

Efficient Parallel Implementation of the Fox Algorithm

Akpan, Okon H.

Computer Science Department
Bowie State University, Bowie MD
oakpan@cs.bowiestate.edu

Abstract

This paper presents an efficient parallel implementation of the Fox algorithm on a shared-memory supercomputer, the SGI Altix 350. The Fox algorithm is concerned with matrix multiplication, $\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}$. In this study, the matrices are block square matrices with order n and with equal block size $\bar{n} = \frac{n}{\sqrt{p}}$, where p is the number of processors involved in the implementation. The performance of the parallel implementation is acceptable with the speedup, s_p , and efficiency, ϵ_p , approaching their respective upper limiting values of p and 1 only when p is not too small and \bar{n} too large or when p not too large and \bar{n} too small. The implementation is also found to be space-optimal but not time-optimal due to the heavy inter-process data traffic and the consequent dominance of that traffic time, t_{comm} , over the computation time, t_{calc} , especially when p is large.

1 Introduction

Over the past decades, a very large number of algorithms have been proposed for matrix multiplication. The extensive literature resulting from the voluminous studies focused on designing time- and space efficient matrix multiplication algorithm for various computational environments and architectures underscores the importance of this operation in many areas of scientific and engineering applications. Matrix multiplication is not only the kernel operation in linear algebra but it is also the fundamental operation in a large number of scientific applications including computer graphics, combinatorial algorithms, and robotics. The evolution of architectures of parallel and distributed computers keeps fresh the interest of finding

the most efficient and cost-optimal method for matrix multiplication [1, 2, 3, 4, 5, 6, 7]. Also, the existence of mathematical libraries and high performance software packages/libraries as well as the recent efforts at standardization some of them, notably the MPI [8, 9, 10], have spurred studies aimed at obtaining the best performance from a number of important scientific operations including matrix multiplications on supercomputers [11, 12, 13, 14, 15]. The supercomputing libraries/packages widely used include *PVM*, *TCGMSG*, *MPI*, *OpenMP*, *Threads*, *Shmem*, the most popular of which being the MPI which is supported and implemented on almost every architecture.

The conventional serial matrix multiplication of $n \times n$ matrices has asymptotical (run-time) complexity of $\mathbf{O}(n^3)$. Strassen's algorithm [16], which recursively divides the matrices into 2×2 blocks and then multiplies the sub-matrices using 7 scalar multiplications and 18 scalar additions and subtractions, takes $\mathbf{O}(n^{2.8074})$ steps, a significant improvement over the conventional serial matrix multiplication. This means that, theoretically, as $n \rightarrow \infty$, the Strassen's algorithm is a lot faster than the conventional algorithm. One drawback, though, of the Strassen's algorithm is its somewhat lower numerical stability as the order of the matrix becomes very large. However, the search for methods to improve upon the Strassen's algorithm has been going on, and it is likely to keep going on into the distant future time. For example, the Winograd's algorithm, which is a variant of the Strassen's algorithm in that the number of scalar subtractions and additions is reduced from 18 to 15, while having the same run-time complexity as the Strassen's, has a slightly smaller multiplicative constant of the big-O. All these efforts aim at achieving the run-time complexity of $\mathbf{O}(n^\alpha)$, where $2 < \alpha < 3$. Currently, the best known run-time complexity of $\mathbf{O}(n^{2.376})$ is from the Coppersmith and Winograd's algorithm [17].

A large number of matrix multiplication algorithms which have been successfully implemented on both distributed- and shared-

memory parallel computers exist [4, 7, 14, 15, 18, 19, 20]. On distributed-memory processors, research has essentially been focused on parallelization of the conventional matrix multiplication. Dekel, Nassimi, and Sahni [20] have shown that matrix multiplication can be done in $\mathbf{O}(n^3/p + \log(p/n^2))$ time on a hypercube with p processors, where $n^2 \leq p \leq n^3$. It has been demonstrated also that two $n \times n$ matrices can be multiplied under the *CREW PRAM* in $\mathbf{O}(\log n)$ time using $\mathbf{O}(n^{\alpha+\epsilon})$ processors for a fixed positive value of ϵ [18]. It is also reported in [19] that matrix multiplication can be achieved in constant time on a reconfigurable mesh with n^4 processors. It should be mentioned that, although such implementations may be fast, they are usually not cost-optimal.

