Matrix Multiplication Algorithms

Khaled Thabet

Faculty of Computer Sciences & IT, King Abdulaziz University, Jeddah, KSA

Sumaia AL-Ghuribi

Faculty of Computer Sciences & IT, King Abdulaziz University, Jeddah, KSA

Summary
Algorithm can be written in various ways, executes sequential or parallel, and gives the same results. One of the main factors of determining the efficient of an algorithm is the execution time factor, how much time the algorithm takes to accomplish its work. Because matrix multiplication widely used in a variety of applications and is often one of the core components of many scientific computations, it will be taken as a problem in this work and different algorithms are given to solve this problem. Then the execution time of all the methods will be calculated to find the best method for matrix multiplication. After testing Twenty three methods, we find that parallel Strassen algorithm is the best method for finding matrix multiplication.

Key words:
Algorithm, parallel execution, matrix multiplication, Strassen algorithm.

1. Introduction

Algorithm is a set of instructions for solving a problem and the efficiency of implementation of the algorithm depends upon speed, size, and resources consumption. Many instructions can give the same result for a particular problem .On the other hand, the execution of these instructions are different. Some of them take less time and space than the others. Some of them become more efficient when it executes parallel. As a result, we try to select the suitable instruction that gives the best execution, less time and space.

Small Algorithms need one processor to execute efficiently and give the required result in a record time. But one processor is not enough for executing large and complex problems, so we need more than one processor to enhance the execution time of algorithm and this called parallel computing.

In the simplest sense, parallel execution is the simultaneous use of multiple compute resources to solve a computational problem. Parallel execution has the following characteristics, use multiple CPUs, the problem to be solved is broken to sub problems that can be solved concurrently, each sub problem is broken down to a series of instructions and instructions from each sub problem executes simultaneously on different CPUs .

The complexity of matrix multiplication has attracted a lot of attention in the last forty years. In this paper we will consider matrix multiplication as the problem, give various methods to solve this problem and find the best one that takes the least time.

Matrix multiplication is the kernel of many scientific applications [8, 9]. It is a binary operation that takes a pair of matrices, and produces another matrix. If A is an n-by-m matrix and B is an m-by-p matrix, the result AB of their multiplication is an n-by-p matrix defined only if the number of columns m of the left matrix A is the equal to the number of rows of the right matrix B. The result of matrix multiplication is a matrix whose elements are found by multiplying the elements within a row from the first matrix by the associated elements within a column from the second matrix and summing the products. The procedure for finding an element of the resultant matrix is to multiply the first element of a given row from the first matrix times the first element of a given column from the second matrix, then add to that the product of the second element of the same row from the first matrix and the second element of the same column from the second matrix, then add the product of the third elements and so on, until the last element of that row from the first matrix is multiplied by the last element of that column from the second matrix and added to the sum of the other products. Ex:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

As we mentioned before there are many methods to calculate the multiplication of matrixes. All of them give the same result but each one consumes different space in memory and takes different processor time. The methods that we will test are:

1. Row by Column method
2. Row by Row method
3. Column by Column method
4. Strassen method

We will test each of them with all the possible types of each one, “sequential, blocked and parallel”.

The rest of this paper is organized as follows: in section II, we give a brief review of some related works about matrix multiplication, in section III we presents Row by Column method with all its type, section IV displays Row by Row

Manuscript received February 5, 2012
Manuscript revised February 20, 2012

method with all its type, section V Column by Column method with all its type, Section VI compares between the previous three methods, Section VII shows Strassen's method and finally we conclude this work and outline future research in Section VIII.

2. Related Works

A brief review of some related works about matrix multiplication has been done.

In [2], the authors in this paper interested in reducing the cost of multiplication for matrices of small size, say up to 30.Following the previous work of Probert & Fischer, Smith, and Mezzarobba, in a similar vein, they base their approach on the previous algorithms for small matrices, due to Strassen, Winograd, Pan, Laderman, and others and show how to exploit these standard algorithms in an improved way. They illustrated the use of their results by generating multiplication codes over various rings, such as integers, polynomials, differential operators and linear recurrence operators.

In [3], the authors analyzed the performance and scalability of a number of parallel formulations of the matrix multiplication algorithm and predict the conditions under which each formulation is better than the other. They showed that the GK algorithm that they present in their paper outperforms all the well known algorithms for a significant range of number of processors and matrix sizes.

In [4], the authors reported on an implementation of Strassen algorithm that uses several unconventional techniques to make the algorithm memory-friendly. First, the algorithm internally uses a nonstandard array layout known as Morton order that is based on a quad-tree decomposition of the matrix. Second, they dynamically select the recursion truncation point to minimize padding without affecting the performance of the algorithm, which they can do by virtue of the cache behavior of the Morton ordering. Each technique is critical for performance, and their combination as done in their code multiplies their effectiveness. Performance comparisons of their implementation with that of competing implementations showed that their implementation often outperforms the alternative techniques (up to 25%). They also noted that the time required converting matrices to/from Morton order is a noticeable amount of execution time (5% to 15%).

In [5], the authors reported on the development of an efficient and portable implementation of Strassen's matrix multiplication algorithm. Their implementation is designed to be used in place of DGEMM, the level 3 BLAS matrix multiplication routine. Efficient performance will be obtained for all matrix sizes and shapes and the additional memory needed for temporary variables has been minimized. Their performance data reconfirms that Strassen's algorithm is practical for realistic size matrices.

In [6], the author proposed a new distribution scheme for a parallel Strassen's matrix multiplication algorithm on heterogeneous clusters. Their scheme achieves both load balancing and reduction of the total operation count. As a result, they achieved a speedup of nearly 21.7% compared to the conventional parallel Strassen's algorithm in a heterogeneous clustering environment.

3. Row by Column Method

Suppose that B is a i x K matrix, and C is a K x j matrix. Then, the matrix product BC results in a matrix A, which has i rows and j columns; and each element in A can be computed according to the following formula: $A_{ij} = \Sigma_k B_{ik}C_{kj}$

Where : A_{ij} = the element in row i and column j from matrix A;B_{ik} = the element in row i and column k from matrix B;C_{kj}= the element in row k and column j from matrix C;Σ_k = summation sign, which indicates that the $B_{ik}C_{kj}$ terms should be summed over k.

Row by column method can be represented in many various ways. Following are six ways for representing it:

III.1 Sequential

```
for ( i = 0; i < size; i++)
   for ( j = 0; j < size; j++)
      for ( k = 0; k < size; k++)
         {
           A[i, j] += B[i, k] * C[k, j];
   }
```

III.2 Enhanced Sequential

```
for (i = 0; i < size; i++)
    f or (j = 0; j < size; j++)
       {
          sum = 0;
          for (k = 0; k < size; k++)
              sum += B[i, k] *C[k,j];
           A[i, j] = sum;
          }
       }
```

III.3 Sequential block

```
for ( i1 = 0; i1 < size; i1 += bsize)
  for ( j1 = 0; j1 < size; j1 += bsize)
    for ( k1 = 0; k1 < size; k1 += bsize)
      for ( i = i1; i < i1 + bsize && i < size; i++)
        for ( j = j1; j < j1 + bsize && j < size; j++)
          for ( k = k1; k < k1 + bsize && k < size; k++)
            A[i, j] += B[i, k] * C[k, j];
```

III.4 Parallel Block

```
System.Threading. Parallel.Do(delegate()
 {  for ( i1 = 0; i1 < size; i1 += bsize)
     for ( j1 = 0; j1 < size; j1 += bsize)
      for ( k1 = 0; k1 < size; k1 += bsize)
       for ( i = i1; i < i1 + bsize && i < size; i++)
        for ( j = j1; j < j1 + bsize && j < size; j++)
         for ( k = k1; k < k1 + bsize && k < size; k++)
          A[i, j] += B[i, k] * C[k, j];
           });
```

III.5 Enhanced Parallel Block

```
System.Threading.Parallel.Do(() =>
  { for ( i1 = 0; i1 < size / 2; i1 += bsize)
     for ( j1 = 0; j1 < size ; j1 += bsize)
      for ( k1 = 0; k1 < size ; k1 += bsize)
       for ( i = i1; i < i1 + bsize && i < size; i++)
        for ( j = j1; j < j1 + bsize && j < size; j++)
         for ( k = k1; k < k1 + bsize && k < size;
 k++)
          A[i, j] += B[i, k] * C[k, j];
       }, () =>
  { for ( i1 = size / 2; i1 < size; i1 += bsize)
     for ( j1 = 0; j1 < size ; j1 += bsize)
      for ( k1 = 0; k1 < size ; k1 += bsize)
       for ( i = i1; i < i1 + bsize && i < size; i++)
        for ( j = j1; j < j1 + bsize && j < size; j++)
         for ( k = k1; k < k1 + bsize && k < size;
 k++)
          A[i, j] += B[i, k] * C[k, j];
       });
```

III.6 Parallel

```
System.Threading.Parallel.For(0, size, (i) =>
          {   for ( j = 0; j < size; j++)
                for ( k = 0; k < size; k++)
                  A[i, j] += B[i, k] * C[k, j];
          }
          );
```

Where
A is the result matrix of multiplying B and C size is the size of the matrix bsize is the size of the block
We tested the previous methods in different cases starting with matrix size 50 elements until 1000 elements and the result appeared as the following diagram

Figure (1). Comparing the average execution time of all algorithms of Row by Column method.

4. Row by Row

Row by Row method can also be represented in many various ways. Following are six ways for representing it:

IV.1 Sequential

```
for ( i = 0; i < size; i++)
   for ( j = 0; j < size; j++)
      for ( k = 0; k < size; k++)
         {
           A[i, k] += B[i, j] * C[j, k];
   }
```

IV.2 Enhanced Sequential

```
for (i = 0; i < size; i++)
     f or (j = 0; j < size; j++)
       {
          a1 = B[i,j];
         for (k = 0; k < size; k++)
           A[i,k] += a1 *C[j,k];
        }
```

IV.3 Sequential block

```
for ( i1 = 0; i1 < size; i1 += bsize)
 for ( j1 = 0; j1 < size; j1 += bsize)
  for ( k1 = 0; k1 < size; k1 += bsize)
   for ( i = i1; i < i1 + bsize && i < size; i++)
    for ( j = j1; j < j1 + bsize && j < size; j++)
     for ( k = k1; k < k1 + bsize && k < size; k++)
      A[i, k] += B[i, j] * C[j,k];
```

IV.4 Parallel Block

```
System.Threading.Parallel.Do(delegate()
   {
     for ( i1 = 0; i1 < size; i1 += bsize)
      for ( j1 = 0; j1 < size; j1 += bsize)
       for ( k1 = 0; k1 < size; k1 += bsize)
        for ( i = i1; i < i1 + bsize && i < size; i++)
         for ( j = j1; j < j1 + bsize && j < size; j++)
          for ( k = k1; k < k1 + bsize && k < size; k++)
           A[i,k] += B[i, j] * C[j,k];
        });
```

IV. 5 Enhanced Parallel Block

```
System.Threading.Parallel.Do(() =>
   {   for ( i1 = 0; i1 < size / 2; i1 += bsize)
         for ( j1 = 0; j1 < size ; j1 += bsize)
           for ( k1 = 0; k1 < size ; k1 += bsize)
             for ( i = i1; i < i1 + bsize && i < size; i++)
               for ( j = j1; j < j1 + bsize && j < size; j++)
                 for ( k = k1; k < k1 + bsize && k < size; k++)
                       A[i,k] += B[i, j] * C[j, k];
       }, () =>
       {  for ( i1 = size / 2; i1 < size; i1 += bsize)
          for ( j1 = 0; j1 < size ; j1 += bsize)
            for ( k1 = 0; k1 < size ; k1 += bsize)
              for ( i = i1; i < i1 + bsize && i < size; i++)
                for ( j = j1; j < j1 + bsize && j < size; j++)
                 for ( k = k1; k < k1 + bsize && k < size; k++)
                       A[i, k] += B[i, j] * C[j,k];
        });
```

