Strassen's Algorithm

Elena Spirova

UP FAMNIT

89201050@student.upr.si

***Abstract*—This report introduces the Strassen's algorithm for matrix multiplication and its sequential, parallel and distributed implementation in Java. The conclusion of the comparison of different modes of implementation of the algorithm states that the parallel execution provides the best results. Strassen's algorithm gives results faster than the standard matrix multiplication algorithm for large matrices, with a better asymptotic complexity, although the naive algorithm is often better for smaller matrices.**

I. Introduction

Let A and B that are two real n x n matrices. Asign C to denote the product of the matrices A and B, such that C = AB. By definition, if we denote by a(i,j) the element of matrix A that is in the i-th row and j-th column, and, similarly for B and C, we have that the element c(i,j) of C is equal to

$$c(i,j) = \sum_{n=1}^{k} a(i,k)b(k,j) \tag{1}$$

The pseudo code for the above equation would be:

Algorithm 1: Naive algorithm

```
/* Since we have three nested for loops,
   the time complexity of this algorithm
   is
                    O(n^3)
   , assuming multiplication and memory
   access takes constant time. A slight
   optimization of naive matrix
   multiplication is Strassen's algorithm,
   which cleverly uses Divide and Conquer
   to get an improvement.                */
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        c(i,j)=0;
        for k ← 0 to n − 1 do
            c(i,j)=c(i,j)+a(i,k)b(k,j);
        end for
    end for
end for
return C
```

II. Design

Without loss of generality let A, B and C be the same matrices defined above, and moreover suppose that all of these matrices have sizes that are powers of two. We are able to do this as we can always add extra all-zero lines and rows (basically padding) in the matrices A and B, such that the value of n is an integer power of 2. In order to proceed with the Strassen's algorithm, we use the Divide-And-Conquer method to rewrite the matrices A, B and C as block matrices:

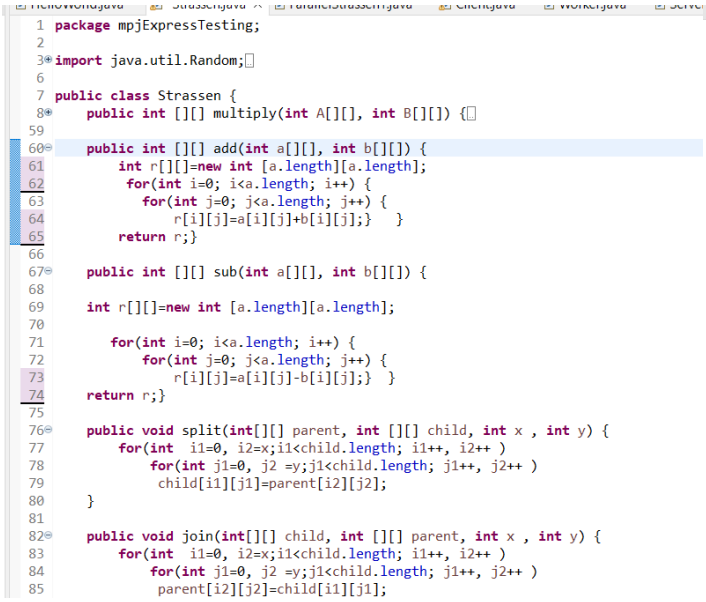
$$\begin{bmatrix} a11 & a12 \\ a21 & a22 \end{bmatrix}, \begin{bmatrix} b11 & b12 \\ b21 & b22 \end{bmatrix}, \begin{bmatrix} c11 & c12 \\ c21 & c22 \end{bmatrix}$$

The dimension of each of the sub-matrices in the above block decomposition is (n/2) x (n/2); this is why assuming that n is a power of 2 is convenient. It is now easy to see that the following equations hold.

$$\begin{aligned} p &= (a11 + a22) * (b11 + b22) \\ q &= ((a21 + a22) * B11); \\ r &= (a11 * (b12 + b22)); \\ s &= (a22 * (b21 - b11)); \\ t &= ((a11 + a12) * b22); \\ u &= ((a21 - a11) * (b11 + b12)); \\ v &= ((a12 - a22) * (b21 + b22)); \end{aligned}$$

