Chain Multiplication of Dense Matrices: Proposing a Shared Memory based Parallel Algorithm

Tirtharaj Dash
Department of Computer Science and Engineering
Veer Surendra Sai University of Technology,
Burla-768018, India

Tanistha Nayak
Department of Information Technology
National Institute of Science and Technology,
Berhampur-761008, India

ABSTRACT
Chain multiplication of matrices is widely used for scientific computing. It becomes more challenging when there is large number of floating point dense matrices. Because, floating point operations take more time than integer operations. It would be interesting to lower the time of such chain operations. Now-a-days every multicore processor system has built in parallel computational power. This power can only be utilized when compatible parallel algorithms were used. So, in this work, a shared memory based parallel algorithms has been proposed to compute the multiplication of a long sequence of dense matrices. The algorithms have been tested with long sequence of matrices as input. The approach has been with 2×10^8 flops. The input matrix sequence length was typically varied from 2 to 30. Maximum number of processors used was eight (Eight core processor). Different parameters like speedup, efficiency etc. were also noted. It was concluded that the parallel algorithms could achieve approximately 90% efficiency at best case. The algorithms also showed improved scalability.

General Terms
Parallel computing, algorithms

Keywords
Chain multiplication, computing, dense matrix, multicore, shared memory, flops, efficiency, speedup, scalability.

1. INTRODUCTION
Chain Multiplication of Matrices or Matrix chain multiplication (MCM) is an optimization problem [1]. In this problem total number of algebraic operations has to be minimized to get to multiplication output of the matrix chain sequence. This problem can be solved using dynamic programming. Mathematical definition of matrix multiplication can be given as, the product C = AB of a x×y matrix A and y×z matrix B is a x×z matrix given by,

$$C(i,j)=\sum_{k=1}^{y}(A(i,k)\times B(k,j)); \quad for\ 1\le i\le x\ and\ 1\le j\le y \qquad (1)$$

The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications with minimal number operations (multiplications). As matrix multiplication possesses associative property, one can multiply the chain of matrices in any order. Again this order of multiplication will decide the efficiency of the multiplications (number of operations performed). So, parenthesization for each multiplication needs to be carried out.

Consider an example; there are five matrices in a sequence A, B, C, D and E to be multiplied. Let consider that the matrices are having compatible rows and columns for multiplications as following.

Matrices sequence: A(100,50), B(50,200), C(200,70), D(70,150) and E(150,300). For a sequence of five matrices, there can be more than five ordering sequence and corresponding number of multiplications (NoM) to be carried out. Different order of multiplications (parenthesization) could be,

Order-1: ((AB)(CD))E
Order-2: (AB)(C(DE))
Order-3: A(((BC)D)E)
Order-4: ((AB)C)(DE)
Order-5: (A((BC)D))E and so on.

For the order-1, NoM can be calculated as,
NoM=(100×50×200)+(200×70×150)+(100×200×150)+(100×150×300)=10600000 operations.

Similarly for orders 2, 3, 4 and 5, NoM will be 14350000, 4975000, 7650000 and 6475000. So, ordered parenthesization-3 requiring lesser multiplications to compute the multiplication of given matrix sequence. In this case, it is to decide order-3 to be best, because lesser number of matrices is considered in sequences and less number of orders. But, it will be too difficult for a human brain to propose a parenthesization order for a sequence having hundred or thousand matrices of larger size. One can go through each possible parenthesization (Brute force method). But, this method will take $O(2^n)$ time, which is too much *slow* for large values of n. Again, if the matrices are dense and contains floating point data, then the multiplication puts more overhead over the processor. It also takes more CPU cycle to compute multiplication of two floating point data than multiplying two integer data. So, it will be interesting to develop some time efficient algorithm to fasten this task. This research work basically focuses on a shared memory based parallel algorithm for multiplication of a sequence of large sized dense matrices with floating point data elements. Rigorous testing has been carried to get into a conclusion for the same.

The next sections describe the MCM problem in brief and revolution in the methods of sequence multiplication. This is further followed by the proposed algorithm and analysis of the result so obtained. Finally, the work is concluded in further section. Different citations in this paper are referred in the reference section.

