][i] * b[1][i];
21              for (int j = 1; j < d; j++)
22                  columnFactor[i] += b[2*j][i] * b[2*j+1][i];
23          }
24
25          for (int i = 0; i < m; i++)
26              for (int j = 0; j < n; j++)
27              {
28                  buffer = (flag ? a[i][n-1] * b[n-1][j] : 0)
                        ;
29                  buffer -= rowFactor[i] + columnFactor[j];
30                  for (int k = 0; k < d; k++)
31                      buffer += (a[i][2*k] + b[2*k+1][j])*(a[
                            i][2*k+1] + b[2*k][j]);
32                  res[i][j] = buffer;
33              }
34
35          return res;
36      }
```

Входные данные: a - первая матрица, b - вторая матрица
Выходные данные: res - результирующая матрица

В процессорах Core i3, Core i7 и некоторых Core i5 была реализована технология **hyper-threading**. При включении технологии каждое физическое ядро процессора определяется операционной системой как два логических ядра.

Преимуществами HTT считаются:

- возможность запуска нескольких потоков одновременно (многопоточный код);
- уменьшение времени отклика;
- увеличение числа пользователей, обслуживаемых сервером.

Количество логических ядер на компьютере, на котором выполнялась лабораторная работа, равняется 4.

Параллельный базовый алгоритм

В каждом потоке заполняются $\frac{n}{N}$ строк результирующей матрицы, где n - размерность матрицы (количество ее строк), а N - количество используемых потоков.

Листинг 3: Параллельный базовый алгоритм умножения матриц

```
1  private static int[][] multiply(int[][] a, int[][] b, int
       countOfThreads)
2  {
3      int resRow = a.length;
4      int resCol = b[0].length;
5      int[][] res = new int[resRow][resCol];
6      MultThread[] threads = new MultThread[countOfThreads];
7      int rowsForThread = resRow/countOfThreads;
8      int firstInd = 0;
9      for (int i = 0; i < countOfThreads; i++) {
10         int lastInd = firstInd + rowsForThread;
11         if (i == countOfThreads-1)
12             lastInd = resRow;
13
14         threads[i] = new MultThread(a, b, res, firstInd,
               lastInd);
15         threads[i].start();
16         firstInd = lastInd;
```

```
17      }
18      try {
19          for (MultThread t : threads)
20              t.join();
21      }
22      catch (InterruptedException e) {
23          e.printStackTrace();
24      }
25      return res;
26  }
```

Входные данные: a, b - матрицы,
countOfThreads - количество потоков
Выходные данные: res - результирующая матрица

Класс MultThread - класс, наследованный от класса Thread, предназначенного для работы с потоками. Метод start() запускает поток на выполнение. Метод join() ожидает завершения потока. Метод getName() возвращает имя потока. В методе run() содержится выполняемый код.

Данные класса MultThread:
firstMatrix - первая матрица;
secondMatrix - вторая матрица;
result - результирующая матрица;
startIndex - с какой строки матрицы поток рассчитывает результат;
endIndex - до какой строки матрицы поток рассчитывает результат.

Листинг 4: Класс MultThread

```
1  class MultThread extends Thread {
2      private int[][] firstMatrix;
3      private int[][] secondMatrix;
4      private int[][] result;
5      private int startIndex;
6      private int endIndex;
7
8      MultThread(int[][] m1, int[][] m2, int[][] res, int
           startInd, int endInd) {
9          this.firstMatrix = m1;
10         this.secondMatrix = m2;
11         this.result = res;
12         this.startIndex = startInd;
13         this.endIndex = endInd;
```

```
14      }
15
16      public void run() {
17          System.out.println("Thread " + getName() + "
                started. (Calculate from "+ startIndex + " to "
               + endIndex + "rows)");
18          int colCount = secondMatrix[0].length;
19          for (int i = startIndex; i < endIndex; i++)
20              for (int j = 0; j < colCount; j++) {
21                  int sum = 0;
22                  for (int k = 0; k < secondMatrix.length; k
                        ++)
23                      sum += firstMatrix[i][k] * secondMatrix
                            [k][j];
24                  result[i][j] = sum;
25          }
26          System.out.println("Thread " + getName() + "
                finished.");
27      }
28  }
```

Параллельный алгоритм Винограда

В каждом потоке заполняются $\frac{n}{N}$ строк результирующей матрицы, где n - размерность матрицы (количество ее строк), а N - количество используемых потоков.

При этом векторы $rowVector$ и $columnVector$ заполняются до распараллеливания и передаются в конструктор для каждого из потоков.

Листинг 5: Параллельный алгоритм умножения матриц Винограда

```
1   private static int[][] multiplyThreadedVinograd(int[][] a,
        int[][] b, int countOfThreads) {
2           int d = a.length/2;
3           boolean flag = a.length % 2 == 1;
4           int m = a.length;
5           int n = b[0].length;
6           int[] rowFactor = new int[m];
7           int[] columnFactor = new int[n];
8           int[][] res = new int[m][n];
9
10          for (int i = 0; i < m; i++)
```

```
11          {
12              rowFactor[i] = a[i][0] * a[i][1];
13              for (int j = 1; j < d; j++)
14                  rowFactor[i] += a[i][2*j]*a[i][2*j+1];
15          }
16          for (int i = 0; i < n; i++)
17          {
18              columnFactor[i] = b[0][i] * b[1][i];
19              for (int j = 1; j < d; j++)
20                  columnFactor[i] += b[2*j][i] * b[2*j+1][i];
21          }
22
23          MultVinogradThread[] threads = new
               MultVinogradThread[countOfThreads];
24          int rowsForThread = m/countOfThreads;
25          int firstInd = 0;
26          for (int i = 0; i < countOfThreads; i++) {
27              int lastInd = firstInd + rowsForThread;
28              if (i == countOfThreads-1)
29                  lastInd = m;
30              threads[i] = new MultVinogradThread(a, b, res,
                   rowFactor,columnFactor ,firstInd, lastInd);
31              threads[i].start();
32              firstInd = lastInd;
33          }
34          try {
35              for (MultVinogradThread t : threads)
36                  t.join();
37          }
38          catch (InterruptedException e) {
39              e.printStackTrace();
40          }
41          return res;
42      }
```

Класс MultVinogradThread - класс, наследованный от класса Thread, предназначенного для работы с потоками.

Данные класса MultVinogradThread:
firstMatrix - первая матрица;
secondMatrix - вторая матрица;
result - результирующая матрица;
rowVector - массив с заранее посчитанными произведениями;

columnVector - массив с заранее посчитанными произведениями;
startIndex - с какой строки матрицы поток рассчитывает результат;
endIndex - до какой строки матрицы поток рассчитывает результат.

Листинг 6: Класс MultVinogradThread

```
1  class MultVinogradThread extends Thread {
2      private int[][] firstMatrix;
3      private int[][] secondMatrix;
4      private int[][] result;
5      private int[] rowVector;
6      private int[] columnVector;
7      private int startIndex;
8      private int endIndex;
9
10     MultVinogradThread(int[][] m1, int[][] m2, int[][] res,
           int[] rowVec, int[] colVec, int startInd, int
          endInd) {
11         this.firstMatrix = m1;
12         this.secondMatrix = m2;
13         this.result = res;
14         this.startIndex = startInd;
15         this.endIndex = endInd;
16         this.rowVector = rowVec;
17         this.columnVector = colVec;
18     }
19
20     public void run() {
21         System.out.println("Thread " + getName() + "
               started. (Calculate from "+ startIndex + " to "
              + endIndex + "rows)");
22         int colCount = secondMatrix[0].length;
23         int halfRowCount = firstMatrix.length/2;
24         int buffer;
25         boolean flag = firstMatrix.length % 2 == 1;
26         for (int i = startIndex; i < endIndex; i++)
27             for (int j = 0; j < colCount; j++)
28             {
29                 buffer = (flag ? firstMatrix[i][colCount-1]
                       * secondMatrix[colCount-1][j] : 0);
30                 buffer -= rowVector[i] + columnVector[j];
31                 for (int k = 0; k < halfRowCount; k++)
32                     buffer += (firstMatrix[i][2*k] +
```

```
                                secondMatrix[2*k+1][j])*(firstMatrix
                                [i][2*k+1] + secondMatrix[2*k][j]);
33                      result[i][j] = buffer;
34                  }
35              System.out.println("Thread " + getName() + "
                    finished.");
36          }
37  }
```

Эксперимент

В качестве эксперимента было замерено время работы алгоритмов умножения матриц размерностями от 100*100 до 1000*1000 с шагом 100 средствами 2, 4 и 8 потоков. А так же произведен замер времени работы однопоточного алгоритма. Полученные данные эксперимента приведены в таблице 1 и на графиках 1 и 2 (по оси абсцисс - размерность матриц, по оси ординат - время в миллисекундах):

Таблица 1. Результаты эксперимента

	Стандартный алгоритм, время (мсек)				**Алгоритм Винограда, время (мсек)**			
Потоки	1	2	4	8	1	2	4	8
100	3	2	1	1	4	3	1	1
200	15	12	10	11	18	13	10	11
300	72	47	36	25	76	47	32	35
400	214	130	102	99	194	109	75	72
500	364	202	159	144	268	176	124	111
600	427	278	200	202	325	201	135	103
700	766	595	426	412	566	476	325	301
800	1780	1302	998	791	801	622	531	499
900	3966	2264	1876	1583	1949	1376	1010	1021
1000	7051	4556	3399	3104	4276	2916	2377	2063

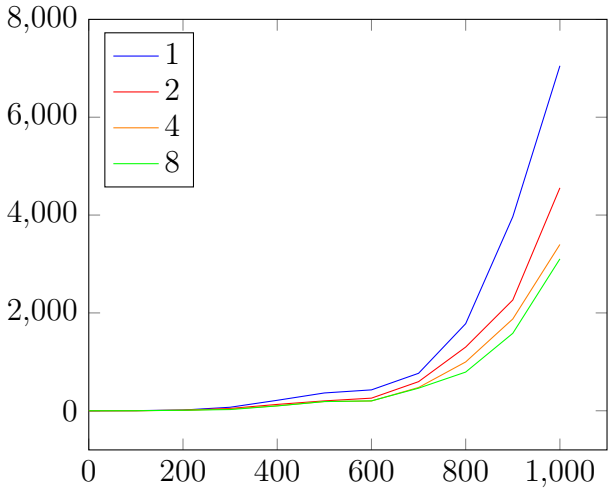


Рисунок 1. Временной график для стандартного алгоритма

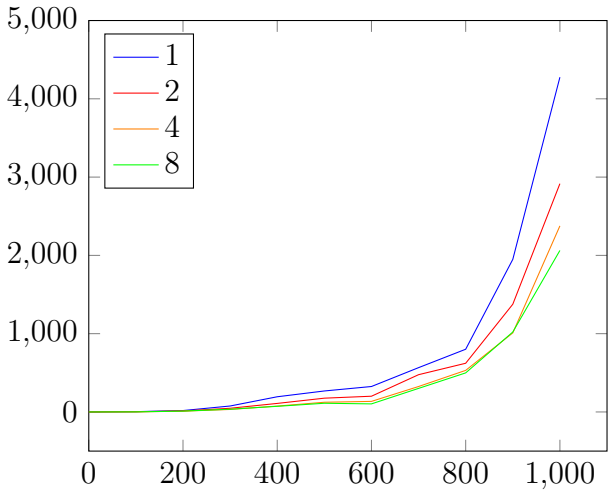


Рисунок 2. Временной график для алгоритма Винограда

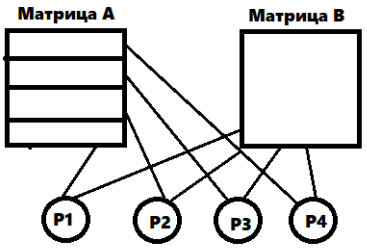
Выводы из эксперимента

В результате эксперимента было выявлено, что использование потоков повышает эффективность алгоритма умножения матриц.
В таблице ниже приведена информация о том, во сколько раз в среднем использование 2,4 или 8 потоков повышает скорость работы алгоритма по сравнению с вычислением в одном потоке.

	2 потока	4 потока	8 потоков
Стандартный алгоритм	1.71	2.28	2.49
Алгоритм Винограда	1.66	2.43	2.56

Способы распараллеливания задачи

В данной лабораторной работе был выбран способ, который заключается в разбиении задачи на N независимых подзадач умножения матрицы размерностью $\frac{n}{N}$ на матрицу размерностью n, где N - количество потоков, n - размерность исходных квадратных матриц. Подобная методика построения параллельных методов решения сложных задач является одной из основных в параллельном программировании и широко используется в практике.

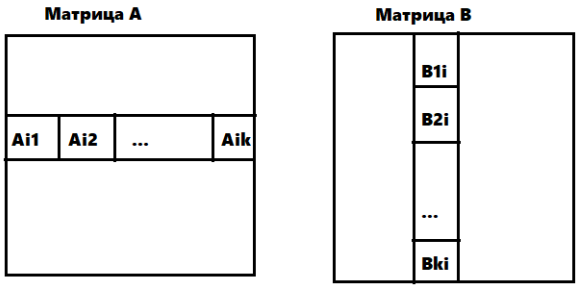


При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц: Пусть количество процессоров составляет $p = k^2$, а количество строк и столбцов матрицы является кратным величине k, т.е. $n = m * k$. Представим исходные матрицы A и B, и результирующую матрицу C в виде наборов прямоугольных блоков размера $m*m$. Тогда операцию матричного умножения

матриц A и B в блочном виде можно представить следующим образом:
$$\begin{bmatrix} A_{11} & ... & A_{1k} \\ ... & ... & ... \\ A_{k1} & ... & A_{kk} \end{bmatrix} * \begin{bmatrix} B_{11} & ... & b_{1k} \\ ... & ... & ... \\ B_{k1} & ... & B_{kk} \end{bmatrix} = \begin{bmatrix} C_{11} & ... & C_{1k} \\ ... & ... & ... \\ C_{k1} & ... & C_{kk} \end{bmatrix}$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением

$$C_{ij} = \sum_{u=1}^{k} A_{iu} B_{uj}$$



Параллельные методы матричного умножения приводят к равномерному распределению вычислительной нагрузки между процессорами.

Вывод

В ходе лабораторной работы были реализованы алгоритмы умножения матриц средствами 2,4 и 8 потоков. Было проведено сравнение алгоритмов многопоточного умножения матриц с однопоточным, проведены эксперименты с замерами времени на матрицах разных размерностей, сделаны выводы об эффективности многопоточного умножения матриц.