Efficient Matrix Multiplication

Karina Narotam Elijah Smith Nathanael Gaulke Garrett Spears Simon Weizman

***Abstract*—The matrix chain multiplication problem is a well-studied optimization problem that finds a solution for the most efficient way to multiply a given set of matrices. Matrix multiplication is associative, so we are free to parenthesize the matrices in the most optimal way possible, which reduces the total number of computations. We investigate the effectiveness of converting existing optimization approaches to parallel implementations that will carry out this task. We also investigate parallel approaches for the actual multiplication of two matrices. We evaluate these approaches and report any applicable speedup to traditional sequential implementations.**

I. Introduction

Matrix multiplication is a fundamental component of many sciences and technologies today such as engineering, statistics, linear algebra, and many more. In high-pressure or computationally-intense environments, it is critical that chains matrices can be computed efficiently using the full capabilities of our constantly-evolving technology. One such evolution that lends itself well to computationally-intensive operations is the growing availability of multi-core processors. In order to speed up the overall process of multiplying out chains of matrices, we can split our computations using parallelization to provide a simple and efficient matrix chain multiplication library. The two main problems we seek to optimize are finding the optimal ordering for the matrix multiplication operations and performing large multiplications efficiently. First, to find the optimal ordering of multiplication, we propose a parallel approach to the common $O(n^3)$ dynamic programming approach. This approach works bottom-up to compute the minimum number of operations for all possible subchains of every length on multiple threads. Second, we utilize the optimal ordering produced by either implementation to perform the full chain computation using a parallel approach to the common $O(n^3)$ direct multiplication algorithm. Our findings and comparison to the sequential algorithms are reported.

II. Background

A. Matrix Chain

A matrix chain is a series of 2 or more matrices which can be multiplied sequentially to produce one resulting matrix. A valid matrix chain demands that all neighboring matrices in the chain have matching inner dimensions - that is, the number of columns in the left matrix of the pair is equal to the number of rows in the right matrix of the pair.

Ex: Of the following matrix chains (where each matrix is represented by it's two dimensions $R \times C$), only the first is valid and can be evaluated.

$$(10 \times 20) \times (20 \times 40) \times (40 \times 30) \tag{1}$$

$$(15 \times 60) \times (35 \times 20) \times (20 \times 30) \tag{2}$$

The dimensions of every matrix in the chain can be aggregated in a list *dims* of size $N+1$, where N is the number of matrices in the chain. The 0^{th} element in this list is the number of rows in the 0^{th} matrix in the chain, and the next i elements are the number of columns in the $i-1^{th}$ matrix (for $i = 1 \rightarrow N$). In this list, the inner elements $(1 \rightarrow N-1)$ specify the shared inner dimension of each neighboring matrix pair in the chain.

B. Matrix Chain Ordering Problem

The well-established matrix chain optimization problem seeks to find the most optimal parenthesization of a chain of matrices such that the total number of operations performed to multiply the chain fully is the minimum of all possible, valid parenthesizations. Certain parenthesizations produce an exponentially higher number of total optimizations than others to multiply out, which can severely hamper the performance of applications that require intense matrix computation. An example of this phenomenon can be seen with the following example.

Ex: If A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix, then the follow equations show the two possible parenthesizations of ABC and the number of operations required to multiply out each:

$$(AB)C : (10 \times 30 \times 5) + (10 \times 5 \times 60) = 4500 \tag{3}$$

$$A(BC) : (30 \times 5 \times 60) + (10 \times 30 \times 60) = 27000 \tag{4}$$

In this example, choosing the second, inefficient parenthesization leads to a 500% increase in the number of operations performed, which may be unacceptable in high-performance programs or those that require many chain multiplications.

The naive or brute force algorithm for finding the most optimal parenthesization is to try all and return the best one. This is a slow process since if there are n matrices, there are $n-1$ places the chain of matrices can be parenthesized. A split after the kth item creates two subchains which can further be parenthesized. This is related to Catalan numbers, which also describe the number of different binary trees on n nodes. It is a fast growing, exponential algorithm that is not feasible for any programming applications with n as an arbitrary value.

This problem has a well-known dynamic programming (DP) solution that can determine the number of operations required for the most optimal parenthesization in $O(n^3)$ time, where n is the number of matrices in the full chain. We are able to store the resultant number of operations required for every possible subchain to prevent redundant calculations and achieve a polynomial-time solution.