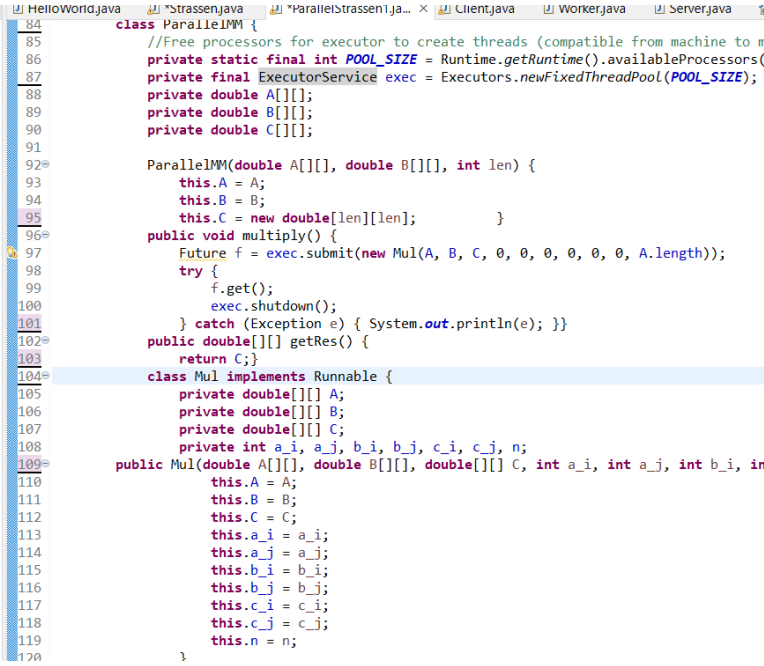
A crucial observation is that each of the matrices can be computed with exactly one recursive call, as the remaining operations are additions. Now the following expression holds:

C11=(((p+s)-t)+v)
C12=(r+t)
C21=(q+s)
C22=(((p+r)-q)+u)
(2)



```
1  package mpjExpressTesting;
2
3  import java.util.Random;
6
7  public class Strassen {
8      public int [][] multiply(int A[][], int B[][]) {
59
60     public int [][] add(int a[][], int b[][]) {
61         int r[][]=new int [a.length][a.length];
62          for(int i=0; i<a.length; i++) {
63            for(int j=0; j<a.length; j++) {
64                r[i][j]=a[i][j]+b[i][j];}   }
65         return r;}
66
67     public int [][] sub(int a[][], int b[][]) {
68
69     int r[][]=new int [a.length][a.length];
70
71        for(int i=0; i<a.length; i++) {
72            for(int j=0; j<a.length; j++) {
73                r[i][j]=a[i][j]-b[i][j];}  }
74     return r;}
75
76     public void split(int[][] parent, int [][] child, int x , int y) {
77         for(int  i1=0, i2=x;i1<child.length; i1++, i2++ )
78             for(int j1=0, j2 =y;j1<child.length; j1++, j2++ )
79              child[i1][j1]=parent[i2][j2];
80     }
81
82     public void join(int[][] child, int [][] parent, int x , int y) {
83         for(int  i1=0, i2=x;i1<child.length; i1++, i2++ )
84             for(int j1=0, j2 =y;j1<child.length; j1++, j2++ )
85              parent[i2][j2]=child[i1][j1];
```