2. MATRIX CHAIN MULTIPLICATION
One solution to above problem is memorization. In this technique, each time a minimum cost is computed, it is saved into memory (memorizing the value). If again the computation is needed, the saved answer is used without re-computing it. Since there are about $n^2/2$ different subsequences, where n is the number of matrices, the space

required to do this is reasonable. It can be observed that this method of memorization brings the runtime down to $O(n^3)$ from $O(2^n)$, which is more than efficient enough for real world applications. This is algorithmically termed as *top-down dynamic programming* [1, 2, 3]. A recursive algorithm for this kind of problem can have following steps [3].

1. Take the sequence of matrices and separate it into two subsequences.
2. Find the minimum cost of multiplying out each subsequence.
3. Add these costs together, and add in the cost of multiplying the two result matrices.

Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

3. EMPIRICAL STUDY ON PREVIOUS WORKS

Although the dynamic programming methodology to MCM proves itself to be efficient for parenthesization of the sequence, many researches are going on to reduce the time complexity of this algorithm further from $O(n^3)$ down to some lower order $(\leq n^3)$. An algorithm published by Hu and Shing in the year 1984 proved that the MCM problem will take $O(nlogn)$ time for parenthesization. In their algorithm, they were able to transform the problem into a *problem of partitioning* [2]. The process was like transforming a convex polygon into a set of non-interesting triangles. Hu and Shing developed the algorithm to find the optimal solution of this portioning [2].

Matrix multiplication has also faced much revolution in this research based world and has been applied in many mathematical and scientific problems. Dou et al. with his co-workers proposed 64-bit floating point FPGA matrix multiplication [4]. In this work they introduced a 64-bit IEEE/ANSI standard 754-1985 floating point design of a hardware matrix multiplier. The multiplier was optimized basically for FPGA implementations. A fast scalable universal matrix multiplication algorithm was proposed by J. Choi in 1997 [4, 5]. One approach to fast multiplication of sparse matrices was proposed Yuster and Zwick in 2004. They presented a new algorithm which could multiply two matrices in $O(m^{0.7}n^{1.2}+n^{2+o(1)})$ time [6]. Matrix multiplication has been applied in many applications. Some of the most critical applications can be found in [7, 8, 9, 10]. Fast matrix multiplication could be applied to find simple and small cycles in a graph [11, 12, 13], subgraphs [13, 14], finding shortest path in the graph [6, 7, 15], string matching problems [10, 16].

Matrix chain product and optimal triangulation problem was solved using parallel algorithm as proposed by Czumaj. In this work, the author tried to reduce the problem to computing certain recurrences in a tree [17]. The work concluded that the proposed parallel algorithm ran in $O(lon^3n)$ time using n^2/log^3n processors on a CREW PRAM. They also showed an efficient algorithm for the triangulation problem in a monotone polygon which could run in $O(n^2)$ time. This value was close to the result provided by [18].

4. PROPOSED ALGORITHM

It is already mentioned that, traditional approach (Sequential) to MCM problem using dynamic programming [1, 19] faces critical stage when a sequence of floating point matrices are supplied. There have been many parallel algorithms proposed for matrix multiplication. But, it will be nice to introduce some parallel algorithms for matrix chain product. There can be two approaches to this *(i) shared memory based algorithm* and *(ii) distributed memory based algorithm* [20]. The former type of algorithm can be implemented on shared memory architecture and the corresponding performance can be noted. The hardware organization of such machine is outlined in the architectural diagram in Figure-1 below [21, 22].

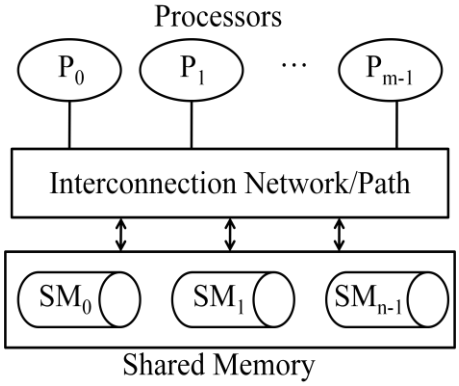

Figure 1 Shared memory architecture

The designed parallel algorithm and its implementation have been discussed below. The parallel algorithm makes the use of a middle-wire (will be discussed in next sections) in its implementation. The complete algorithm can be divided into two sub-algorithms.

1. *Parallelizing parenthesized sequence* (after chain ordering & optimal parenthesization algorithm)
2. *Parallelizing matrix multiplication* (for multiplication of two large dense floating point matrices)

4.1 Parallelization of parenthesized sequence

The developed algorithm to parallelize the parenthesized sequence is given in Algorithm-I below.

