Parallel Algorithms for Matrix Multiplication

Sandip. K.Bhagat\textsuperscript{1} Rahul .P. Limbole\textsuperscript{1} Abhijit .L. Sangale\textsuperscript{1} Sudhir .B. Sawant\textsuperscript{1} P.M. Chawan\textsuperscript{2}

\textsuperscript{1}Student Dept of Computer Science VJTI,Mumbai  
\textsuperscript{2}Prof Dept of Computer Science VJTI,Mumbai

Abstract

Many type of problem can be converted into matrix form. If we do matrix multiplication in the efficient way many problems can be solved easily and efficiently. This paper describes the various algorithms of Matrix Multiplication that can be implemented on parallel processing environment. Which includes Strassen’s Matrix Multiplication which can be done in $O(n^{2.81})$ using sequential computing. SUMMA algorithm works in $O(n^2)$ in the parallel environment. And Matrix Multiplication that can be done on SIMD processors in $O(\log n)$ times.

Key Words: Strassen's Matrix Multiplication, SUMMA Algorithm.

1. Introduction:

The parallel processing is the processing of program instructions by dividing them among multiple processors with the objective of running a program in less time. In serial processing, next instruction will not be executed until the pervious instruction completed. Parallel processing demands concurrent execution of many programs in computer. The highest level of parallel processing is conducted among multiple programs through multiprogramming, time sharing and multiprocessing. This level requires development of the parallel process able algorithms. The implementation of parallel algorithms depends on the efficient allocation of limited hardware-software resources to multiple programs being used to solve large manipulation problems.

2. Sequential Method:

Suppose we want to multiply two n-by-n matrices A and B, resulting in matrix C. The standard sequential algorithm (without cache blocking) is usually written as three nested loops on i (row number in C and A), then j (column number in C and B), and then k (inner index, column of A and row of B). However, we can nest the loops in an order and still get the same answer. Using the order kij instead of ijk, we get the following code:

Algorithm :

\begin{verbatim}
C = zeros(n,n);
for k = 1:n
  for i = 1:n
    for j = 1:n
      C(i,j) = C(i,j) + A(i,k) * B(k,j);
    end
  end
end
\end{verbatim}

In serial processing, the $n^3$ cumulative multiplications are carried out by a serially coded program with three levels of DO loops corresponding to three indices to be used. The time complexity of this sequential program is proportional to $n^3$.

3. Strassen’s Matrix Multiplication Algorithm for Sequential Processing :

Strassen has developed sequence of equations that are applied on matrix to reduce the number of multiplications. Because multiplication requires more time to compute the result. He reduces the number of multiplications and increases number of additions and subtractions to get the better results. Strassen's original algorithm whose asymptotic complexity is $O(n2.81)$ are considered the most practical. Strassen's algorithm compute the product C of two matrices A and B by first decomposing each matrix into 4 roughly equal sized blocks. Strassen's algorithm computes C by performing 7 matrix multiplications and 18 add/subtracts using the following equations:

\begin{itemize}
  \item $C_{12} = A_{21} - A_{11}$
\end{itemize}
2  \( C_{21} = B_{11} + B_{12} \)
3  \( C_{22} = C_{12} \times C_{21} \)
4  \( C_{12} = A_{12} - A_{22} \)
5  \( C_{21} = B_{21} + B_{22} \)
6  \( C_{11} = C_{12} \times C_{21} \)
7  \( C_{12} = A_{11} + A_{22} \)
8  \( C_{21} = B_{11} + B_{22} \)
9  \( T_{1} = C_{12} \times C_{21} \)
10 \( C_{11} = T_{1} + C_{11} \)
11 \( C_{22} = T_{1} + C_{22} \)
12 \( T_{2} = A_{21} + A_{22} \)
13 \( C_{21} = T_{2} \times B_{11} \)
14 \( C_{22} = C_{22} - C_{21} \)
15 \( T_{1} = B_{21} - B_{11} \)
16 \( T_{2} = A_{22} \times T_{1} \)
17 \( C_{21} = C_{21} + T_{2} \)
18 \( C_{11} = C_{11} + T_{2} \)
19 \( T_{1} = B_{12} - B_{22} \)
20 \( C_{12} = A_{11} \times T_{1} \)
21 \( C_{22} = C_{22} + C_{12} \)
22 \( T_{2} = A_{11} + A_{12} \)
23 \( T_{1} = T_{2} \times B_{22} \)
24 \( C_{12} = C_{12} + T_{1} \)
25 \( C_{11} = C_{11} - T_{1} \)

Figure 1: Steps in Strassen implementation

When this block decomposition is applied recursively until the block dimensions reach (or fall below) a threshold value, the arithmetic complexity of Strassen’s algorithm becomes \( O(n^{2.81}) \). Strassen’s method uses the following equations to compute \( C \) with 7 matrix multiplies and 18 add/subtracts.

4. The SUMMA Algorithm

The SUMMA algorithm runs the \( k \) loop sequentially, and parallelizes the \( i \) and \( j \) loops on a two-dimensional grid of processors. It remains to describe how to do a single iteration of the sequential \( k \) loop, i.e. a single update to \( C \), in parallel. The first question (as always) is, where’s the data? We will think of the processors as forming a two-dimensional grid (with \( pr \) rows and \( pc \) columns in the picture on the slides). Each of the matrices \( A \), \( B \), and \( C \) is divided into blocks, one block per processor, each block having size approximately \( n/pr \) by \( n/pc \).
At iteration number \( k \), each processor needs to update its own block of \( C \), for which it requires just part of column \( A(:,k) \) and part of row \( B(k,:) \).