The Fox [1] and the Cannon's [21] algorithms have been parallelized and implemented on parallel computers. Both Fox and Cannon's algorithms use a number of stages to carry out the multiplication, $\mathbf{C} \leftarrow \mathbf{A} \times \mathbf{B}$, in which the component matrices are square. The Fox algorithm involves row-wise broadcasts of $a_{i,j}$ and column-wise upward shifts of $b_{i,j}$ at every (but the first) stage of computation. The Cannon's algorithm, on the other hand, involves row-wise/column-wise rotations of the sub-matrices and also an inter-leaving of communication and sub-matrix multiplication. Both implementations have been found to be at least memory-efficient.

This study is focused on implementation of the Fox algorithm on a shared-memory multiprocessor using MPI. MPI communication primitives are used to decompose the components matrices and distributing them to the processes, and, then carrying out the parallel computing on the computer. The results of these computation are compared to those of the sequential computing.

2 The Fox Algorithm

2.1 Sequential Fox Algorithm

Given square matrices $\mathbf{A} = (a_{ij})$ and $\mathbf{B} = (b_{ij})$ of order n , the product $\mathbf{A} \times \mathbf{B} = \mathbf{C} = (c_{ij})$ is also a square matrix of the same order, n . The conventional matrix multiplication

$$c_{ij} \leftarrow \sum_{k=0}^{n-1} a_{ik}b_{kj}, \quad i = 0, 1, \dots, n-1, \\ j = 0, 1, \dots, n-1, \quad (1)$$

being of order $\mathbf{O}(n^3)$, is excessively expensive in terms of execution time (and memory too!) especially as $n \rightarrow \infty$. Because the multiplication involves n^2 independently-computed dot products, one obvious solution toward time saving is to parallelize the first two loops thereby reducing the order to $\mathbf{O}(n^2)$, and, consequently, improving the overall execution time. An implementation of the above algorithm works equally correctly, producing the same results, if the matrices \mathbf{A} , \mathbf{B} , and \mathbf{C} are block matrices in which each block is of order \bar{n} and an element, say, a_{ij} , in the algorithm is now an $\bar{n} \times \bar{n}$ block matrix.

The sequential Fox algorithm for multiplication of \mathbf{A} , and \mathbf{B} proceeds in n stages where n , again, is the order of the matrices:

$$\begin{aligned} \text{stage } 0: c_{ij} &\leftarrow a_{ii} \times b_{ij} \\ \text{stage } k: (1 \leq k < n) &: c_{ij} \leftarrow c_{ij} + a_{i\bar{k}} \times b_{\bar{k}j}, \end{aligned} \quad (2)$$

where $\bar{k} = (i + k) \bmod n$. That is, at stage 0, c_{ij} is computed as the product of a_{ii} and b_{ij} , and at stage k , c_{ij} is computed as the product of $b_{i\bar{k}}$ and $b_{\bar{k}j}$, where $\bar{k} = (i + k) \bmod n$.

2.2 Parallel Fox Algorithm

The product of the matrices \mathbf{A} and \mathbf{B} can also be computed in parallel provided there are p processors ($p < n$) available. First, the elements of the operand matrices should be appropriately distributed to the p processors to ensure good load-balance. If \mathbf{A} , \mathbf{B} and \mathbf{C} are

$q \times q$ block matrices in which each of their constituent elements is an $\bar{n} \times \bar{n}$ sub-block, where $q = \sqrt{p}$ and $\bar{n} = \frac{n}{q}$, then the distribution involves routing the component sub-blocks to the appropriate processors. In practice, at every stage $k > 0$, process p_{ij} ($0 \leq i, j \leq q$) computes the sub-block c_{ij} as the product of $a_{i\bar{k}}$ sub-block broadcast by process $p_{i,\bar{k}}$ (on the same row i) and $b_{\bar{k}j}$ obtained from p_{ij} 's southern neighboring process $p_{\bar{k}j}$. Actually, a single south-north toroidal shift at the end of each stage k ensures the routing of the correct $b_{\bar{k}j}$ to processor p_{ij} for its $(k + 1)^{th}$ stage product operation.