Algorithm-I Parallelize_Parenthesized_Sequence(input)

Input: Parenthesized sequence
Output: Calls to parallel matrix multiplication algorithm, reduced parenthesized sequence

Algorithmic Steps:

1. Take the input parenthesized sequence and find out the length of it (len)
2. Set the current processor as *Master* (theoretically processor-0/P_0)
3. In *Master*, search the parenthesized matrices to be multiplied (with in open-close parenthesis). Eg. (A_1A_2) or (A_5A_6)
4. Send the found set to slave processors for further computation.
   Eg. (A_1A_2) to *Processor-1 (Slave-0)* or (A_5A_6) to *Processor-2 (Slave-1)*
5. In *Slave-0*: Calls to Parallel matrix multiplication algorithm *(Algorithm-II)* setting the slave processor to master for *Algorithm-II.*
6. Repeat step-5 for other slaves also.
7. Continue steps 3 to 6 until the input sequence is ended with the pattern **(X)**; where **X** is the single matrix (output).

Example:
Let's consider an parenthesized sequence (((A_1 A_2) A_3) ((A_4 (A_5 A_6)) A_7)).

If this sequence is served as input to the Algorithm-I, the corresponding parallel intermediate steps will be as demonstrated in figure (Figure -2) below.

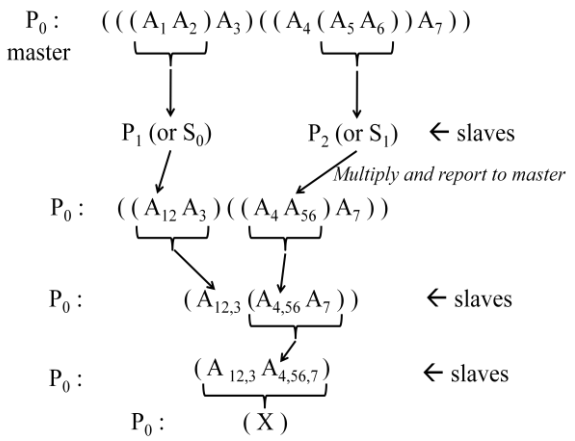

Figure 2 Figure demonstrating algorithmic steps in Algorithm-I showing master and slaves

As demonstrated in Fig-2, the input sequence will reduce to a single matrix (X) in the master processor (processor-0). The slaves getting the input matrices (two matrices) will multiply the input matrices using parallel matrix multiplication algorithm as proposed in *Algorithm-II*. However, these parallel algorithms are developed using OpenMP middleware package. This package performs to its best in shared memory architecture based on program developed by the programmer. Other specifications are given in a separate section later on.

4.2 Parallelization of Matrix Multiplication

It is already mentioned that parallel implementation is completely based on shared memory architecture using an API package OpenMP [23]. The library works based on the instruction *(directive)* written by the programmer. The programmer exclusively specifies which loop has to be parallelized and the corresponding parallel instructions. But, user/programmer does not perform the duty of sending and receiving instructions to and from the processor respectively. The library itself does the task reducing overhead of the programmer. The main advantage of this library is that it can be easily used to parallelize a sequential program in shared memory architecture, as currently most of the multicore systems are developed with such architecture. The implementation has been carried out in 'C' programming language [23, 24, 25]. Dynamic memory allocation has been used to avoid memory failure during matrix initialization for testing purpose as well as for efficiency of the code. In this algorithm, each processor will be considered as a thread. Thread is the smallest executable unit of a program [25, 26]. Each processor is assigned a single independent task at a time. Input to the algorithm, output and algorithmic steps are discussed below.

Algorithm-II Parallelize_Matrix_Multiplication(input)

Input: Two compatible floating point matrices (A, B)
Output: Matrix C (=A×B)

Algorithmic Steps:

1. Set number of threads by using following command in Linux terminal.
   $ export OMP_NUM_THREADS = 4
2. Set the current processor as *Master* (theoretically processor-0/P_0)
3. Create a parallel region and declared the used variables as private or shared based on requirement and scope.
   #pragma omp parallel shared(var_1, var_2, chunk) private(thread_id,iteratiors)
4. In *Master*, initialize the matrices A, B and C in parallel fashion as given below.

```
#pragma omp for schedule (static, chunk)
for(i=0;i<ROW; i++)
{
    for(j=0;j<COLUMN; j++)
    {
        A[i][j] = 0.80*i + 3.451*j;
    }
}
```

