Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

Московский государственный технический университет имени Н.Э.Баумана
(МГТУ им. Н.Э.Баумана)

# ОТЧЕТ
## По лабораторной работе № 4
## По курсу «Анализ алгоритмов»
## на тему «Параллельное умножение матриц»

Выполнил Студент: Московец Н.С
Группа: ИУ7-51

Москва, 2017

## Оглавление

**Оглавление** **2**

**Постановка задачи** **2**

**Алгоритм Винограда** **2**

Описание 2

Реализация 2

**Распараллеливание алгоритма** **3**

Идея 3

Реализация 3

**Сравнение алгоритмов** **5**

**Заключение** **6**

## Постановка задачи

Изучить и реализовать параллельный алгоритм Винограда для умножения матриц, сравнить зависимость времени работы алгоритма от числа параллельных потоков исполнения и размера матриц и провести сравнение стандартного и параллельного алгоритма.

## Алгоритм Винограда

1. Описание

Рассмотрим два вектора V = (v1, v2, v3, v4) и W = (w1, w2, w3, w4). Их скалярное произведение равно:

V • W = v1w1 + v2w2 + v3w3 + v4w4.

Это равенство можно переписать в виде:

V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4.

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.

Выражение v1w1 + v2w2 + v3w3 + v4w4 имеет большую трудоемкость, чем (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3).

2. Реализация

```
1. Matrix multVinograd(const Matrix &a, const Matrix &b)
2. {
3.     assert(a.col == b.row);
4.
5.     Matrix res(a.row, b.col);
```

```
6.
7.     elemType rowFactor[a.row];
8.     elemType colFactor[b.col];
9.  // f1 =  2 + 6*n + 15*n*m / 2
10.        for(int i = 0; i < a.row; i++) { // 2 + a.row(2 + 4 + 15*a.col / 2)
11.           rowFactor[i] = 0; //2
12.           for(int j = 0; j < a.col / 2; j++) { // 2 + (a.col/2)(3 + 12) = 2 +
    15*a.col / 2
13.              rowFactor[i] = rowFactor[i] + a.arr[i][2 * j + 1] * a.arr[i][2 *
    j]; // 12
14.           }
15.        }
16.      // f2 =  2 + 6*k + 15*m*k / 2
17.        for(int i = 0; i < b.col; i++) {  //2 + 6*b.col + 15*a.col*b.col / 2
18.           colFactor[i] = 0;
19.           for(int j = 0; j < a.col / 2; j++) {
20.              colFactor[i] = colFactor[i] + b.arr[2 * j + 1][i] * b.arr[2 *
    j][i];
21.           }
22.        }
23.     // f3 = 2 + 2n + 11nk + 26nmk/2
24.        for(int i = 0; i < a.row; i++) { //2 + a.row(2 + 11*b.col +
    26*a.col*b.col/2)
25.           for(int j = 0; j < b.col; j++) { //2 + b.col*(11 + 26*a.col/2)
26.              res.arr[i][j] = -rowFactor[i] - colFactor[j]; //7
27.              for(int k = 0; k < a.col / 2; k++) { // 2 + (a.col/2)(3 + 23)
28.                 res.arr[i][j] = res.arr[i][j] + (a.arr[i][2*k+1] +
    b.arr[2*k][j]) *
29.                                                 (a.arr[i][2*k] + b.arr[2*k+1][j]);
    //23
30.              }
31.           }
32.        }
33.        //f4 = 2 - если m четное
34.        //f4 = 4 + 4n + 15nk - если m нечетное
35.        if(a.col % 2 == 1) { //4 + a.row(4 + 15*b.col)
36.           for(int i = 0; i < a.row; i++) //2 + a.row(4 + 15*b.col)
37.              for(int j = 0; j < b.col; j++) //2 + b.col(2 + 13)
38.                 res.arr[i][j] = res.arr[i][j] + a.arr[i][a.col-1] *
    b.arr[a.col-1][j]; //13
39.        }
40.
41.        return res;
42.     }
```

**Распараллеливание алгоритма**

1. Идея

Как было показано в лабораторной работе №2, трудоемкость алгоритма

винограда имеет сложность O(nmk) для умножение матриц n×m на m×k. Таким образом, чтобы значительно улучшить алгоритм, следует распараллелить ту часть алгоритма, которая содержит 3 вложенных цикла.

Можно заметить, что вычисление результата для каждой строки происходит независимо от результата выполнения умножения для других строк. Поэтому возможно распараллелить участок кода, соответствующий строкам 24-32. Каждый поток будет выполнять вычисление некоторых строк результирующий матрицы. Это сделано потому, что проход по строкам матрицы является более эффективным с точки зрения организации данных в памяти.

