An Effective Solution for Matrix Parenthesization Problem through Parallelisation

Muhammad Hafeez, Muhammad Younus

Abstract--Dynamic programming can be used to solve the optimization problem of optimal matrix parenthesization problem, which is discussed in detail in the paper. 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. Foremost improvement of the parallel algorithm used is its independency on the number of matrices. Moreover, work has been uniformly distributed between processors, besides its confirmation to single processor algorithm results.

Key-Words--Matrix Parenthesization Problem Parallel Processing Algorithm

I. 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_l, A_2,…,A_n) of n matrices, where for i = 1, 2,…,n, matrix A; has dimension p_{i-l} x p_i, fully parenthesize the product A_l, A_2,…,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_l,…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[l…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].

Computer Science Department, College of Telecommunication Engineering (National University of Sciences and Technology), Hamayun Road, Rawalpindi - PAKISTAN
E-mail: chmhafeez11@yahoo.com

II. 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]+ $p_{i-1}p_kp_j$
        {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 I

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 I 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.

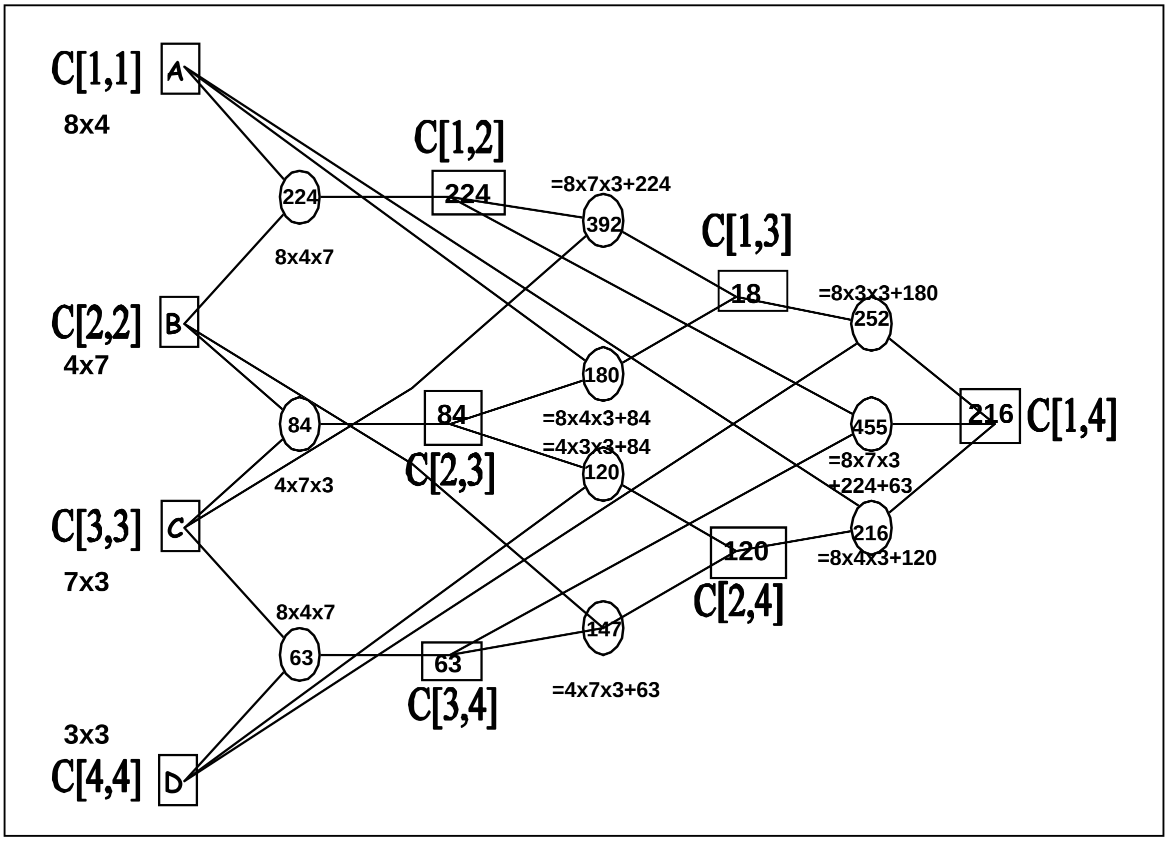

Fig 1: Optimal Matrix Parenthesization for a Chain of Four Matrices

TABLE II
IMPLEMENTATION OF MATRIX PARENTHESIZATION ALGORITHM, NO. OF MATRICES:1-10, SEQUENCE OF DIMENSIONS:1-10

No. of Matrices	Sequence of Dimensions	Optimal Arithmetic Multiplications	Left to Right Multiplications	Optimal Parenthesizations	%age Reduction of Time (d-c)/d*100
a	b	c	d	e	f
3	6,4,6,6	288	360	A(BC)	20
4	8,4,7,3,3	216	464	A((BC)D)	53
5	9,10,6,6,8,3	702	1512	A(B(C(DE)))	54
6	7,4,4,1,7,6,9	203	861	(A(BC))((DE)F)	76
7	7,4,2,8,9,6,7,8	616	1736	(AB)((((CD)E)F)G)	65
8	8,7,6,2,5,4,4,2,2	316	896	A(B(C(((DE)(FG))H)))	65
9	5,2,2,8,3,1,4,5,4,1	100	475	A(B((C(DE))(F(G(HI))))	79

TABLE III
IMPLEMENTATION OF MATRIX PARENTHESIZATION ALGORITHM, NO. OF MATRICES:1-10, SEQUENCE OF DIMENSIONS:1-25

No. of Matrices	Sequence of Dimensions	Optimal Arithmetic Multiplications	Left to Right Multiplications	Optimal Parenthesizations	%age Reduction of Time (d-c)/d*100
a	b	c	d	e	f
3	18,19,7,15	4284	4284	(AB)C	0
4	10,21,7,10,18	3970	3970	((AB)C)D	0
5	21,5,19,5,21,25	6250	17220	A(((BC)D)E)	64
6	5,3,23,21,17,1,2	934	4640	(A(B(C(DE))))F	80
7	16,23,14,24,24,14,19,17	29386	34544	((AB)(C(DE)))(FG)	15
8	21,17,6,22,20,14,16,7,5	8235	27825	A(B(C(D(E(F(GH))))))	70
9	4,21,11,23,13,18,1,2,9,8	1223	4508	(A(B(C(D(EF)))))((GH)	73

TABLE IV
IMPLEMENTATION OF MATRIX PARENTHESIZATION ALGORITHM, NO. OF MATRICES:1-24, SEQUENCE OF DIMENSIONS:1-100

No. of Matrices	Sequence of Dimensions	Optimal Arithmetic Multiplications	Left to Right Multiplications	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)R)	96
21	57,92,76,77,28,13,47,27,3,67,8 9,14,93,16,24,34,14,83,89,92,3 3,19	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, 91,79,81,31,11,4,87,90,90,72,5 7,92,36,72,59	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

A. Analysis of Implementation of Matrix Parenthesization Algorithm

The results for implementation of algorithm for optimal solution to matrix parenthesization problem are shown in Table II, III and IV. In Table II, number of matrices and sequence of dimensions varies from 1-10. In Table III, number of matrices and sequence of dimensions varies from 1-10 and 1-25 respectively, whereas in Table IV, number of matrices and sequence of dimensions varies from 1-24 and 1-100 respectively. Figures include the graph showing the time reduction in the optimal solution. In Figure 2, number of matrices and sequence of dimensions varies from 1-10. In Figure 3, number of matrices and sequence of dimensions varies from 1-10 and 1-25 respectively, whereas in Figure 4, number of matrices and sequence of dimensions varies from 1-24 and 1-100 respectively. Input includes number of matrices and then the dimensions of each matrix. The column of matrix A must be equal to the row of matrix B for all the dimensions.

Analyzing Tables and Graphs 2,3 and 4, 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. Table V shows the confirmation of the analysis. In serial 5, 6, 7 and 8, the fact is verified by getting sample values in such a manner that only the first value of the dimension has been increased. It is also observed that the percentage of time reduction has no effect if the values of sequence of dimensions is continuously increasing or decreasing; as verified in serial 9, 10, 11 and 12 of the table.

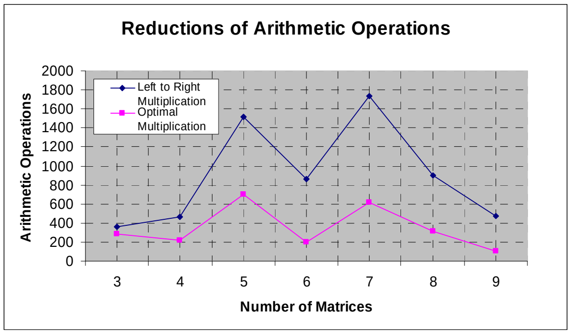

Fig 2: Reductions of Operations in Optimal Solution
No. of Matrices:1-10, Sequence of Dimensions:1-10