In this solution, we construct two 2-D arrays of integers, with each dimension of size $N+1$. The first, which we will abbreviate as *dp*, holds at each index the minimum number of operations required to multiply out the subchain of matrices from $[i, j]$ for $i = 1 \rightarrow (N-1), j = (i+1) \rightarrow N$. This approach works bottom-up by computing all such values for subchains of length $l = 2 \rightarrow N$. For each value of l, we calculate the cost of multiplying the subchain if we were to partition it into two smaller chains at each possible partition $k = i \rightarrow (j-1)$ and call it q as below:

(5)
$$q = dp[i][k] + dp[k+1][j] + dims[i-1] \times dims[k] \times dims[j]$$

In this equation, *dims* is an $N+1$ length list containing the inner dimensions of each adjacent matrix pair, plus the number of rows of the first matrix at the front of the list and the number of columns of the last matrix at the end. $dp[i][j]$ is set to be the minimum of all q for each (i, j) pair.

Additionally, for each (i, j) pair, we store the value of k corresponding to the minimum q in $s[i][j]$. After our algorithm finishes, we can recursively use the matrix s to break down our chain all the way into its parenthesization and multiply using the optimal ordering bottom-up (similar to how a merge-sort operates).

This approach in total runs in $O(n^3)$ time using three looping variables l, i, k and the minimum number of computations for the entire chain can be found at $dp[1][N]$. For longer chain lengths N, this algorithm may run somewhat slowly, which opens opportunities to parallelize this computation to maximize performance.

C. Previous Work on Ordering Problem

Our project proposes enhancements to the DP solution to this problem by distributing computation between multiple threads. While the $O(n^3)$ algorithm is often reasonable for standard applications or for when the matrices in the chain are few in number but very large in individual dimensions, we aim to create a parallel approach for comparison on very long chains.

Several non-parallel approaches have been designed to solve the matrix chain ordering problem with a run-time less than $O(n^3)$ at the cost of higher complexity. One such algorithm published by T. C. Hu and M.T. Shing that runs in $O(n * log(n))$ time reduces this problem into the problem of triangulation of a regular polygon [1]. A simplified version of this algorithm published by Xiaodong Wang, Daxin Zhu, and Jun Tian uses $O(n * log(m))$ time, where n is still the number of matrices in the chain and m is the number of local minimums in the sequence of dimensions for each matrix in the chain [2]. Finally, an approximation algorithm independently created by both Francis Chin [3] as well as Hu and Shen [4] can produce a parenthesization at most 15.47% worse than the optimal choice in $O(n)$ time. Each of these algorithms improve on the DP solution to this problem, but are high-enough complexity to provide a great challenge in finding efficient additions for parallelization. Our solutions aim to replace speedups produced by these efficient algorithms with speedups produced by parallelization.

III. Methodologies

A. Chain Multiplication DP Optimization

Our proposed implementation for a concurrent DP algorithm splits the computation for total required operations for each possible subchain length among multiple threads.

Let l be the length of a given matrix chain, meaning the number of matrices in the chain. For any given chain of length l, this chain can be divided at any point between two matrices into subchains of length l_0 and l_1, where $1 \leq l_0, l_1 < n$, that can be themselves multiplied out separately. As a result, the DP solution to this problem requires that the minimum number of operations to multiply that subchain be computed sequentially for increasing values of l to be available when computing chains of greater lengths. Multiplying a chain of length $l = 1$ is a 0-cost operation since the matrix does not multiply with any other matrices. We initialize all spaces in our DP 2-D array to be 0 initially, so we can ignore the $l = 1$ case and start immediately with $l = 2$.

In our implementation, a fixed number of threads will be generated at the start and given references to storage arrays *dp* and *s*. At execution start and after finishing with its previous value, each thread will poll for the next-largest, unprocessed value of l from a thread-safe queue, waiting as necessary. The queue will contain values $[2, N]$ at the start of the algorithm and will distribute the next lowest value from the front of the queue to each requesting thread.

Each thread will compute all minimums for each possible subchain of the length given to it from the queue. This will require retrieving the minimum from subchains of smaller lengths that must already be completed. If a thread currently working on this exact, smaller subchain has not yet computed the result, the thread waiting will spin until this value has been calculated. Because we start computations with $l = 2$ and because values in *dp* are only filled for indexes (i, j) where $i < j$, each thread will only wait on subproblems where either of these conditions are true (we only need to choose and wait for one). While the value at such index is zero, the thread will continue spinning to avoid the overhead of sleeping/waiting for a lock until the value changes as a result of the other thread having calculated the value. Threads will stop when the queue is empty and the entire computation has been completed.

We expect to only use two threads for this approach and will test the efficiency of this algorithm on a variety of matrix chain lengths. Because the size of the individual matrices will produce very large computations on long chains, we will convert the dp array to use BigIntegers if we discover overflow.

B. Chain Computation

When multiplying a chain of matrices together, we first considered splitting up the work of multiplying each pair of matrices across different threads. However, this would not be ideal for our implementation of matrix chain multiplication. Explained more in-depth above, before multiplying a chain of matrices, we are computing the most efficient parenthesization of a matrix chain to reduce total computation. This ordering of multiplying a chain of matrices can exponentially reduce total computation. However, if we were to concurrently multiply separate pairs of matrices in the chain, we would be unable to always follow the efficient parenthesization of matrices that would be computed prior. This is due to the fact that the most efficient parenthesization of a matrix chain will often require that a matrix must wait on an intermediate matrix to be multiplied to. For example, if $A(B(CD)$ was the most efficient parenthesization of four matrices, then we can clearly not multiply two pairs of matrices at the same time since B must wait on CD to finish and A must wait on $B(CD)$ to finish. Since it is in our best interest to follow this efficient ordering of multiplying matrices, we focused on how we can speed up the multiplication of two matrices with parallelization instead of splitting up the chain of matrix multiplication concurrently.

After deciding the ordering of the matrices to be multiplied, the next step was deciding the best method to concurrently multiply matrices together and how to efficiently do so. When discussing matrix multiplication there were two main methodologies to consider: a naive approach and Strassen's algorithm. Given any two matrices A and B, the naive approach of matrix multiplication is multiplying row A_i by column B_j to create the $(i,j)^{th}$ position of Matrix C. This approach takes $O(n^3)$ time. Strassen's algorithm creates submatrices from the matrices A and B in order to create Matrix C. This approach uses recursion and uses less multiplication than the naive approach. As a result, the algorithm performs in roughly $O(n^2.807)$ time. While Strassen's algorithm is faster overall, its speed decreases for very small matrices or very large matrices. Furthermore, the algorithm's recursive nature makes parallelizing the algorithm nearly impossible. Thus, the team decided to stick with the naive approach for solving matrix multiplication.

The next step was deciding how to parallelize the naive approach for a chain of matrices that need to be multiplied.

One approach was to first create the intermediary matrix that would hold the results of Matrix A and Matrix B. In this approach, one thread is created and assigned to every column of matrix B. Each thread would multiply the first row of A with their respective column of B. After each thread multiplies its column by the first row, the first row of the intermediary matrix C will have been created. The threads can then move on and begin multiplying their column by the second row of A (if it exists). Meanwhile, we can begin multiplying matrix C by the next matrix to be multiplied, matrix D. This is due to the fact that we need only one row to begin multiplication for the next set of matrices. We follow the same process as the beginning of creating a matrix to hold the results of C and D. Again, we create as many threads as there are columns in D. Each thread would multiply the first row of C with their respective column of D. After each thread multiplies its column by the first row, the first row of the next intermediary matrix E will have been created. By then, matrices A and B would have created the next row of C and so the threads can begin multiplying their threads by the second row of C (if it exists). Meanwhile E can be multiplied by the next matrix and so on until the last matrices are multiplied together. This approach creates matrices as soon as it is possible to begin multiplying even one row by the next matrix's columns. This approaches issue is starvation. Threads might be available to do work on the next set of matrices. However, if the next row has not been created, then they simply will be waiting. While multiplication and addition should occur in constant time, there is a chance that some calculation will be performed more quickly for one value than another. Thus, again more waiting. Ultimately, while this approach is viable, the next approach seemed the better option.

After considering that the first approach we had for matrix multiplication would not be nearly as efficient as we had previously thought, we decided to further explore the second idea we had for naive matrix multiplication. Instead of assigning one thread to do the computation to fill up a single row of the resulting matrix, our second idea explored the process of having each thread simultaneously compute a single value in the resulting product matrix. When multiplying two matrices $A \times B$ to produce a result matrix C, each row i in matrix A needs to be multiplied by each column j in matrix B producing the value for position (i,j) in C. The fundamental idea around the approach we came up with is that each thread will do the computation of multiplying a row in matrix A and a column in matrix B to produce a single value that will populate a unique cell in matrix C.

For example, imagine that we are multiplying two matrices and the product matrix is of size 3×2, then there would need to be 6 total threads that each multiply a single row and column from the two input matrices. The result of each thread's row, column multiplication can then be written to a corresponding cell in the product matrix. Once all 6 threads finish executing, then all 6 cells of the product matrix will be filled up, meaning that the two input matrices have been successfully multiplied together.

Before fully considering this approach, we wanted to ensure that this concurrent approach of multiplying two matrices would not lead to any contention or race conditions. Since each thread is only reading from the two input matrices without modifying any of the values in the input matrices, there should never be any issues where two threads read different values from the same cell in an input matrix. Furthermore, this means that threads will never have to wait on anything in the input matrices since none of the input matrices' values will ever be modified during the time that these two matrices are being multiplied.

Additionally, there should never be any issues where multiple threads are writing a value into the product matrix at the same time. This is due to the fact that each thread will be in charge of multiplying a unique row and column combination that will result in filling up a unique cell in the product matrix. Each cell in a matrix is represented by a location in memory that stores the value for that cell. Since each thread is responsible for writing to a unique cell in the product matrix, they should never write to the same location in memory.

IV. Experimentation

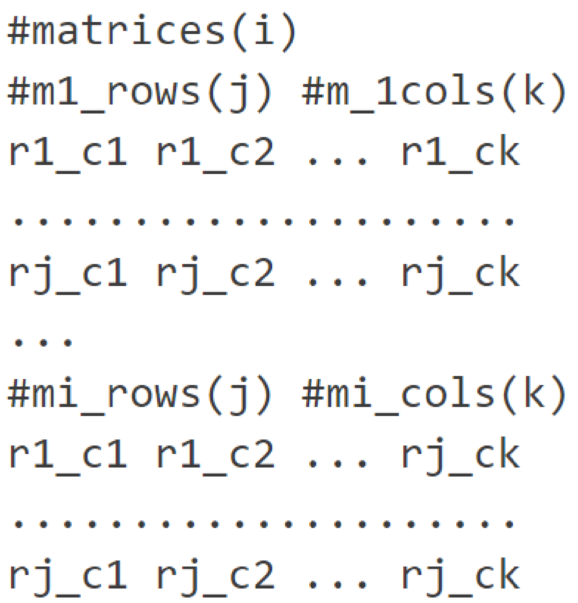
A. Testing Tools

In order to efficiently test any algorithms we developed, we created a MatrixGenerator class that outputs a chain of matrices to a single text file based on several parameters:

- The maximum number of columns in each matrix (min 1)
- The maximum number of rows in each matrix (min 1)
- The maximum value of each item in each matrix
- The minimum number of matrices in the chain (min 2)
- The maximum number of matrices in the chain

This generator allowed us to quickly create chains of any length that fit the needs of the specific components we were testing. This generator is invoked via a bash script for easy compilation, execution, and cleanup.

Matrix chains are represented in file inputs with the following format:



```
#matrices(i)
#m1_rows(j) #m_1cols(k)
r1_c1 r1_c2 ... r1_ck
.....................
rj_c1 rj_c2 ... rj_ck
...
#mi_rows(j) #mi_cols(k)
r1_c1 r1_c2 ... rj_ck
.....................
rj_c1 rj_c2 ... rj_ck
```

B. Chain Optimization

To evaluate the parallel approach to finding the optimal multiplication ordering, we created a separate ParallelOptimizationChain class that extends our MatrixChain class and exposes the same methods. This class overrides the private methods that are used to compute the optimal ordering so that the user of the library may swap out the two approaches at will. To test these two approaches, each test case would be loaded into both a MatrixChain and a ParallelOptimizationChain class and each would be timed in milliseconds for how long it takes to complete only the optimization step. We output the length of each side-by-side, as well as the total number of operations determined by each to verify correctness, before moving on to the next test case.

A separate folder of test cases was created for tests using generator parameters specific suited for this problem (and less so for testing direct multiplication strategies). Only for large values of N (REALLY long chains) do we expect any benefit from this algorithm, so we generated test cases of varying lengths with matrices of smaller dimensions (≤ 10) and values of either 0 or 1, since the specific values in each matrix are irrelevant for this problem.

Our testing plan for this algorithm was to create matrix chains of lengths $100, 250, 500, 1000$, and 5000. We expected to see longer runtimes for up to the $N = 500$ case due to the overhead of managing multiple threads outweighing the benefits.