## 2. Реализация

Функция, передаваемая каждому потоку:

```
1. void funcThread(const Matrix &a, const Matrix &b, Matrix &res,
2.                       elemType* rowFactor,  elemType* colFactor, int num,
   int count)
3. {
4.    for(int i = num; i < a.row; i += count) {
5.       for(int j = 0; j < b.col; j++) {
6.          res.arr[i][j] = -rowFactor[i] - colFactor[j];
7.          for(int k = 0; k < a.col / 2; k++) {
8.             res.arr[i][j] = res.arr[i][j] + (a.arr[i][2*k+1] + b.arr[2*k][j])
   *
9.                   (a.arr[i][2*k] + b.arr[2*k+1][j]);
10.            }
11.         }
12.      }
13.   }
```

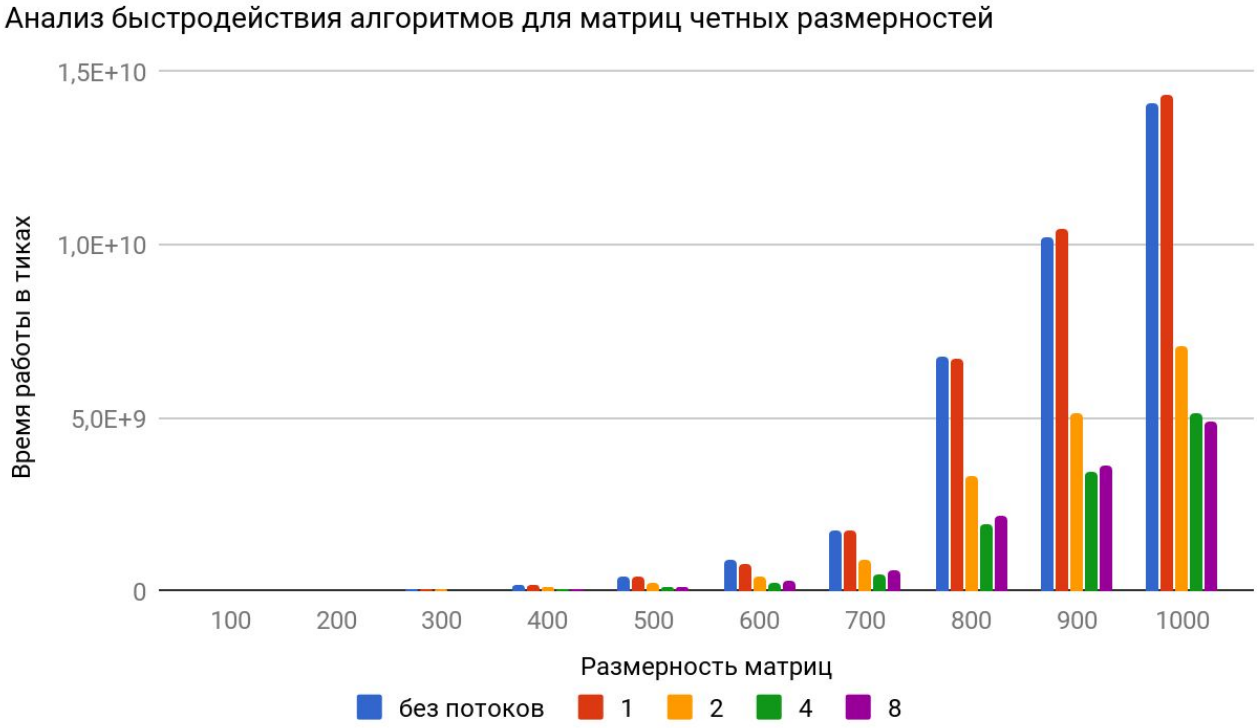
Алгоритм:

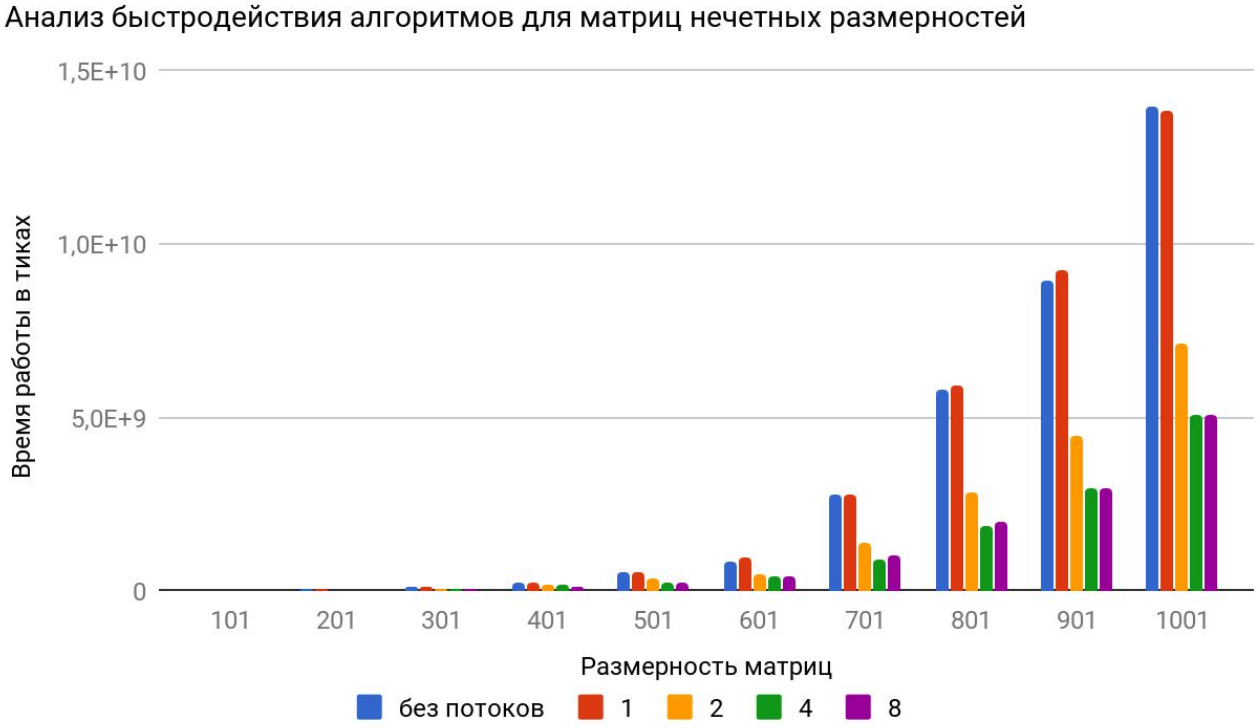
```
1. Matrix multThreadVinograd(const Matrix &a, const Matrix &b, int count)
2. {
3.    assert(a.col == b.row);
4.
5.    Matrix res(a.row, b.col);
6.
7.    elemType rowFactor[a.row];
8.    elemType colFactor[b.col];
9.
10.
11.      for(int i = 0; i < a.row; i++) {
12.         rowFactor[i] = 0;
13.         for(int j = 0; j < a.col / 2; j++) {
14.            rowFactor[i] = rowFactor[i] + a.arr[i][2 * j + 1] * a.arr[i][2 *
   j];
15.         }
16.      }
17.
```

```
18.       for(int i = 0; i < b.col; i++) {
19.          colFactor[i] = 0;
20.          for(int j = 0; j < a.col / 2; j++) {
21.             colFactor[i] = colFactor[i] + b.arr[2 * j + 1][i] * b.arr[2 * j][i];
22.          }
23.       }
24.
25.       std::thread threads[count];
26.
27.       for(int i = 0; i < count; i++) {
28.          threads[i] = std::thread(&funcThread, std::ref(a), std::ref(b),
    std::ref(res),
29.                                   &rowFactor[0], &colFactor[0], i,
    count);
30.       }
31.       for(int i = 0; i < count; i++) {
32.          if(threads[i].joinable()) {
33.             threads[i].join();
34.          }
35.       }
36.
37.
38.       if(a.col % 2 == 1) {
39.          for(int i = 0; i < a.row; i++)
40.             for(int j = 0; j < b.col; j++)
41.                res.arr[i][j] = res.arr[i][j] + a.arr[i][a.col-1] *
    b.arr[a.col-1][j];
42.       }
43.
44.       return res;
45.    }
```

## Сравнение алгоритмов

Было произведено сравнение алгоритма Винограда и распарралливаемого алгоритма. Выполнения программы производилось на компьютере, который может проводить вычисления в 4 реально параллельных потоках. Тогда для анализа быстродействия алгоритмов алгоритм был протестирован для 1, 2, 4 и 8 потоков. Для сравнения алгоритмов было посчитано время работы для матриц размерностью 100×100, 200×200, …, 1000×1000 и 101×101, 202×202, …, 1001×1001.




Выводы:

- Параллельный алгоритм, выполняемый на одном потоке выполняется медленнее, чем обычный алгоритм Винограда. Это можно объяснить тем, что на подготовку потока требуется дополнительное время и ресурсы.
- Увеличение количества потоков (до 2, 4) сокращает время выполнения алгоритма.
- Дальнейшее увеличение количества потоков практически не влияет на

повышение быстродействия алгоритма, так как реально вычисления на компьютере, выбранном для проведения эксперимента, выполняются только в 4 потока.

## Заключение

Во время выполнения работы был изучен и реализован параллельный алгоритм Винограда для умножения матриц. Было произведено сравнение зависимости времени работы алгоритма от числа параллельных потоков исполнения и размера матриц и проведено сравнение стандартного и параллельного алгоритма.