2.3 MPI Implementation

2.3.1 Data Distribution

Every supercomputing library or package has several commands with which a matrix can be effectively decomposed and distributed to processors. In MPI [8, 9, 10], one of the simplest ways of doing this is to realize the sub-blocks of **A**, **B**, and **C** matrices as a data object of an MPI's derived data type (MPI_Datatype) and then distributing them to the appropriate processors. This is the data distribution method used in this study.

2.3.2 Cartesian Virtual Topology

Because at any stage $k > 0$ of the parallel Fox algorithm, c_{ij} is computed (by process p_{ij}) as the product of **A**'s sub-block broadcast by process $p_{\bar{k},i}$ on row i to all other processes on the same row and **B**'s sub-block shifted from p_{ij} 's southern neighbor, $p_{\bar{k},j}$, the parallel implementation calls for realizing each row and each column processes as different process groups with their respective communicators. This greatly facilitates the data communication required by the implementation. In MPI, the process's groupings are very easily accomplished in a two-step process: 1) create a Cartesian communicator, say, *Gridcomm*,

from MPI_COMM_WORLD, the MPI's default communicator, 2) from *Gridcomm*, create row Cartesian communicator, *Rcomm*, and column communicator, *Ccomm*, this one with periodicity so as to permit the toroidal shifts of **B**'s sub-blocks from the northern-most process to the southern-most process. No such periodicity is needed among the *Rcomm* row processes.

2.3.3 Parallel Implementation

Figure 1 below shows the code fragment for MPI implementation of the the Fox algorithm in ANSI C. The implementation is based upon the following assumptions:

1. Available are p processors, where p is a perfect square so that $q = \sqrt{p}$.
2. Matrices **A**, **B**, and **C** with order n are $q \times q$ block matrices in which each sub-block is $\bar{n} \times \bar{n}$, where $\bar{n} = \frac{n}{q}$. In the code fragment, the component sub-blocks of matrices **A**, **B**, and **C** are variables with identifier names *aBlock*, *bBlock*, and *cBlock* respectively, all of which sub-blocks having the data type attribute of MPI_Datatype.
3. On the $q \times q$ Cartesian grid, each process on row i and column j ($0 \leq i, j < q$) is in groups of processes having communicators *Rcomm* and *Ccomm*. Also each process has an x-coordinate (*myRow*), a y-coordinate (*myCol*), a row rank, *myRowRank*, a column rank, *myColRank*, and, finally, a northern neighbor (*northNeighbor*), and a southern neighbor (*southNeighbor*).
4. At any stage k , $0 \leq k < q$ and on any row the process, *bCastProc*, which broadcasts its *aBlock* to other processes (on the same row), is the only process with *myRowRank* = *myRow* + $k \bmod q$.
5. *MatMult* is a function which implements the sequential matrix multiplication given in (1).

6. Finally, the broadcast statements in the scopes of the **if** and **else** constructs broadcast $a_{i,\bar{k}}$ to processes on the same row (that is, among the $Rcomm$ processes) while $MPI_Sendrecv$ statement causes a single south-north shift of $b_{\bar{k}j}$ on the same column (that is, among column the $Ccomm$ processes).

```
//STEP 1: Determine neighboring processes,
southNeighbor = (myRow + 1) % q;
northNeighbor = (myRow + q - 1) % q;

//STEP 2: Loop q times to compute cij
for(int k = 0; k < q; k++){
    bCastProc = (myRow + k) % q;
    if(myRowRank == bCastProc){
        MPI_Bcast(&aBlock[0][0], 1, BlockType,
                bCastProc, Rcomm);
        MatMult(aBlock[0][0], bBlock, cBlock);
    }
    else{
        MPI_Bcast(&tempBlock[0][0], 1, BlockType,
                bCastProc, Rcomm);
        MatMult(tempBlock[0][0], bBlock, cBlock);
    }
    MPI_Sendrecv_replace(&bBlock[0][0], 1,
        BlockType, northernNeighbor, sendMsgTag,
        southNeighbor, recvMsgTag, Ccomm);