C. Matrix Multiplication

The first component of testing is to determine the optimal thread count for dividing the labor of multiplying two matrices. While the exact values will vary from computer architecture to architecture, our goal was to choose a value that results in near-optimal performance on both a lower-end and a higher-end device. We chose to evaluate the parallel algorithm on a single, large test case of $10 - 20$ matrices with 1000×1000 maximum dimensions. This algorithm will be run on multiple occasions with increasing thread counts (until the execution duration flatlines) and the runtime will be recorded for each.

The second component of testing is to test evaluate the runtimes of three different strategies for computing the result of the entire chain of matrices:

1) Brute Force
    a) This approach multiplies matrices from left to right sequentially and includes no parallel strategies or constructions. This is expected to be very slow and is used as a baseline.
2) Optimized ordering with 1 thread
    a) This approach uses the optimal ordering produced by the DP algorithm but computes the product of each multiplication using only one thread. This is used to determine if adding additional threads to each matrix product computation has meaningful effect.
3) Optimized ordering with N threads
    a) This approach also uses the optimal ordering but splits each product computation among N threads, where N is the optimal number of threads determined by the previous testing step.

Additionally, the resulting matrix of each strategy on each test case is compared to the other strategies for correctness.

We tested for two independent variables, matrix chain size and individual matrix dimensions. Separate tests were run to determine the effect of using multiple threads for computation on chains with differing values for each variable (with the non-tested variable remaining constant).

For testing against matrix dimensions, each test case consisted of a matrix chain of length 8. We tested three different trials where each matrix was 500×500, 1000×1000, and 2000×2000 respectively. For testing against matrix chain length, all matrices in the chain were fixed to maximum dimensions of 100×100 with different chain lengths of $50, 100,$ and 1000.

V. Results

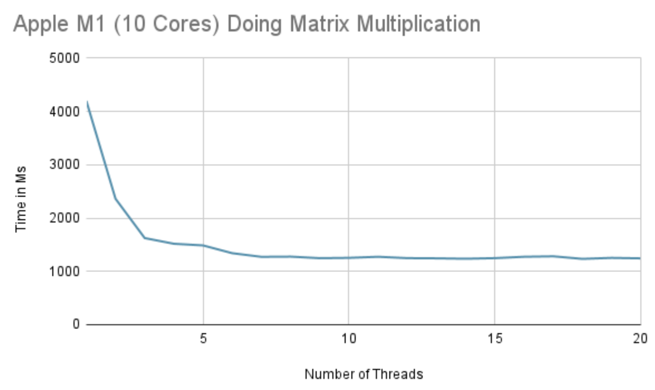
A. Chain Optimization

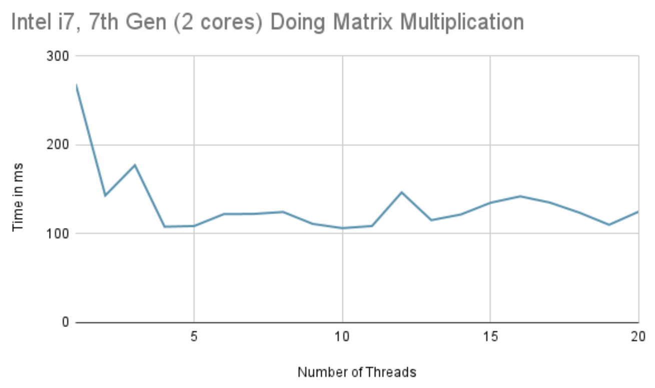
Our first test case of $N = 100$ completed using the ParallelOptimizationChain in 9ms compared to the MatrixChain completing in less than 1ms, which indicated to us that we were correct in assuming this approach would outweigh the benefits for smaller values of N. Lengths of $N = 250$ and $N = 500$ similarly showed a $10\times$ larger runtime when using the ParallelOptimizationChain than with the standard MatrixChain. We decided that evaluating test cases for N in the thousands or ten thousands (where speed may finally be a concern) is unrealistic due to the common usages of matrix multiplication in the real world.

It is far more likely that a matrix chain would be of a reasonable length but composed of very large individual matrices (in this case, our time would be better spent optimizing the actual matrix multiplication process itself, which became the focus) or that matrix multiplication may be performed in high volumes but of short chains. Even with small individual values in each index of the matrix, multiplying thousands of matrices together in one chain produces impossibly-large values in the result matrix. In either case, the parallel approach to determining the optimal multiplication ordering has a negative impact on the process and using the standard DP approach is more efficient.