IV.6 Parallel

```
System.Threading.Parallel.For(0, size, (i) =>
            {
               for ( j = 0; j < size; j++)
                  for ( k = 0; k < size; k++)
                     A[i,k] += B[i, j] * C[j,k];
            }
            );
```

Where
A is the result matrix of multiplying B and C size is the size of the matrix bsize is the size of the block

We also tested the previous methods in different cases starting with matrix size 50 elements until 1000 elements and the result appeared as the following diagram

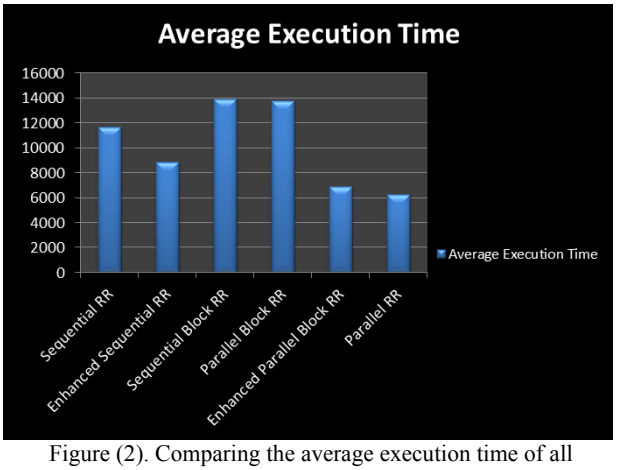

Figure (2). Comparing the average execution time of all algorithms of Row by Row method.

5. Column by Column

Column by Column method can also be represented in many various ways. Following are six ways for representing it:

V.1 Sequential

```
for ( i = 0; i < size; i++)
    for ( j = 0; j < size; j++)
        for ( k = 0; k < size; k++)
           {
             A[k,i] += B[k,j] * C[j,i];
                                }
```

V.2 Enhanced Sequential

```
for (i = 0; i < size; i++)
    f or (j = 0; j < size; j++)
       {
          c1 = C[j,i];
          for (k = 0; k < size; k++)
           A[k,i] += B[k,j] *c1;
        }
```

V.3 Sequential block

```
for ( i1 = 0; i1 < size; i1 += bsize)
 for ( j1 = 0; j1 < size; j1 += bsize)
  for ( k1 = 0; k1 < size; k1 += bsize)
   for ( i = i1; i < i1 + bsize && i < size; i++)
    for ( j = j1; j < j1 + bsize && j < size; j++)
     for ( k = k1; k < k1 + bsize && k < size; k++)
      A[k,i] += B[k,j] * C[j,i];
```

V.4 Parallel Block

```
System.Threading.Parallel.Do(delegate()
   {
     for ( i1 = 0; i1 < size; i1 += bsize)
       for ( j1 = 0; j1 < size; j1 += bsize)
        for ( k1 = 0; k1 < size; k1 += bsize)
          for ( i = i1; i < i1 + bsize && i < size; i++)
            for ( j = j1; j < j1 + bsize && j < size; j++)
              for ( k = k1; k < k1 + bsize && k < size; k++)
                A[k,i] += B[k,j] * C[j,i];
        });
```

V .6 Parallel

```
System.Threading.Parallel.For(0, size, (i) =>
            {
               for ( j = 0; j < size; j++)
                  for ( k = 0; k < size; k++)
                     A[k,i] += B[k, j] * C[j,i];
            }
            );
```

Where
A is the result matrix of multiplying B and C size is the size of the matrix bsize is the size of the block

We again tested the previous methods in different cases starting with matrix size 50 elements until 1000 elements and the result appeared as the following diagram

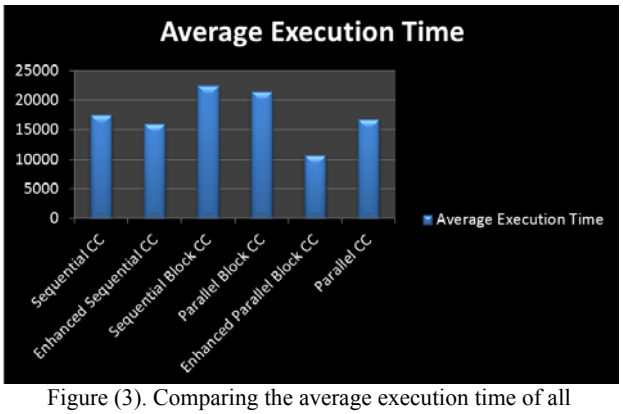

Figure (3). Comparing the average execution time of all algorithms of Column by Column method.

6. Comparing among the Previous Three Methods

The previous 3 algorithms with their different types differ in instructions but give the same result .By making one diagram for all of them, it appears as follow:

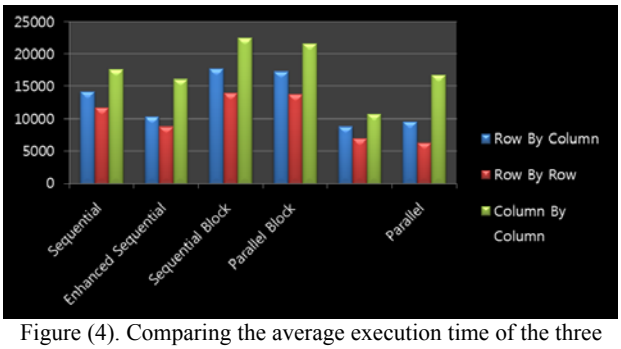

Figure (4). Comparing the average execution time of the three methods.

It is clearly appeared that Row by Row method gives the best average execution time.

7. Strassen's Method

In 1969, Strassen [10] introduced an algorithm to multiply M x M matrices which has a lower complexity than the classical O(M3). .It is based on a scheme for the product of two 2 x 2 matrices which involves 7 multiplications and 18 additions instead of the usual 8 multiplications and 4 additions. Strassen algorithm is better than standard matrix multiplication algorithm because additions and subtractions of matrices can be computed in linear time without communications. Strassen's algorithm is O (M2.81).On the other hand, Strassen's method needs more memory than the traditional methods. The following shows how the matrix multiplication using Strassen's method works:

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$\begin{aligned} P_1 &= (A_{11}+A_{22})(B_{11}+B_{22}) \\ P_2 &= (A_{21}+A_{22}) * B_{11} \\ P_3 &= A_{11} * (B_{12}-B_{22}) \\ P_4 &= A_{22} * (B_{21}-B_{11}) \\ P_5 &= (A_{11}+A_{12}) * B_{22} \\ P_6 &= (A_{21}-A_{11}) * (B_{11}+B_{12}) \\ P_7 &= (A_{12}-A_{22}) * (B_{21}+B_{22}) \end{aligned}$$

$$\begin{aligned} C_{11} &= P_1 + P_4 - P_5 + P_7 \\ C_{12} &= P_3 + P_5 \\ C_{21} &= P_2 + P_4 \\ C_{22} &= P_1 + P_3 - P_2 + P_6 \end{aligned}$$

Strassen's Algorithm can be written as follows:

```
void matmul(int *A, int *B, int *R, int n)
{
        if (n == 1)
                (*R) += (*A) * (*B);
        else {
                matmul(A, B, R, n/4);
                matmul(A, B+(n/4), R+(n/4), n/4);
                matmul(A+2*(n/4), B, R+2*(n/4), n/4);
                matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
                matmul(A+(n/4), B+2*(n/4), R, n/4);
                matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);
                matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
                matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
            }
}
```

We made four cases in implementing the Strassen's algorithm (Regular, Enhanced, Block and Parallel) and tested each case n times with a change in the size of the matrix each time.

Then we compared the result with the best traditional methods Row by Row method to investigate the best method for the matrixes multiplication.

The next diagram concludes what we said:

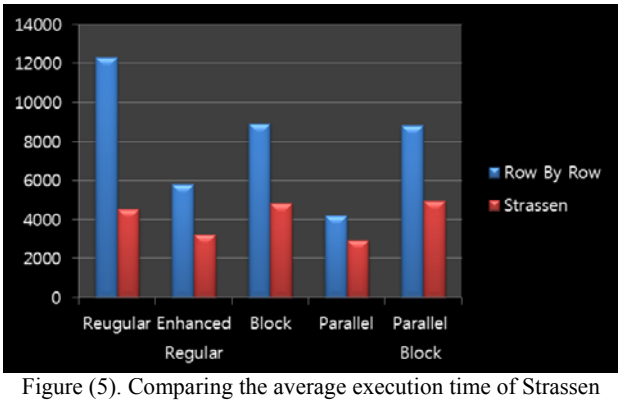

Figure (5). Comparing the average execution time of Strassen and Row by Row methods.

It is completely clear that Strassen takes less time than Row by Row and the parallel Strassen is the best case of Strassen. Parallel Strassen is 43% faster than parallel Row by Row.

8. Conclusion and Future works

When you want to solve any problem, try to choose the best method that consumes less space and time to execute efficiently. For matrix multiplication , we tested some methods Row by Row , Row By Column , Column By Column and Strassen. After our experiments we found that Strassen's method is the best method for implementing the matrix multiplication and Strassen parallel method is the best of all. It is 43% faster than the Row by Row method which is the better than Row by Column and Column By Column.

For the future work, we will test many other matrix multiplication algorithms and for every algorithm we will test the space complexity.

References

[1] http://en.wikipedia.org/wiki/Matrix_multiplication.

[2] Drevet , C , Islam , M and Schost , r.(2011).” Optimization techniques for small matrix multiplication”. ScienceDirect . Theoretical Computer Science 412 (2011) 2219–2236.

[3] Gupta ,A and ICumar ,V.(1993).”Scalability of Parallel Algorithms for Matrix Multiplication”. 1 993 International Conference on Parallel Processing.

[4] Thottethodi , M , Chatterjee , S and Lebeck , A.(1998).” Tuning Strassen's Matrix Multiplication for Memory Efficiency”. ACM/IEEE SC98 Conference (SC’98).

[5] Lederman , S , Jacobson , E ,Johnson , J , Tsao ,A and Turnbull , T .(1996).”Implementation of Strassen’s algorithm for Matrix Multiplication”. ACM/IEEE Conference on Supercomputing (SC’96).

[6] Ohtaki , Y .(2004).” Parallel Implementation of Strassen’s Matrix Multiplication Algorithm for Heterogeneous Clusters”. IEEE. 18th International Parallel and Distributed Processing Symposium (IPDPS’04).

[7] Desprez , F and Suter , F .(2001).” Mixed Parallel Implementations of the Top Level Step of Strassen and Winograd Matrix Multiplication Algorithms”. IEEE.

[8] Kågström , B , Ling, P and Loan, C.(1995). “GEMM-Based Level 3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark”. Technical Report UMINF-95.18, Umeå University, Oct 1995.

[9] Rönsch , W and Strauß , H . (1989).” The Level 3BLAS Forms of Parallel Factorization Methods”. In D. Evans, G. Joubert, and F. Peters, editors, Parallel Computing 89, pages 85– 92. Elsevier Science Publisher B.V., 1989.

[10] Strassen, V. (1969). “Gaussian Elimination Is Not Optimal”. Numerische Mathematik, 14(3):354–356, 1969.