Optimal Solution to Matrix Parenthesization Problem Employing Parallel Processing Approach

MUHAMMAD HAFEEZ, DR. MUHAMMAD YOUNUS, ABDUR REHMAN, ATHAR MOHSIN

Computer Science Department, College of Telecommunication Engineering
(National University of Sciences and Technology), Hamayun Road, Rawalpindi - PAKISTAN

Abstract: - Optimal matrix parenthesization problem is an optimization problem that can be solved using dynamic programming. The paper discussed the problem in detail. The results and their analysis reveal that there is considerable amount of time reduction compared with simple left to right multiplication, on applying the matrix parenthesization algorithm. Time reduction varies from 0% to 96%, proportional to the number of matrices and the sequence of dimensions. It is also learnt that on applying parallel matrix parenthesization algorithm, time is reduced proportional to the number of processors at the start, however, after some increase, adding more processors does not yield any more throughput but only increases the overhead and cost. Major advantage of the parallel algorithm used is that it does not depend on the number of matrices. Moreover, work has been evenly distributed between the processors.

Key-Words: - Matrix Parenthesization Problem Parallel Processing Algorithm

1 Introduction

In most systems there are many processes that are running simultaneously. Recall that multiplying an x x y matrix by a y x z matrix creates an x x z matrix. Thus multiplying a chain of matrices from left to right might create large intermediate matrices, each taking a lot of time to calculate. Matrix multiplication is not commutative, but it is associative, so the chain can be parenthesized in whatever manner deemed best without changing the final product. A standard dynamic programming algorithm can be used to construct the optimal parenthesization. Note that optimizing is over the sizes of the dimensions in the chain, not the actual matrices themselves.

The problem is not actually to perform the multiplications, but merely to decide in what order to perform the multiplications. For example, if there are four matrices A, B, C, and D, there may be:

((AB)C)D=(AB)(CD)=A((BC)D)=(A(BC))D=A(B(CD))

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose to multiply a sequence of matrices with dimensions A(30 × 1), B(1 × 40), C(40 × 10) and D(10 x 25). Multiplying an X x Y matrix by a Y x Z matrix takes X x Y x Z number of multiplications. The number of arithmetic operations required for three different parenthesizations are:

((AB)C)D=30x1x40 + 30x40x10 + 30x10x25= 20,700
(AB)(CD)=30x1x40 + 40x10x25 + 30x40x25= 41,200
A((BC)D)=1x40x10 + 1x10x25 + 30x1x25 = 1,400

Clearly the last method is the more efficient. Now that the problem is identified, how to determine the optimal parenthesization of a product of n matrices? One of the way is to go through each possible parenthesization (brute force), but this would require time $O(2^n)$, which is very slow and impractical for large n. The solution, is to break up the problem into a set of related subproblems. By solving subproblems one time and reusing these solutions many times, the time required is reduced drastically. This is known as dynamic programming [1][2].

The matrix-chain multiplication problem can be stated as follows: given a chain $(A_1, A_2,\ldots,A_n)$ of n matrices, where for i = 1, 2,…,n, matrix A; has dimension p_{i-1} x p_i, fully parenthesize the product $A_1, A_2,\ldots,A_n$, in a way that minimizes the number of scalar multiplications. Note that in the matrix-chain multiplication problem, matrices are not actually multiplied; rather the goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 1,400 scalar multiplications instead of 41,200 multiplications).

Undermentioned standard pseudocode assumes that matrix A; has dimensions p_{i-1} x p_i for i = 1,2,…,n. The input is a sequence $p = (p_o, p_1,\ldots p_n)$, where length[p] = n+1. The procedure uses an auxiliary table m[1…n, 1…n] for storing the m[i, j] costs and an auxiliary table s[1…n,1…n] that records which index of k achieved the optimal cost in computing m[i, j]. The table s is used to construct an optimal solution [3][4][6].

2 Matrix Parenthesization Algorithm

```
n ← length[p]-1 {p is an array containing pi-1 to pj
                 and n is number of matrices in chain}
for i ← 1 to n
  do m[i,j]←0 {Single matrices take 0 multiplications}
for l ← 2 to n   {l is length of chain}
   do for i ← 1 to n – l + 1    {All possible starting
                                 indices for length l}
      do j ← i + l – 1{Ending index of chain of length l}
         m[i,j] ← INF  {Large value to start to
                        find minimum}
         for k ← i  to j –1  {Try all possible splits of
                              this chain}
      do q ← m[i,k]+m[k+1,j]+ p_{i-1}p_k p_j
      {Smaller chains are already computed}
         if q < m[i,j] {If minimum, then store it}
            then m[i,j] ← q
                 s[i,j] ← k
   return m, s
```

Table 1 : Completed Arrays m and s

m

	1	2	3	4
1	0	224	180	216
2		0	84	120
3			0	63
4				0

s

	2	3	4
1	1	1	1
2		2	3
3			3
4			

Table 1 represents the application of the algorithm for four matrices with dimensions 8 x 4, 4 x 7, 7 x 3 and 3 x 3. Top most right entry represents the optimal parenthesizations. Figure 1 represents the corresponding dynamic programming formulation for finding an optimal matrix parenthesization for this chain. A square node in the figure represents the optimal cost of multiplying a matrix chain. A circle node represents a possible parenthesization.

2.1 Analysis of Implementation of Algorithm

The results for implementation of algorithm for optimal solution to matrix parenthesization problem are shown in Table 2 and Figure 2. It is evident that there is considerable amount of time reduction proportional to the number of matrices and the sequence of dimensions on applying the Matrix Parenthesization Algorithm. It also seems that percentage of time reduction to the linear left to right arithmetic operations is less, if the first dimension is smaller. Similarly, if the first dimension is larger, percentage of time reduction to the linear left to right arithmetic operations is more. It is because of the reason that in linear left to right arithmetic multiplication, first dimension keeps on multiplying with all of the rest of the dimensions. So if the first dimension is larger, it gives larger linear left to right arithmetic multiplication value.

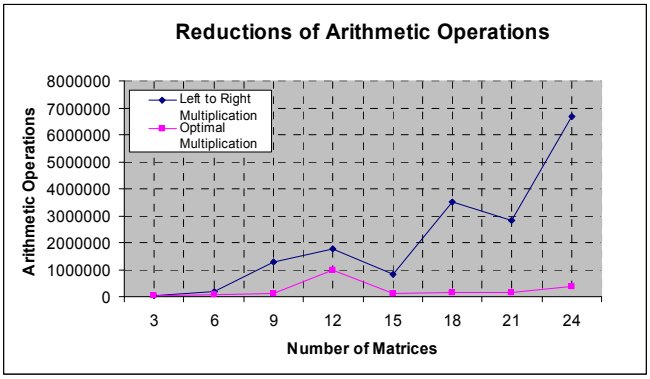

Figure 2:Reductions of Operations in Optimal Solution No. of Matrices:1-24, Sequence of Dimensions:1-100

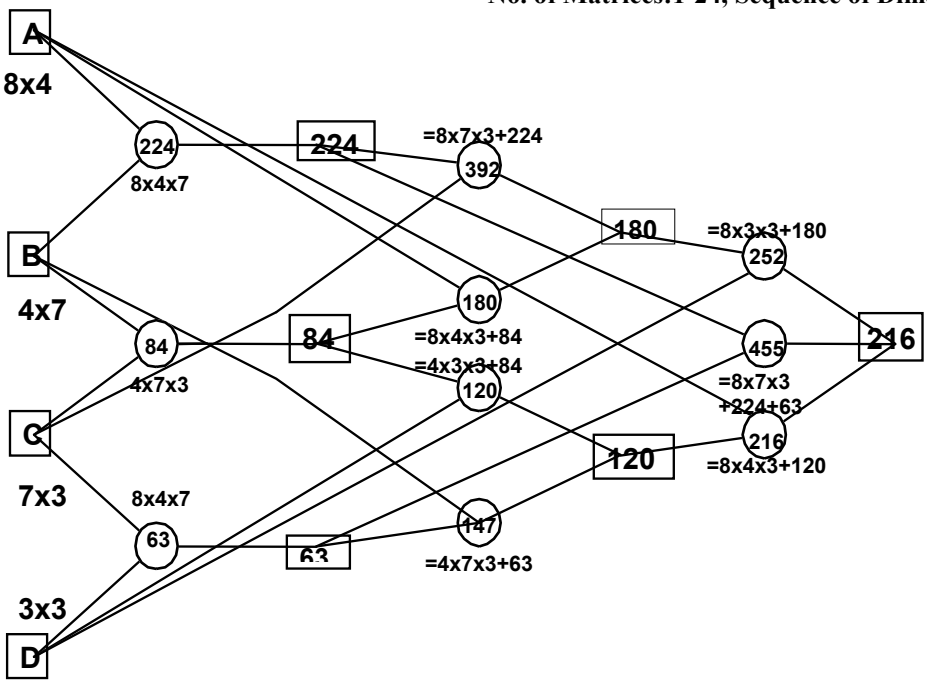

Figure 1: Optimal Matrix Parenthesization for a Chain of Four Matrices

Table 2: Implementation of Matrix Parenthesization Algorithm
No. of Matrices: 1-24, Sequence of Dimensions: 1-100

No. of Matrices	Sequence of Dimensions	Optimal Arithmetic Multiplica-tions	Left to Right Multiplica-tions	Optimal Parenthesizations	%age Reduction of Time (d-c)/d*100
a	b	c	d	e	f
3	9,95,21,78	32697	32697	(AB)C	0
6	30,10,71,58,9,25,22	56982	183750	A((B(CD))(EF))	69
9	94,67,56,17,80,68,10,78,7,5	98220	1273230	A(B(C(D(E(F((GH)I))))))	92
12	42,54,49,22,62,46,93,97,82,59,24,86,56	970214	1777734	((A(BC))((((((DE)F)G)H)I))J))(KL)	45
15	27,98,89,40,36,82,6,11,3,23,15,91,87,35,3,43	101322	816480	(A(B(C(D(E(F((GH)((IJ)(K(L(MN)))))))))))O	88
18	94,30,63,79,52,10,6,13,93,97,3,8,67,40,38,6,89,61,71	139845	3518984	(A(B(C(D(E(F(G(H(IJ)))))))))(((((((KL)M)N)O)P)Q	96
21	57,92,76,77,28,13,47,27,3,67,89,14,93,16,24,34,14,83,89,92,33,19	166938	2827257	(A(B(C(D(E(F(GH)))))))((((((((((IJ)K)L)M)N)O)P)Q)R)S)T)U)	94
24	79,68,62,22,98,35,62,99,21,39,91,79,81,31,11,4,87,90,90,72,57,92,36,72,59	377216	6688377	(A(B(C(D(E(F(G(H(I(J(K(L(M(NO)))))))))))))(((((((PQ)R)S)T)U)V)W)X)	94

3 Parallelization

Refer to the time required to find an optimal product sequence for a chain of matrices as the ordering time and the time required to execute the product sequence as the evaluation time [7]. Many parallel algorithms aimed at reducing the evaluation time have been studied. Sascha Hunold proposed "Multilevel Hierarchical Matrix Multiplication on Clusters" [8]. Manojkumar Krishnan proposed "Memory Efficient Parallel Matrix Multiplication Operation for Irregular Problems" [9] and Qingshan Luo gives "A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed Memory Computers" [10]. Any of the mentioned approach to reduce evaluation time can be used along with the parallel algorithm aimed at reducing the ordering time. Some of the parallel algorithms to reduce ordering time have been studied using the dynamic programming method and the convex polygon triangulation method [11] [12], however the research is scarce. Figure 3 shows the filling of m and s table diagonally for optimal matrix parenthesization problem using p_n processors, proposed by Grama and Gupta [5]. One of the major drawbacks of the approach is that it requires number of processors equal to the number of matrices, difficult to fulfil in most of the cases. Moreover, the processors do not share the uniform work load. Although Strate [13] introduced an important idea with clue that the goal should always be to minimize the idle time of all the processors, but not exploited in the mentioned approach.

3.1 Parallel Processing Algorithm

Table 3 shows the same Table 1 with the sequence of calculations. The sequential algorithm begins by solving all subproblems of length two matrices.

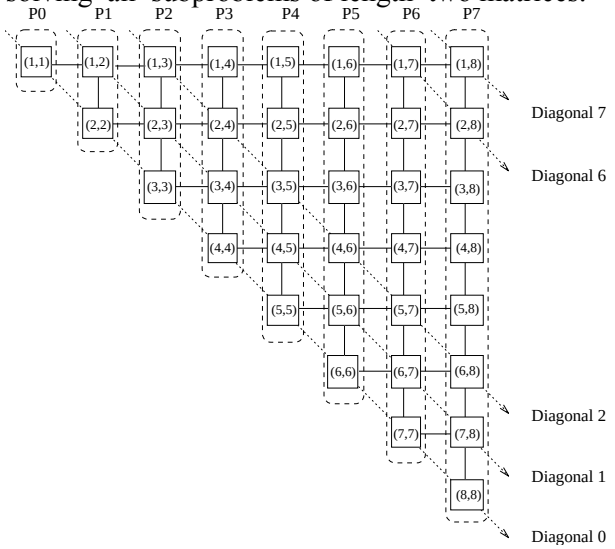

Figure 3: Using pn Processors Proposed by Grama and Gupta

Table 3 : Sequence of Calculations of Array m

	1	2	3	4	
1	0	224	180	216	
2		0	84	120	Diagonal 3
3			0	63	Diagonal 2
4				0	Diagonal 1

That is, the cost of multiplying matrices A_1A_2, A_2A_3, and A_3A_4 are determined. The cost is 224, 84

and 63 respectively. These values are entered in the above table along the first main diagonal with sequence of top to bottom and left to right. The next diagonal, entries A_1A_3 and A_2A_4 are calculated based on the previous results. The process continues until finally the A_1A_4 entry in the table is determined. This is the optimal solution. The sequential algorithm solves all subproblems on the main diagonal of the table, followed by each of the upper diagonals until a solution is determined in the upper right comer of the table. Under mentioned parallel algorithm for allocating tasks for the optimal solution to matrix parenthesization problem views the table as shown in Figure 4.

TASKING PROCESSORS (P)

```
t ← (n*n)-n)/2          {Total calculations for n matrices}
trcountbottom(m) ← 1    {Temp row count from bottom}
trcounttop(m) ← n-1     {Temporary row count from top}
p ← number of processors  {Total number of processors}
Avcalc ← t/p  {Average calculations for each processor}
for m ← 0 to p-1                        {For all processors}
    tcalc(m) ← 0        {Temporary calculations for p(m)}
    while calcp(m)<Avcalc      {calcs for each processor}
          do  calcp(m) ← tcalcp(m){Calcs for p(m)}
              tcalcp(m) ← tcalcp(m) + trcountbottom
                  rcountbottom(m) ← trcountbottom(m)
                  {Row count from bottom}
                  rcounttop(m) ← trcounttop(m)
                  {Row count from top}
                  trcountbottom(m) ++
                  trcounttop(m) - -
        End while
        return rcounttop(m),calcp(m)
End for
```

3.2 Functioning of Parallel Algorithm

p(0), p(1), p(2)……p(n) are the processors which are numbered from bottom to top. The rows are allocated numbers from top to bottom as i and also bottom to top i.e. matching to processors p(0) to p(n). Processor 1 will calculate the bottom set of rows in the table, processor 2 will calculate the next set of rows, until processor n calculates the topmost set of rows. In this arrangement processor n will finally determine the solution.

Each processor simultaneously calculates the entries in the portion of the table it is assigned. The entries in the table are processed diagonally left to right, top to bottom. This is almost same to the traditional sequential algorithm. Each time processor i, (i = 0...n), completes an entire diagonal, the entries is sent to processor i+1. Furthermore, each time processor i, begins to work on a new diagonal, it receives entries for the same column previously calculated from processor i - 1. Figure 4 illustrates these principles. In this example N=26 matrices, and n=4 processors. The numbers in the table entries represent the order in which they are calculated. Each processor has the same order. The x entries indicate calculated table entries.

The goal is to keep a processor busy, while at the same time minimizing the idle time of the other processors. Several factors must be taken into consideration [13]. Notice calculating each table entry by processor i requires more CPU time than calculating a table entry by processor i-1. This is because all previously calculated column entries from higher numbered processors must be considered. In considering all these factors the table should be partitioned in such a manner that, for a given problem, there should be proper load balance. In the above mentioned algorithm total number of calculations are ((n*n)-n)/2. Parallel processing algorithms for optimal solution to matrix parenthesization problem are mentioned below. First algorithm is used for processor p(0). Second algorithm is used for all other processors p(i). Major changes from the standard matrix parenthesization algorithm are underlined.

PARALLEL MATRIX PARENTHESIZATION(P(0))

```
n ← length[P]-1 {p is an array containing pi-1 to pj and n
                 is the number of matrices in chain }
for i ← 1 to n
   do m[i,j] ← 0{Single matrices take 0 multiplications}
for l ← 2 to n-rcounttop(1) {l is length of chain starting
                            from top of the processor p(o)}
  do for i ← rcounttop(1)+1 to n–l+1 {All possible
                            starting indices for length l}
    do j ← i + l – 1 {Ending index of chain of length l}
    m[i,j] ← INF  {Large value to start to find minimum}
    for k ← i  to j   {Try all possible splits of this chain}
          do q ← m[i,k]+m[k+1,j]+ p_{i-1}p_k p_j
          {Smaller chains are already computed}
          if q < m[i,j]        {If minimum, then store it}
                then m[i,j] ← q
                     s[i,j] ← k
      return m, s
```

PARALLEL MATRIX PARENTHESIZATION(P(i))

```
for l ← 2 to n-rcounttop(m+1) {l is length of chain
          starting from top of the processor p(m)}
 if l < (n – rcounttop(m))+2  then   ilimit = rcounttop(m)
                              else   ilimit  =  n-l+1
   for i = rcounttop(m+1)+1 to ilimit
      do j ← i + l – 1 {Ending index of chain of length l}
         m[i,j] ← INF   {Large value to start to find min}
         for k ← i  to j   {Try all possible splits of chain}
            do q ← m[i,k]+m[k+1,j]+ p_{i-1}p_k p_j
                  {Smaller chains are already computed}
             if q < m[i,j]        {If minimum, then store it}
                   then m[i,j] ← q
                        s[i,j] ← k
        return m, s
```

j

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	Processors/ Rows from Bottom
1	0	1	5	9	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**25**
2		0	2	6	10	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	94 calcs				x	**24** **P(3)**
3			0	3	7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x					x	**23**
4				0	4	8	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**22**
5					0	1	5	9	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**21**
6						0	2	6	10	x	x	x	x	x	x	x	x	x	x	x	x	78 calcs				x	**20**
7							0	3	7	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**19**
8								0	4	8	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**18**
9									0	1	6	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	**17**
10										0	2	7	x	x	x	x	x	x	x	x	x	75 calcs				x	**16**
11											0	3	8	x	x	x	x	x	x	x	x					x	**15**
12												0	4	9	x	x	x	x	x	x	x	x	x	x	x	x	**14**
13													0	5	x	x	x	x	x	x	x	x	x	x	x	x	**13**
14														0	1	13	x	x	x	x	x	x	x	x	x	x	**12**
15															0	2	14	x	x	x	x	78 calcs				x	**11**
16																0	3	15	x	x	x					x	**10**
17																	0	4	16	x	x	x	x	x	x	x	**9**
18																		0	5	x	x	x	x	x	x	x	**8**
19																			0	6	x	x	x	x	x	x	**7**
20																				0	7	x	x	x	x	x	**6**
21																					0	8	x	x	x	x	**5**
22																						0	9	x	x	x	**4**
23																							0	10	x	x	**3**
24																								0	11	x	**2**
25																									0	12	**1**
26																										0	

Figure 4: Sequences of Calculations and Partitioning of Tasks into Rows
No. of Matrices: 26, No. of Processors: 4

3.3 Implementation of Parallel Algorithm

The results for implementation of parallel algorithm for optimal solution to matrix parenthesization problem are shown in Table 4. In the Table 4, number of matrices are 20 – 100 with number of processors 1 – 10. Figure 5 includes the graph showing reduction of computations in the parallel algorithm as compared to single processor with different numbers of processors. Input includes number of matrices, number of processors and the dimensions of each matrix. The column of matrix A must be equal to the row of matrix B for all the dimensions.

3.4 Analysis of Parallel Processing Algorithm

Analyzing Table 4 with graph of Figure 5, it is obvious that there is considerable amount of time reduction proportional to the number of processors at the start. However, after some increase it is just the increase of processors without any gain. One should be mindful of that number and may call it a saturation point for that input. After that point adding more

Table 4:
Implementation of Parallel Processing Algorithm
No. of Processors: 1-4, No. of Matrices: 20-100

No. of Matrices	Total Computations with Single Processor	Maximum Computations by any Processor Using No. of Processors					
		2	3	4	6	8	10
20	190	99	85	54	54	70	70
30	435	225	159	135	110	110	135
40	780	402	284	219	185	219	150
50	1225	630	445	364	322	279	279
60	1770	909	642	495	444	392	339
70	2415	1239	875	645	524	462	462
80	3160	1620	1080	882	745	604	532
90	4005	2052	1377	1079	845	684	684
100	4950	2535	1710	1380	945	855	855

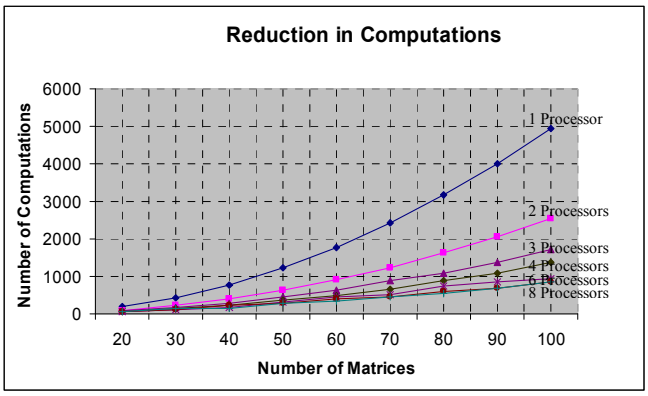

Figure 5:
Reductions of Computations in Parallel
No. of Processors:1-10, No. of Matrices:20-100

processors does not yield any more throughput but only increases the overhead and cost. Therefore, the number of processors must be used economically to get the optimal results.

For number of matrices between 26 and 104, best results are found till number of processors nine. With number of matrices 26, best results are received with number of processors seven. Therefore, one can say that algorithm is best suited for processors 2 to 10 for number of matrices till 100. Moreover, the results of parallel algorithm confirm the results of single processor algorithm.

4 Conclusion

There is substantial amount of reduction in arithmetic operations on applying matrix parenthesization algorithm proportional to the number of matrices and the sequence of dimensions. It also seems that percentage of time reduction compared to the linear left to right arithmetic operations is less, if the first dimension is smaller. Similarly, if the first dimension is larger, percentage of time reduction to the linear left to right arithmetic operations is more. Time reduction varies from 0% to 96%, proportional to the number of matrices and the sequence of dimensions. It is also learnt that on applying parallel matrix parenthesization algorithm, the amount of time reduction varies 50% and more, proportional to the number of processors at the start, however, after some increase, adding more processors does not produce any more reduction in time; rather increasing cost and effort.

References:

[1] Nikos Drakos, "Introduction to Dynamic Programming, Computer Based Learning, University of Leeds, Lecture 12, Feb 5, 1996

[2] Dr. Sanath Jayasena, "Dynamic Programming Algorithms, CS222, Lecture 11, University of Moratuwa, November 2003

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Introduction to Algorithms, The MIT Press, Cambridge, Massachusett London, England, McGraw-Hill Book Company, Boston Burr Ridge, IL Dubuque, IA Madison, WI, New York San Francisco St. Louis Montreal Toronto, 2004

[4] "Fundamental Data Structures and Techniques", Dynamic Graphics Project (dgp), Department of Computer Science, University of Toronto, CSC 270, Fall 2002

[5] Ananth Grama, Anshul Gupta, George Karypis and Vipin Kumar, "Introduction to Parallel Computing, Addison Wesley, 2003

[6] Dr. Harry Hochheiser , "The Design and Analysis of Algorithms, COSC 483, Lecture 10, Department of Computer and Information Sciences Towson University, 8000 York Road, Towson, Maryland, Fall 2006

[7] Heejo Lee, Jong Kim, Sung Je Hong, and Sunggu Lee, "Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems", 2003

[8] Sascha Hunold, Thomas Rauber and Gudula Runger, "Multilevel Hierarchical Matrix Multiplication on Clusters", ICS 04, Saint Malo, France, Jun 2004

[9] Manojkumar Krishnan, Jarek Nieplocha, "Memory Efficient Parallel Matrix Multiplication Operation for Irregular Problems", Pacific Northwest National Laboratory, Richland, ACM, CF 06, Ischia, Italy, May 2006

[10] Qingshan Luo and John B. Drake, "A Scalable Parallel Strassen's Matrix Multiplication Algorithm for Distributed Memory Computers", The University of South, ACM 0-89791-658-1, 1995

[11] P.G. Bradford, G.J. Rawlins, and G.E. Shannon, "Efficient Matrix Chain Ordering in Polylog Time", SIAM J. Computing, vol. 27, no. 2, pp.466-490, 1998

[12] A. Czumaj, "Parallel Algorithm for the Matrix Chain Product and the Optimal Triangulation Problems", Research Paper in Institute of Informatics, Warsaw University, ul Banacha, Warszawa, Poland, 1993

[13] Steve A. Strate and Roger L. Wainwright, "Parallelization of the Dynamic Programming Algorithm for the Matrix Chain Product on a Hypercube", The University of Tulsa, 1990