```

Figure 1: Parallel Fox Program

2.3.4 Computational Environment

The environment used for parallel implementation of the Fox algorithm was the *SGI Altix 350* of the *Alabama Supercomputer Center (ASC)* at Huntsville, Alabama. *Altix 350* is a mid-range 64-bit supercomputing platform built by SGI specifically for scientific computing applications. It has 64-bit *Intel Itanium 2* processors which can easily scale to 128 in a single system image (SSI), and thousands more via clustering using the powerful SGI's

SGI NUMAflex global shared-memory architecture. *Altix 350* supports MPI, SGI's *Message Passing Toolkit* (MPT) for distributed memory parallelism, OpenMP, SHMEM, and POSIX threads for shared memory parallelism.

Altix 350 is built with rack-mountable *bricks*. The *C-brick*, which consists of 2 *Itanium 2* processors and a cache, is the computational module. The front-side buses of the two processors are connected to custom SuperHub (SHub) ASICS and NUMAlink which interface the two processors to the memory DIMMS, to the I/O subsystems, and to other SHubs via the NUMAflex network. The NUMAlink inter-connect channel between all modules in the system creates a single, contiguous memory in the system up to 384GB, and enables each process a direct access to every I/O slot in the system.

The *Alabama Supercomputer Center's Altix 350* has sets of 16 CPUs clustered into shared-memory nodes with *Infiniband* (for message passing between processors), *SHubs ASICS* and *NUMAlink-4* inter-connect (for shared global memory), *fiber channel switch* (for *CXFS* file system data), and with *Gigabit Ethernet*. This Altix cluster has 256 CPUs and a theoretical performance rate of over 380 Gflops.

3 Time Optimal Analysis

3.1 Speedup and Efficiency

Given a parallel algorithm implemented with p processors, the speedup or s_p is given as the ratio

$$s_p = \frac{t_1}{t_p}, \quad (3)$$

where t_1 is the time that a sequential implementation of that same algorithm executes with 1 processor, and t_p the time a parallel implementation of the algorithm takes to execute with p processors. Normally, t_p is a sum

of t_{comm} which the time for inter-processor exchanges (of data exchange, messages, etc.) t_{calc} or the time spent on the actual computation (by the p processors). The speedup is limited as $1 \leq s_p \leq p$. Efficiency, ϵ_p , is given as

$$\epsilon_p = \frac{t_1}{pt_p} = \frac{s_p}{p}.$$

The efficiency is limited by $\frac{1}{p} \leq \epsilon_p \leq 1$.

3.2 The MPI Communication Primitives

The two MPI inter-process communication primitives heavily utilized in the implementations are the *point-to-point* and the *collective* primitives. The model of these primitives is given immediately below.

3.2.1 Point-to-Point Communication

The point-to-point communication primitives are *MPI_Send* and *MPI_Recv* pair and the *MPI_Sendrecv* for sending and receiving a block of m bytes between 2 processors. The model for *MPI_Send* and *MPI_Recv* pair is

$$\gamma + m\beta, \quad (4)$$

where γ is communication latency, and β the communication time per byte. (The reciprocal of β is the *bandwidth*). *MPI_Sendrecv* combination, assumed to take twice as long as the *MPI_Send* and *MPI_Recv* pair, is double the value the time of (4). Therefore, the combined time for the point-to-point primitives is

$$3\gamma + 3m\beta. \quad (5)$$

3.2.2 Collective Communication

The *collective* communication primitive is *MPI_Bcast* in which a block of data of m bytes is broadcast to p processors in a group of p processors involved. If the broadcast is implemented on a linear array of nodes of a minimum span tree, then the broadcast primitive

can be modeled with

$$\lceil \log p \rceil \gamma + \frac{p-1}{p} m\beta. \quad (6)$$

3.2.3 Combined Communication

A combination of (5) and (6) gives a model of the inter-process communication of p processors:

$$(\lceil \log p \rceil + 3)\gamma + \frac{4p-1}{p} m\beta. \quad (7)$$

3.3 Time Computation Complexity

For a sequential multiplication of matrices of order $n \times n$ with a single processor, the computational time is given as

$$t_1 = n^3\alpha, \quad (8)$$

where α is the time for floating point computing. The memory requirement for matrices **A**, **B**, and **C** is each \bar{n}^2s , where $\bar{n} = \frac{n}{\sqrt{p}}$ is the size of the matrix blocks, and s the number of bytes per floating point number.