B. Matrix Multiplication

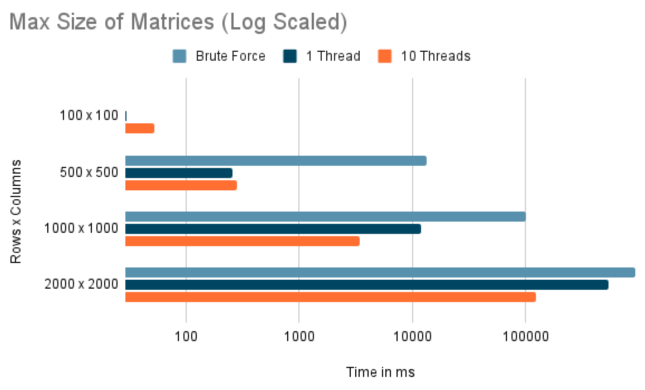
To determine the optimal number of threads for the parallel approach to multiplying two matrices, we tested the large test case on two different processors: 1) An Apple M1 (10 cores), and 2) Intel i7, 7th Gen (2 cores). The resulting runtimes with differing numbers of threads involved in each multiplication can be seen below:

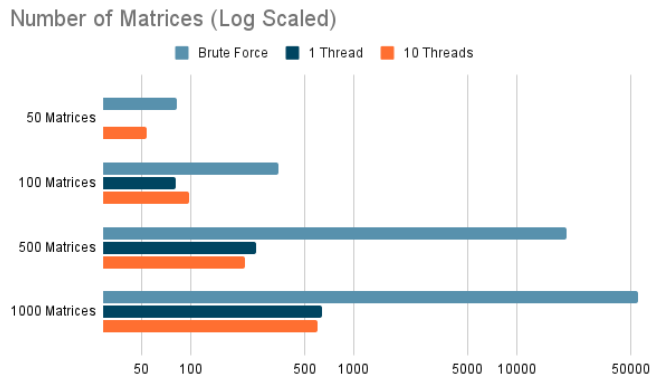




These results show that using 10 threads produced a near-minimum runtime on both the low-end and high-end processor and is an ideal static choice for our algorithm.

With the ideal thread count chosen, we tested performance for both differing matrix dimensions and differing matrix chain lengths according to the testing plan described in the evaluation section. The results of each test, using a logarithmic scale to better display the differences between each test case, are shown below:





As expected, the brute force approach to matrix chain multiplication is far and away the worst performer, proving that optimizing the ordering of multiplication operations is an invaluable component of efficient chain multiplication. In both tests, smaller matrix dimensions or chain lengths show a slight performance hit for using multiple threads. However, for larger matrices, using multiple threads can greatly increase performance of multiplying the entire chain. Longer chains also show a slight but less significant performance increase for using multiple threads.

C. Conclusion

This project demonstrates that distributing the operations involved in multiplying two matrices together can speed up the performance of multiplying out an entire matrix chain. While using 10 threads to distribute this work showed significant runtime improvements for chains comprised of matrices with large dimensions, it also provided benefit for chains of increasing length. Additionally, we have created a simple, user-friendly Java library for Matrix multiplication that can be used to take advantage of this approach in performance-demanding environments. Further exploration is possible to determine thresholds for when increasing or decreasing thread count is valuable to adjust to different qualities of the matrix chain (including length and matrix dimensions).

References

[1] Hu, TC; Shing, MT (1981). Computation of Matrix Chain Products, Part I, Part II (PDF) (Technical report). Stanford University, Department of Computer Science. Part II, page 3. STAN-CS-TR-81-875.

[2] Wang, Xiaodong; Zhu, Daxin; Tian, Jun (April 2013). "Efficient computation of matrix chain". 2013 8th International Conference on Computer Science Education: 703–707. doi:10.1109/ICCSE.2013.6553999.

[3] Chin, Francis Y. (July 1978). "An O(n) algorithm for determining a near-optimal computation order of matrix chain products". Communications of the ACM. 21 (7): 544–549. doi:10.1145/359545.359556.

[4] Hu, T.C; Shing, M.T (June 1981). "An O(n) algorithm to find a near-optimum partition of a convex polygon". Journal of Algorithms. 2 (2): 122–138. doi:10.1016/0196-6774(81)90014-6.