Therefore, at the beginning of iteration \( k \), each of the \( pr \) processors that owns part of column \( A(:,k) \) sends that partial column to each of the other processors in its row of the grid, and similarly each of the \( pc \) processors that owns part of row \( B(k,:) \) sends that partial row to each of the other processors in its column of the grid. One nice thing about SUMMA is that it doesn’t require that \( pr \) and \( pc \) be the same, or that \( n \) be divisible by either one, or even that the matrices \( A, B, \) and \( C \) be square. It’s okay to assume that \( pr = pc = pp \), the matrices are all \( n \)-by-\( n \), and \( n \) is divisible by \( pp \).

There are two ways to improve on the basic algorithm. The first is to work with blocks of rows and columns instead of individual rows and columns. Instead of doing \( n \) iterations of the outer loop, each of which uses one column of \( A \) and one row of \( B \) to do a rank-1 update to \( C \), we pick some block size \( b \) (anything between 1 and \( n/pp \)), and we do \( n/b \) iterations of the outer loop, each of which uses \( b \) columns of \( A \) and \( b \) rows of \( B \) to do a rank-\( b \) update to \( C \). The Matlab pseudocode then becomes

\[
C = zeros(n,n);
for k = 1:b:n \quad \% \text{k goes from 1 to } n \text{ in steps of } b \\
kk = k:(k+b-1); \quad \% \text{kk is } [k \ k+1 \ k+2 \ldots \ k+b-1] \\
C = C + A(:,kk) * B(kk,:); \\
end
\]

Now the inner loop involves multiplying the \( n \)-by-\( b \) matrix \( A(:,kk) \) by the \( b \)-by-\( n \) matrix \( B(kk,:) \) which gives an \( n \)-by-\( n \) matrix that is added to \( C \). The simplest choice of the block size \( b \) is \( n/pp \). This means that the outer loop has \( pp \) iterations, and at each iteration the appropriate processors just send their whole block of \( A \) and \( B \) at once. This is the version you should implement for full credit—you can experiment with other block sizes (including 1) for extra credit. Using a block size different from 1 improves things in two ways. First, it requires fewer communication actions. The same total volume of communication is done, but it’s done with a factor of \( b \) fewer messages. Second, the computation on each processor now involves multiplying two matrices (an \( n/pp \)-by-\( b \) piece of \( A \) and a \( b \)-by-\( n/pp \) piece of \( B \)) instead of two vectors. This sequential matrix multiplication can use blocking (as described in class) to improve its cache behavior. Rather than trying to write a well-blocked sequential matrix multiplication code yourself, I recommend that you just call the sequential BLAS library routine “\( \text{dgemm} \)” to do this multiplication. (TeraGrid has the BLAS library installed; documentation is on the SDSC web pages.) The second improvement to the basic SUMMA algorithm is in the communication pattern. At a particular step \( k \), how does a processor that has a piece of \( A(:,k) \) send it to the other \( pp - 1 \) processors that need that piece?

One way is to do \( pp - 1 \) calls to MPI Send with appropriate destinations. A second way is to do a tree of sends—one processor sends to a second, then each of them sends to another, then each of those four sends to another, until all \( pp \) have gotten the message. This is the same volume of communication, but there’s not the bottleneck of one processor doing \( pp \) sends; nobody does more than \( \log pp = (\log p)/2 \) sends.
A third way is to use the MPI Bcast primitive, but instead of using the MPI COMM WORLD communicator that includes all the processes, set up communicators that include just the processes in one row or one column of the processor grid. This is a bit complicated, but it might give the best performance.

5. Matrix Multiplication on SIMD Processor:
If we increase the number of PE’s used in an array processor to \( n^2 \), an \( O(n \log n) \) algorithm can be devised to multiply the two \( n \times n \) matrices \( A \) and \( B \). Consider an array processor whose \( n^2 = 2^{2m} \) PEs are located at the \( 2^{2m} \) vertices of a \( 2^m \)-cube network. A \( 2^m \)-cube network can be considered as two \((2^m - 1)\)-cube networks linked together by \( 2^m \) extra edges.

An \( O(n \log_2 n) \) algorithm for matrix multiplication

1. Transpose \( B \) to form \( B^t \) over the \( m \) cube \( x2^m-1..x0..0 \) in \( n \log_2 n \) steps
2. \( N \)-way broadcast each row of \( B^t \) to all PEs in the \( m \) cube
   \[ p_{2^m - 1...p_m}^{p_{2^m - 1...p_0}} \]
   in \( n \log_2 n \) steps
3. \( N \)-way broadcast each row of \( A \) residing in PE
   \[ x_{2^m-1..xmpn} - 1...p_{0} \]
   to all PEs in the \( m \) cube
   in \( n \log_2 n \) steps. All the \( n \) rows can be broadcast in parallel.
4. Each PE now contains a row of \( A \) and a column of \( B \) and can form the inner product in \( O(n) \) steps. The \( n \) elements of each result row can be brought together within the same PEs which held a row of \( A \) in \( O(n) \) steps.

The above algorithm takes a total of \( 3n \log_2 n + O(n) \) time steps to complete, which equals \( O(n \log_2 n) \). This demonstrates a gain in speed over the \( O(n^2) \) algorithm at the expense of using \( n^2 \) PEs over the use of only \( n \) PEs in the slow algorithm.

Conclusion:
In this paper we have described the various algorithms that can be implemented in parallel environment. Strassen’s algorithm gives better performance in sequential computing. It can be also implemented in parallel environment. The SUMMA algorithm and SIMD matrix multiplication can be only implemented in parallel environment. Which require more number of PE’s but can give better performance for matrix multiplication.

Reference: