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 the parallelization 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 long chains of 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 implement a parallel approach to the common $O(n^3)$ dynamic programming approach and evaluate its effectiveness compared to its counterpart. 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 Strassen's algorithm, which runs in $O(n^{2.807})$ time. Our findings and comparison to the sequential algorithms are reported.

II. Background

A. 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:

$$(1) \quad (AB)C : (10 \times 30 \times 5) + (10 \times 5 \times 60) = 4500$$

$$(2) \quad A(BC) : (30 \times 5 \times 60) + (10 \times 30 \times 60) = 27000$$

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.

B. Previous Work on Ordering Problem

Our project proposes enhancements to both the brute-force and DP solutions 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 but very large in individual dimensions, we aim to create two possible implementations of parallelization in this process 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 the minimum number of operations to multiply that subchain be computed sequentially for increasing values of l. In an $l = 1$ chain, we consider this to be a 0-cost operation since the matrix does not multiply against itself. 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 to assist with filling up the storage array. Each thread will spin until it is able to receive the next largest unprocessed value of l from a queue. 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 thread that finishes working on a previous value.

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 the smaller subchain has not yet computed this result for the exact required subchain, the thread waiting will spin until this value has been calculated. Threads will stop when the queue is empty and the entire computation has been completed.

We will experiment with the number of threads sharing the computation at once as well as the length of the matrix chain and the size of the inner dimensions of matrices in the chain to identify whether parallelization is helpful or detrimental in a significant number of cases.

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.

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.