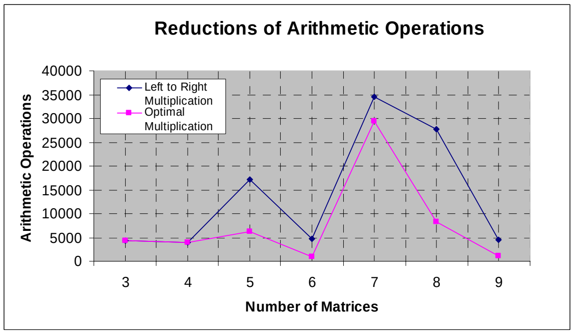

Fig 3: Reductions of Operations in Optimal Solution
No. of Matrices:1-10, Sequence of Dimensions:1-25

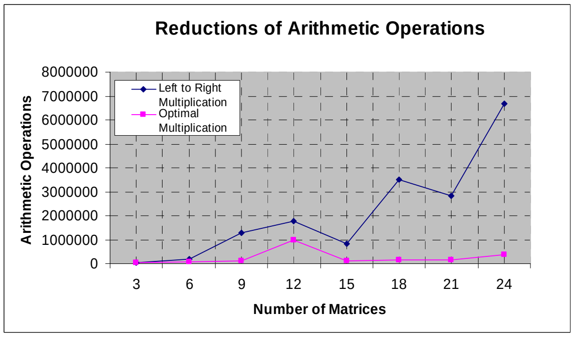

Fig 4: Reductions of Operations in Optimal Solution
No. of Matrices:1-24,Sequence of Dimensions:1-100

TABLE V

ANALYSIS OF IMPLEMENTATION OF MATRIX PARENTHESIZATION ALGORITHM

NO. OF MATRICES: 1-24, SEQUENCE OF DIMENSIONS: 1-100

Ser-ial	No. of Matrices	Sequence of Dimensions	Optimal Arithmetic Multiplica-tions	Left to Right Multiplications	Optimal Parenthesizations	%age Reduction of Time (d-c)/d*100
	a	b	c	d	e	f
1	3	9,95,21,78	32697	32697	(AB)C	0
2	3	18,19,7,15	4284	4284	(AB)C	0
3	4	10,21,7,10,18	3970	3970	((AB)C)D	0
4	3	9,95,21,78	32697	32697	(AB)C	0
5	6	7,34,8,32,30,30,40	25116	25116	((((AB)C)D)E)F	0

6	6	50,34,8,32,30,30,40	54080	179400	(AB)(((CD)E)F)	70
7	6	75,34,8,32,30,30,40	68880	269100	(AB)(((CD)E)F)	74
8	6	100,34,8,32,30,30,40	83680	358800	(AB)(((CD)E)F)	77
9	6	50,15,16,20,25,30,40	71550	150500	A((((BC)D)E)F)	52
10	6	50,45,40,35,30,25,20	145000	275000	A(B(C(D(EF))))	47
11	6	7,45,40,35,30,25,20	38500	38500	((((AB)C)D)E)F	0
12	6	7,15,16,20,25,30,40	21070	21070	((((AB)C)D)E)F	0

III. Parallelization of Optimal Solution to Matrix Parenthesization Problem

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 5 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 and provided a clue that the goal should always be to minimize the idle time of all the processors, but not exploited in the mentioned approach.

A. Parallel Processing Algorithm

Table VI shows the same Table I with the sequence of calculations. The sequential algorithm begins by solving all subproblems of length two matrices. 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 6.

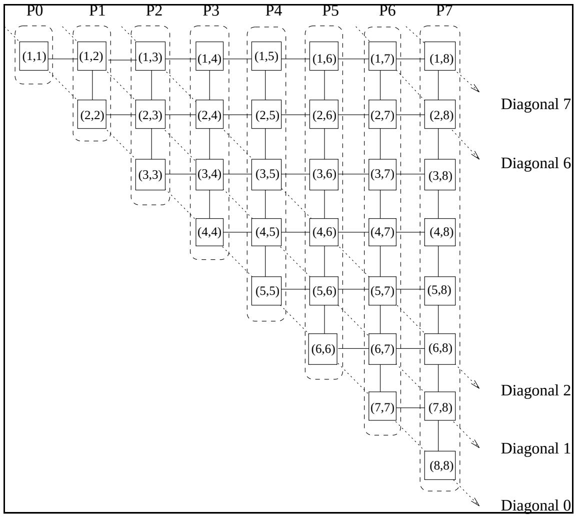

Fig 5: Using p_n Processors Proposed by Grama and Gupta

TABLE VI

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

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
```

B. *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 6 illustrated 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.

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	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	x	**20**	**P(2)**
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	x	x	x	x	**15**	**P(1)**
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	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**	**P(0)**
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		

Fig 6: Sequences of Calculations and Partitioning of Tasks into Rows
No. of Matrices: 26, No. of Processors: 4

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. Figure 8 reveals the result with the application of the mentioned algorithms with Number of Matrices: 26, Number of Processors: 3 and Sequences of Dimensions: 9,8,7,6,5,4,3,2,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,2,3,4.

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]+ $p_{i-1}p_kp_j$
                 {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]+ $p_{i-1}p_kp_j$
          {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
```

C. *Implementation of Parallel Algorithm*

The results for implementation of parallel algorithm for optimal solution to matrix parenthesization problem are shown in Table VII. In the Table VII, number of matrices are 20 – 100 with number of processors 1 – 10. Figure 7 includes the graph showing the 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.

TABLE VII

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

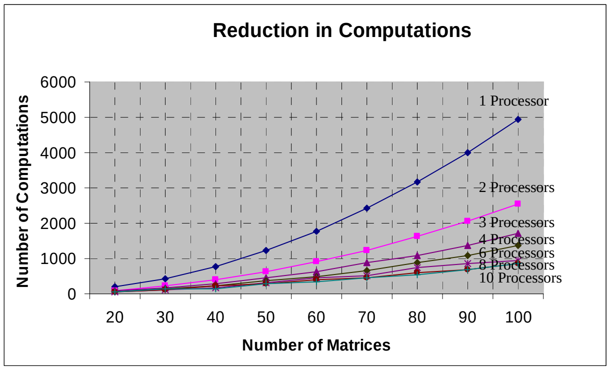

Fig 7: Reductions of Computations in Parallel
No. of Processors: 1-10, No. of Matrices: 20-100

D. *Analysis of Parallel Processing Algorithm*

Analyzing Table VII with graph of Figure 7, 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 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.

IV. 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 Unit, 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*, ieeexplore isnumber = 26889, 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

	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 Botton
1	0	504	768	850	800	660	464	476	530	572	630	708	810	940	1102	1264	1390	1484	1550	1592	1614	1620	1586	1594	1648	1690	25
2		0	336	490	512	444	320	332	380	420	476	552	652	780	940	1102	1230	1326	1394	1438	1462	1470	1442	1450	1498	1538	24 P(2)
3			0	210	288	276	208	220	262	300	354	428	526	652	810	972	1102	1200	1270	1316	1342	1352	1330	1338	1380	1418	23 {115 calcs}
4				0	120	150	124	136	172	208	260	332	428	552	708	870	1002	1102	1174	1222	1250	1262	1246	1254	1290	1326	22
5					0	60	64	76	106	140	190	260	354	476	630	792	926	1028	1102	1152	1182	1196	1186	1194	1224	1258	21
6						0	24	36	60	92	140	208	300	420	572	734	870	974	1050	1102	1134	1150	1146	1154	1178	1210	20
7							0	12	30	60	106	172	262	380	530	692	830	936	1014	1068	1102	1120	1122	1130	1148	1178	19
8								0	12	36	76	136	220	332	476	638	782	894	978	1038	1078	1102	1110	1118	1130	1154	18 P(1)
9									0	24	64	124	208	320	464	626	770	882	966	1026	1066	1090	1102	1110	1122	1146	17 {105 calcs}
10										0	60	150	276	444	660	903	1119	1287	1413	1503	1563	1599	1090	1102	1120	1150	16
11											0	120	288	512	800	1124	1412	1636	1804	1924	2004	1563	1066	1078	1102	1134	15
12												0	210	490	850	1255	1615	1895	2105	2255	1924	1503	1026	1038	1068	1102	14
13													0	336	768	1254	1686	2022	2274	2105	1804	1413	966	978	1014	1050	13
14														0	504	1071	1575	1967	2022	1895	1636	1287	882	894	936	974	12
15															0	648	1224	1575	1686	1615	1412	1119	770	782	830	870	11
16																0	648	1071	1254	1255	1124	903	626	638	692	734	10
17																	0	504	768	850	800	660	464	476	530	572	9 P(0)
18																		0	336	490	512	444	320	332	380	420	8 {105 calcs}
19																			0	210	288	276	208	220	262	300	7
20																				0	120	150	124	136	172	208	6
21																					0	60	64	76	106	140	5
22																						0	24	36	60	92	4
23																							0	12	30	60	3
24																								0	12	36	2
25																									0	24	1
26																										0	

Fig 8: Partitioning of Tasks into Rows, No. of Processors: 3, No. of Matrices: 26,

Sequences of Dimensions: 9,8,7,6,5,4,3,2,2,3,4,5,6,7,8,9,9,8,7,6,5,4,3,2,2,3,4