5. Multiply the matrices in parallel. (Master will call to slaves)

```
#pragma omp for schedule (static, chunk)
for(i=0;i<A_ROW; i++)
{
    Print "Thread (tid)"
    Print " computing row(i)"
    for(j=0;j<A_COLUMN; j++)
    {
        for(k=0;k<B_COLUMN; k++)
        {
            C[i,j]=C[i,j] + (A[i,k]*B[k,j]);
        }
    }
}
```

6. Ouput time in *Master:*
   $ time ./output_file
   The output will come like the following.

real	1m20.067s
user	1m30.453s
sys	0m0.098s

The time in parallel algorithm can also be calculated using *omp.h* library function *omp_get_wtime().* This returns the total number wall clock ticks used for the execution of the section [23, 24, 25]. The next section explains our device specification in which the developed algorithms were tested.

5. DEVICE SPECIFICATION

For the execution of the algorithms (Sequential and Parallel programs) we used a system which had a processor *of* 2GHz speed. The processor had *eight (8) cores* and the system had *4GB of main memory storage*. As mentioned, the proposed approach has been tested with up to 2×10^8 *flops (floating point operations)*. The input matrix sequence length was also varied from 2 to 30. This sequence length also affects the number of *flops* to be carried out. Performance of our algorithms is also evaluated with many parameters. The next section describes the results, time complexities, speed up, scalability [20, 27, 28, 29] and other crucial issues [30].

6. PERFORMANCE EVALUATION

It is very much crucial and mandatory process to test any algorithm based on time and space complexities. The analysis will be basically worst case analysis, so that the considered algorithm will prove itself to be efficient with any worst case inputs. In addition to this, for a parallel algorithm the major parameters to be considered in this section are ***(i)*** *Experimental serial fraction (e)* ***(ii)*** *Speed up* ***(iii)*** *Efficiency and* ***(iv)*** *Scalability*. The first parameter, *e* determines how much part of the sequential code could be successfully parallelized and its effect on the output. These parameters will be considered for the whole program which consisted of

Algorithm-I and Algorithm-II both. These terms are defined in following equations (Equations 2-4).

$$Experimental\_serial\_fraction(e) = \frac{\frac{1}{Speedup} - \frac{1}{No\_of\_processors}}{1 - \frac{1}{No\_of\_processors}}$$

$$e = \left(\frac{1}{Speedup}\right)\frac{No\_of\_processors - Speedup}{No\_of\_processors - 1} \quad (2)$$

$$Speedup = \frac{Sequential\_execution\_time}{Parallel\_execution\_time} \quad (3)$$

$$Efficiency(\mu) = \left(\frac{1}{No\_of\_proessors\_used}\right) \times \frac{Seq\_Exe\_Time}{Par\_Exe\_Time}$$

$$\mu = \left(\frac{1}{No\_of\_processor\_used}\right) \times Speedup \quad (4)$$

The sequential as well as the parallel programs were tested with matrix sequence of varying length and varying matrix dimensions (multiplication compatible). Tables 1 & 2 show performance of the algorithms with ten test cases. It is not possible to incorporate all the test cases in this paper due to space constraints. It should be noted that the execution time showed in the tables are average of ten executions.

Table 1 Table showing input matrix dimension sequence and output optimal parenthesized sequence (Each sequence is given a ID for future reference purpose)

Sequence ID	Sequence length	Dimension array	Optimal Parenthesized sequence *(Output)*
1	5	[50 100 200 70 80]	$(((A_1A_2)A_3)A_4)$
2	6	[100 80 90 220 70 80]	(A1((A2(A3A4)) A5))
3	7	[100 200 50 85 90 100 80]	((A1A2)(((A3A4)A5)A6))
4	8	[60 80 100 300 90 100 120 130]	(((((A1A2)(A3A 4))A5)A6)A7)
5	9	[200 50 60 100 60 80 90 250 100]	(A1(((((A2(A3A 4))A5)A6)A7)A 8))
6	10	[50 100 50 80 100 200 300 100 50 80]	(((A1A2)(A3(A4 (A5(A6(A7A8))))))A9)
7	10	[50 60 100 200 300 90 100 120 250 110]	(((((((A1A2)A3)A4)A5)A6)A7) A8)A9)
8	12	[100 80 100 120 50 70 90 120 35 70 90 100]	((A1(A2(A3(A4(A5(A6(A7A8)))))))((A9A10)A11))
9	12	[120 120 80 100 300 210 56 22 10 20 30 100]	((A1(A2(A3(A4(A5(A6(A7(A8)))))))((A9A10)A1 1))
10	15	[100 100 50 90 80 10 120 120 11 90 80 70 120 45 56]	((A1(A2(A3(A4 A5))))(((((((A6 A7)A8)A9)A10) A11)A12)A13)A 14))

Table 2 Table showing total number of flops needed for multiplications and corresponding performance of program with different number of used processors

Sequence ID	Number of flops needed to multiply the parenthesized matrices *(flops)*	Execution time (millisecond)			
		Sequential Program	Parallelized Program		
			2 processors	*4 processors*	*8 processors*
1	1980000	670000	553700	310185	109299
2	2978000	810000	637000	288260	131089
3	2895750	890000	671000	303754	143317
4	5916000	1500000	987866	480769	233281
5	4515000	1200000	815400	352941	179453
6	12800000	1900000	1353100	590062	279371
7	9575000	1300000	817000	401235	205404
8	2796500	950000	972463	31772	135695
9	3177040	1200000	1015006	364741	189573
10	718600	1305000	1089766	427868	188584

It should be mentioned that *speedup* is a major concern when parallel algorithm is considered. Scalability of any parallel algorithm is directly dependently on speedup of the algorithm. So, it will be much crucial to analyze speedup [20, 23] achieved with different number of processors used for the implantation. Figure-3 shows scalability curves for two, four and eight number of processors. Figure 4 & 5 show analysis result for other important parameters as mentioned earlier in this section.

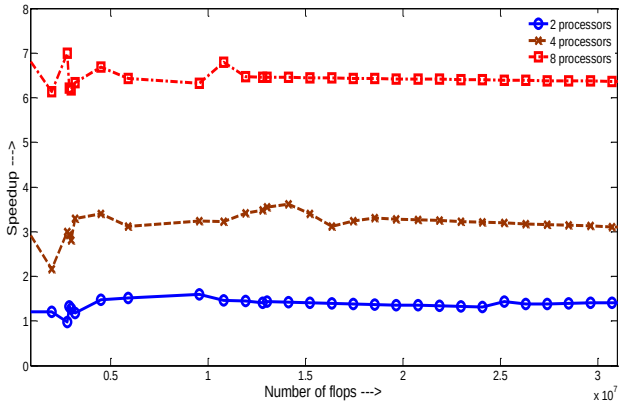

Figure 3 Figure showing number of speedup achieved with different number of processors

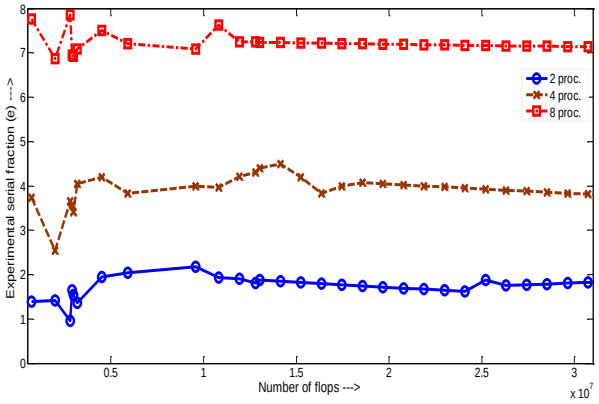

Figure 4 Plot showing experimental fraction with different number used processors

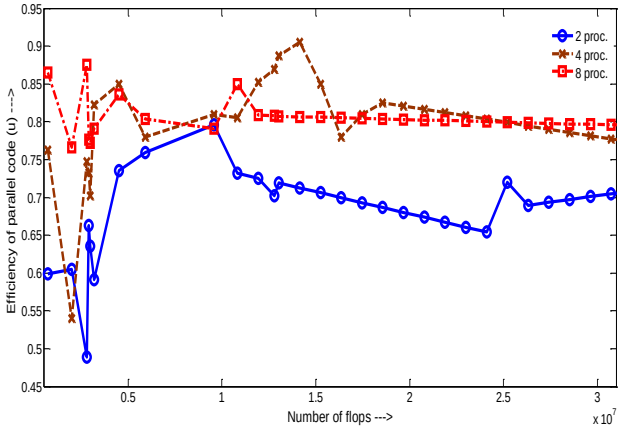

Figure 5 Efficiency achieved by parallel program for different processor system configurations

The performance characteristics showed in figures 3-5 helps us to analyze the developed parallel algorithms. The main focus will be on efficiency characteristics (see Figure 5). This characteristic is dependent on time and space complexities as discussed in next subsection.

6.1 Time and Space Complexity analysis

The matrix chain ordering algorithm performs with running time of $O(n^3)$ due to three nested loop structure and each loop will iterate for atleast n-1 times, where n is the length of the input matrix sequence [1, 19]. And this algorithm uses $\Theta(s^2)$ to store the value matrices, where *s* is the row or column size of a matrix. To call the matrix multiplication algorithm, the optimal sequence function takes a running time complexity of $O(n^2)$. So, for each call the sequential matrix multiplication yields $O(m^3)$, where m is the size of the row or column of the argument matrices. So, the total time to perform the multiplication is $O(n^2m^3)$. So, the total time complexity of the sequential algorithm can be written as,

$$T_s(n,m) = O(n^2) + O(n^2m^3)$$
$$\Rightarrow T_s(n,m) = O(n2[1+m^3])$$
$$\Rightarrow \boldsymbol{T_s(n,m) = O(n^2m^3)} \quad (5)$$

For a sequence of length 'n', total space required to store the matrices is $\Theta(nm^2)$. The shared memory algorithms worked with different number of processors used *(p)*. So, depending on this parameter and memory storage capacity of the system the time complexity will vary in a parallel environment [20, 24]. If *p* processors were used during execution of the parallel program, then the total time complexity will be reduced approximately by the term $\boldsymbol{p^2}$. This is because, one processor works with *n/p* rows/column of the matrices. So, multiplication of two matrices will be completed in *(Total time required)/*p^2. So, time complexity of parallel program can be expressed as,

$$T_p(n,m,p) = O(\frac{n^2m^3}{p^2}) \quad (6)$$

Equation-6 will be much significant when $p=n$ which makes the time complexity to be equal to $O(m^3)$. However, it is not guaranteed that this will happen always with the parallel algorithms. Depending on the communication between processors [20, 21, 23], it will vary significantly. If there is more inter-processor communications and less computations then execution time will be more. So, this is also the parameter which indirectly affects efficiency of the parallel algorithms.

7. CONCLUSION

In this paper, a shared memory based parallel algorithm to multiply a chain of large floating point dense matrices has been proposed. Two algorithms were proposed to (i) parallelize the parenthesized sequence and to (ii) parallelize the matrix multiplication. From analysis it was shown that, the parallel algorithms perform to their best when there is less communication among processors. Time complexity was found out to be $O(m^3)$ in best case $(n=p)$ and $O(\frac{n^2m^3}{p^2})$ in worst case. It was also shown that efficiency of parallel program touched approximately *90%* with four and eight processors. But, with two processors, the algorithm performed with 80% efficiency only when number of flops is limited. So, a user with only 2 processors may not be able to get chain multiplication output with maximum efficiency. However, the algorithms can be used for shared memory systems with more than eight cores very efficiently.

8. ACKNOWLEDGMENTS

The authors would like to thank all the persons who were directly or indirectly associated with this work. They would also thank those who helped to improve this work. The authors also express their sincere thanks to anonymous reviewer of IJCA for reviewing this work and providing necessary comments.

9. REFERENCES

[1] Cormen, Thomas H.; Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009). "15.2: Matrix-chain multiplication". *Introduction to Algorithms*. Second Edition. PHI Learning Private Limited. pp. 331-338. ISBN 978-81-203-2141-0.

[2] Hu, T C.; M T. Shing (1984). Computation of matrix chain products. Part II. *SIAM Journal on Computing (Univ. of California at San Diego: Springer-Verlag)* 13 (2): 228–251. doi:10.1137/0213017. ISSN 0097-5397.

[3] Kreyszig, E; Advanced Engineering Mathematics. Wiley-India Edition. 8th edition. ISBN: 978-81-265-0827-3.

[4] Dou, Y.; Vassiliadis, S.; Kuzmanov, G. K.; Gaydadjiev, G. N.; 64-bit Floating-Point FPGA Matrix Multiplication. *FPGA'05, February 20–22, 2005, Monterey, California, USA*, pages 86-95.