Fig. 1. Sequential mode



```
84      class ParallelMM {
85          //Free processors for executor to create threads (compatible from machine to m
86          private static final int POOL_SIZE = Runtime.getRuntime().availableProcessors(
87          private final ExecutorService exec = Executors.newFixedThreadPool(POOL_SIZE);
88          private double A[][];
89          private double B[][];
90          private double C[][];
91
92          ParallelMM(double A[][], double B[][], int len) {
93              this.A = A;
94              this.B = B;
95              this.C = new double[len][len];          }
96          public void multiply() {
97              Future f = exec.submit(new Mul(A, B, C, 0, 0, 0, 0, 0, 0, A.length));
98              try {
99                  f.get();
100                 exec.shutdown();
101             } catch (Exception e) { System.out.println(e); }}
102         public double[][] getRes() {
103             return C;}
104         class Mul implements Runnable {
105             private double[][] A;
106             private double[][] B;
107             private double[][] C;
108             private int a_i, a_j, b_i, b_j, c_i, c_j, n;
109         public Mul(double A[][], double B[][], double[][] C, int a_i, int a_j, int b_i, in
110                 this.A = A;
111                 this.B = B;
112                 this.C = C;
113                 this.a_i = a_i;
114                 this.a_j = a_j;
115                 this.b_i = b_i;
116                 this.b_j = b_j;
117                 this.c_i = c_i;
118                 this.c_j = c_j;
119                 this.n = n;
120             }
```

Fig. 2. Parallel mode

III. Problem Description

The program can be run in three modes: sequential, parallel and distributed mode. In all of them the user specifies the size of the matrices, the parameter that influences the run time, and the program fills them with random values.

A. Sequential implementation

Figure 1 shows that it is quite intuitive to see what we are doing in the sequential part of our code. In the multiply function we have the part of the code where we divide the matrices into 4 halves that calculate the values p, q, r, s, t, u, v. This method uses all other methods which are defined in the above code, in order to calculate the matrices p, q, r, s, t, u, v. Those methods are splitting the matrices, addition and subtraction of matrices and joining the results in the resulting matrix C

B. Parallel implementation

The parallel implementation of the program uses the number of available processors and we create a fixed thread pool, then each thread takes care of a different task - to each thread is assigned a sub matrix for calculation. So the parallel algorithm is subdividing the matrices into small size matrices and distributing them among the number of available processors to achieve faster running time. The rest of the code follows the serial algorithm to get the job done, which is mostly methods for computation of sum and difference among two matrices.

C. Distributive mode

The distributive mode introduces multiple servers, who can be run locally as well as on different machines that compute the product of two matrices. To achieve that the parallel mode of Strassen's algorithm is put into use. There's no limit on their numbers but their addresses are hard-coded. Thus, a server cannot be added or removed dynamically. The size of the matrices is read as as input in the Client class, it then uses the first iteration of Strassen's algorithm, but each multiplication is assigned to a worker server. The later is chosen randomly, we don't try estimate the load of each server so we assume that they have the same computing capabilities. The communication between the client and the server uses Java's Remote Method Invocation (basically equivalent to a Remote Procedure Call). For testing purposes we run the workers locally but on different ports. Before running the test, they should be started. By default, there are 3 of them listening on port 3000, 3001 and 3002

IV. Testing

Inputs of multiple sizes were tested, namely 500, 1000, 1500, 2000, 2500. The experiments were performed on a machine that runs on Intel® Core™ i3-1005G1 CPU @ 1.20GHz 1.19 GHz CPU, with 8,00 GB of RAM In the table bellow we can see the input sizes and running time of each program mode. We observe that the sequential implementation gives the best results for the smaller matrices, but as the size of the matrices increases the parallel implementation is better, hence we see the its true power. As the distributed implementation of the algorithm is based on the sequential mode, the results are moreover the same for both modes.

V. Conclusion

Strassen's algorithm has the problem of instability. We observed that the parallel implementation is the most

size	sequential	parallel	distributive
500	6,7248133 seconds	6,5955277 seconds	12.021652 seconds
1000	31,6066432 seconds	24,0858473 seconds	67.11915 seconds
1500	127,5462153 seconds	153,8485561 seconds	558,03394 seconds
2000	148,2766699 seconds	182,9191135 seconds	exceeds
2500	exceeds	exceeds	exceeds

efficient. Another observation is that the running time of the program depends on the user input - the size of the matrices. Our conclusion is that the parallel mode is the way to improve the efficiency and speed of the Strassen's algoritham for multiplication of matrices.