For the parallel Fox algorithm implemented by p processes the total inter-process communication time, t_{comm} , according to (7), is given as

$$\begin{aligned} t_{comm} &= (\lceil \log p \rceil + 3)\gamma + \frac{4p-1}{p} \left(\frac{m^2}{p} \right) \beta \\ &= (\lceil \log p \rceil + 3)\gamma + \frac{4p-1}{p^2} m^2\beta. \end{aligned} \quad (9)$$

Assuming that t_1 of (8) is distributed equally over the p processors, then the time for parallel implementation with p processors, t_{calc} , is

$$t_{calc} = \frac{t_1}{p} = \frac{n^3}{p}\alpha. \quad (10)$$

The total parallel execution time, t_p , which results from combination of (9) and (10) is

$$\begin{aligned} t_p &= t_{comm} + t_{calc} = (\lceil \log p \rceil + 3)\gamma + \\ &\quad \frac{4p-1}{p^2} m^2\beta + \frac{n^3}{p}\alpha. \end{aligned} \quad (11)$$

The speedup is then given as

$$s_p = \frac{n^3\alpha}{([\log p] + 3)\gamma + \frac{4p-1}{p^2}m^2\beta + \frac{n^3}{p}\alpha}, \quad (12)$$

and the efficiency as

$$\epsilon_p = \frac{n^3p\alpha}{([\log p] + 3)\gamma + \frac{4p-1}{p^2}m^2\beta + \frac{n^3}{p}\alpha}. \quad (13)$$

4 Space Optimal Analysis

The parallel Fox program given in Figure 1 is space-optimal as each processor, at any stage of computation, holds at most $4\bar{n}^2$ data points from matrices **A**, **B**, and **C** – 2 blocks of **A**, 1 block of **B** and 1 block of **C** each block being $\bar{n} \times \bar{n}$. One extra block of **A** on each node is that broadcast from another node on the same row group with communicator *Rcomm*. **B**'s blocks are shared among the same column group processes with communicator *Rcomm*, and there is no extra **B**'s block on any node because, when the **B**'s block involved in the local matrix multiplication on a node is shifted to a northern neighbor, a **B**'s block is shifted to the same node from its southern neighbor at the same time. This shift-out-shift-in operation is carried out by the MPI operation

```
MPLSendrecv_replace(&bBlock[0][0], 1, Block-
Type, northNeighbor, sendMsgTag, southNeig-
hbor, recvMsgTag, Ccomm);
```

in the parallel code, Figure 1. Thus the space requirement $4\bar{n}^2 \ll M$ (M being the memory size of each node) is *optimal* because, matrices **A**, **B**, **C** being square, the memory requirement cannot be further reduced by any partitioning and data distribution scheme.

4.1 Computation Parameters

The following are the estimates of the *Altix 350* *NUMALink*-based parameters for MPI in C:

1. *Bandwidth* (bi-directional) ≈ 520 MB. Hence, the time for a bi-directional transfer of a byte = $\beta \approx \frac{1}{520} = 0.0018\mu s$.

2. *Latency* (send/receive) = $\lambda \approx 7.6\mu s$.
3. *Floating Point Operation Time* = $\alpha \approx 0.002\mu s$.
4. (*sizeof*) *C Data Type* (*double*) = $s = 8$ bytes.
5. (*sizeof*) *MPI Data Type* (*Derived Data for Matrix Blocks*) = $m = 8\bar{n}$ bytes, ($\bar{n} = \frac{n}{\sqrt{p}}$).

5 Experimental Results

The sequential Fox algorithm given in (2) and the parallel Fox algorithm whose MPI code fragment is given in Figure 1 were implemented on the ASC's *SGI Altix 350* for matrix orders (n) 20 to 1,600. The sequential algorithm was implemented on a single processor while the parallel code ran on 4 to 100 processors. The results are given in Figure 2 below.

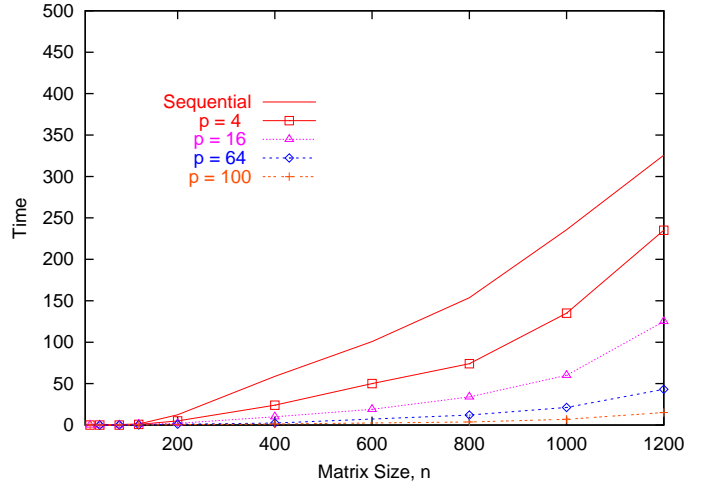


Figure 2: Results of Implementation of Sequential and Parallel Fox Algorithms for Various Values of n and p

6 Observations

Summarized below are the important observations of the performance of the implementations:

- The performance of the parallel Fox program, in terms of the speedup (12) and efficiency (13), very strongly depends on the magnitudes of p and \bar{n} . If p is small and \bar{n} large or p large and \bar{n} small, both s_p and ϵ_p tend to approach their lower limiting values of 1 and $\frac{1}{p}$ respectively. For example, when $p = 4$, $n = 1,200$ ($\bar{n} = 600$), $s_4 = 1.69, \epsilon_4 = 0.42$; when $p = 100$, $n = 80$ ($\bar{n} = 8$), $s_{100} = 1.25$ and $\epsilon_{100} = 0.01$.
- Outside the two extreme situations given above, both s_p and ϵ_p improve but with neither attaining its upper limiting value for any value of p and \bar{n} . For example, when $p = 4$, $n = 100$ ($\bar{n} = 50$), $s_4 = 3.64, \epsilon_4 = 0.92$; when $p = 64$, $n = 800$ ($\bar{n} = 100$), $s_{64} = 23.33$ and $\epsilon_{64} = 0.36$; when $p = 100$, $n = 800$ ($\bar{n} = 80$), $s_{100} = 91.30$ and $\epsilon_{100} = 0.91$.

7 Conclusion

The performance of the parallel implementation depends on the number of processors, p , and the order of the constituent matrix blocks, \bar{n} . The performance improves (but never attains the optimal values of $s_p = p$, $\epsilon_p = 1$) when \bar{n} is not too small or p too large, otherwise, the performance deteriorates. When p is large and \bar{n} small, then t_{comm} or the time for inter-processor data traffic t_{comm} given in (9) dominates the computation time. On the other hand, when p is small and \bar{n} large, t_{calc} (the computation time) dominates but the overall implementation is close to being serial, hence, $s_p \approx 1$. Therefore, realization of a reasonable performance of the method given in this study comes from a judicious choice of both p and \bar{n} . Lastly, even though the performance may improve at moderate choices of the values of \bar{n} and p , the parallel multiplication is recommended only for matrix orders $n \geq 500$ and serial computation for smaller orders because the small matrix orders are not worth the resources demanded by the parallel computation.

8 Acknowledgment

I am very grateful to the administration and management team of the *Alabama Supercomputer Center*, Huntsville, AL which gave me an opportunity to use the wealth of their supercomputing resources to carry out the experimentations reported in this paper, when I taught in the MCIS department of *Jacksonville State University*, Jacksonville, AL. I am also grateful to Dr. Sadanand Srivastava, the chair of *Computer Science Department*, Bowie State University, Bowie, MD, the department I am presently serving, without whose encouragement and support the timely completion of this study would have been impossible.

References

- [1] Fox, G. S. Otto, and A. J. G. Hey *Matrix Algorithm on a Hypercube I: Matrix Multiplication*, *Parallel Computing*, **3**, pp. 17-31, 1987.
- [2] Lederman, S. H., E. M. Jacobson, and A. Tsao, *Comparison of Scalable Parallel Matrix Multiplication Libraries*, Proc. of Scalable Parallel Libraries Conf., IEEE Comp. Society Press, pp. 142 - 149, 1994.
- [3] Agarwal, R. C., F. G. Gustavson, and M. Zubair, *A High Performance Matrix Multiplication on a Distributed Memory Parallel Computer Using Overlapped Communication*, *IBM Journ. Res. Develop.*, **vol. 38**, pp. 673 - 681, 1994.
- [4] Rees, S. A., and J. P. Black, *An Experimental Investigation of Distributed Matrix Multiplication Techniques*, *Software-Practice and Experience*, **vol. 21**(10), pp. 1041 - 1063, Oct. 1991.
- [5] Dongara, J., et al, *Source book of Parallel Computing*, Morgan Kaufmann Pub. Co., San Francisco, 2003.

- [6] Grellck, and Sven-Bolo S., *SAC from High Level Programming with Arrays to Efficient Parallel Computing*, Par. Proc. Letters, **vol** 13(3), pp. 401 - 412, 2003.
- [7] Johnsson, S. L., and C. T. Ho, *Algorithms for Multiplying Matrices of arbitrary Shapes Using Shared Memory Primitives on Boolean Cubes*, Tech. Report TR-569, Yale Univ., New Haven, CT, 1987.
- [8] Message Passing Interface Forum:, *MPI: A Message Interface Standard*, Int'l Jour. of Supercomp. Appl. and High Perf. Comp., **vol.** 8, **no.** 3/4, pp. 165 - 414, Fall/Winter 1994.
- [9] Snir, M. et al, *MPI: The Complete Reference, Volume 1, The MPI Core*, Oxford Univ. Press, New York, New York, 1998.
- [10] Brightwell, R., et al, *Design, Implementation, and Performance of MPI on Portals 3.0*, Int'l Jour. of Supercomp. Appl., **vol.** 17, **no.** 1, pp. 7 - 20, Spring 2003.
- [11] Choi, J., J. Dongarra, and D. W. Walker, *PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers*, Concurrency: Pract. and Exper., **vol.** 6, pp. 543 - 570, Oct. 1994.
- [12] Geijn, R., and J. Watts, *SUMMA: Scalable Universal Matrix multiplication Algorithm*, Tech. Report TR 95-13, Dept. of Comp. Sc., Univ. of Texas at Austin, TX, 1995.
- [13] Parello, D. et al, *On Increasing Awareness in Program Optimization to Bridge the Gap Between Peak and Sustained Processor Performance: Matrix Multiply*, Proceed. of the 2002 ACM/IEEE Conf. on SuperComp., Baltimore, MD., pp. 1 - 11, Nov. 16, 2002.
- [14] Jagadish, and T. Kailath *A Family of new Efficient Arrays of Matrix Multiplication*, IEEE Trans. Comp., **vol.** c-38, pp. 140 - 155, Ja. 1989.
- [15] Siddhartha, C., et al *Recursive Array Layouts and Fast Parallel Matrix Multiplication*, Annual ACM Symp. on Par. Algorithms and Architectures, St. Malo, France, pp. 222 - 231, June 27, 1999.
- [16] Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms*, Comp. Sc. Press, Potomac, MD, 1978.
- [17] Coppersmith, D., and S. Winograd, *Matrix Multiplication via Arithmetic Progressions*, Jour. of Symb. Comp., **vol.** 9, pp. 251 - 280, 1990.
- [18] Bini, D., and V. Pan, *Polynomial and Matrix Computations*, Fundamental Algorithms, Birkhäuser, Boston, 1994.
- [19] Golub, G. H., and C. F. van Loan, *Matrix Computations*, John Hopkins Univ. Press, Baltimore, MD, 1996.
- [20] Dekel, E., D. Nassimi, and S. Sahni, *Parallel Matrix and graph Algorithms*, SIAM Jour. on Comput., **vol.** 10, pp. 657 - 673, 1981.
- [21] Cannon, L. e., *A Cellular Computer to Implement the Kalman Filter Algorithm*, Ph.D. Thesis, Montana State Univ., 1969.