[5] Choi, J.; A Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. *In11th IEEE International Parallel Processing Symposium* (IPPS '97), pages 310–314, April 1997.

[6] Yuster, R.; Zwick, U.; Fast sparse matrix multiplication. *In the proceedings of the 12th Annual European Symposium on Algorithms (ESA'04).*

[7] Demetrescu, C.; Italiano, G.F.; Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. *In Proc. of 41st FOCS, pages 381–389, 2000.*

[8] Roditty, L; Zwick, U; Improved dynamic reachability algorithms for directed graphs. *In Proc. of 43rd FOCS*, pages 679–688, 2002.

[9] Mulmuley, K; Vazirani, U. V.; Vazirani, V. V.; Matching is as easy as matrix inversion. *Com-binatorica*, 7:105–113, 1987.

[10] Rabin, M.O; Vazirani, V.V.; Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10:557–567, 1989.

[11] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42:844–856, 1995.

[12] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:209–223, 1997.

[13] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. *In Proc. of 15th SODA*, pages 247–253, 2004.

[14] J. Neˇsetˇril and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathe-maticae Universitatis Carolinae*, 26(2):415–419, 1985.

[15] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. *In Proc. of 43rd FOCS*, pages 679–688, 2002.

[16] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.

[17] Czumaj, A; Parallel Algorithm for Matrix Chain Product and the Optimal Triangulation Problems (Extended abstract). *STACS'93 version*. *Supported in part by the EC Cooperative Action IC 1000 Algorithms for Future Technologies "ALTEC" and by the grant KBN 2-1190-91-01*, Pages 1-12.

[18] T.C. Hu, M.T. Shing; Some theorems about matrix multiplications, *FOCS 1980*, pp. 28-35.

[19] E. Horowitz, S. Sahani, S. Rajasekaran; Fundamental of Computer Algorithms, 2nd edition. *Universities Press (India) Private Limited* (2008), ISBN: 81-7371-612-9.

[20] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, Parallel programming in OpenMP, *Morgan Kaufmann Publisher* (2001). ISBN 1-55860-671-8.

[21] D. A. Patterson, J. L. Hennessy; Computer Organization and Design: The Hardware/Software Interface ARM edition. 4th edition. *Morgan Kaufmann Publisher* (2009), ISBN: 978-81-312-2274-4.

[22] C. Hamacher, Z. Vranesic, S. Zaky; Computer Organization. International Edition 2002. *McGraw-Hill Higher Education*, ISBM: 007-120411-3.

[23] M.J. Quinn; Parallel Programming in C with MPI and OpenMP. International Edition 2003. *McGraw-Hill Higher Education*, ISBM: 007-282256-2.

[24] T. Dash, T. Nayak, S. Chattopadhyay; Offline Handwritten Signature Verification using Associative Memory Net. *International Journal of Advanced Research in Computer Engineering & Technology*, Volume 1, Issue 4, pp. 370-374, 2012.

[25] T. Dash, S. Chattopadhyay, T. Nayak; Handwritten Signature Verifications Using Adaptive Resonance Theory Type-2 (ART-2) Net. *Journal of Global Research in Computer Science*, Volume 3, No. 8, pp. 21-25, 2012.

[26] A. Silberschatz, P.B. Galvin, G. Gagne; Operating System Concepts. International student version, 8th edition. *John Wiley & Sons Inc., U.K.*, ISBN: 978-81-265-2051-0.

[27] M.A. Ismail, S.H. Mirza, T. Altaf; Concurrent Matrix Multiplication on Multi-Core Processors. *International Journal of Computer Science and Security (IJCSS)*, Volume (5): Issue (2): 2011, pp. 208-220.

[28] T. Dash, T. Nayak, S. Chattopadhyay; Handwritten Signature Verification (Offline) using Neural Network Approaches: A Comparative Study. *International Journal of Computer Applications.* ISSN: 0975-8887, November'12. *(accepted, in press)*

[29] T. Dash; Time Efficient Approach to Offline Hand Written Character Recognition using Associative Memory Net. International Journal of Computing and Business Research. ISSN: 2229–6166, Vol. 3, Issue-3, 2012.

[30] T. Dash, T. Nayak, S. Chattopadhyay; Offline Verification of Hand Written Signature Using Adaptive Resonance Theory Net (Type-1). In Proc. of ICECT-2012, Vol-2, pp.205-210.