Parallelization of the Needleman-Wunsch Algorithm

Klasing, Blake
University of Central Florida
Orlando, FL
BKlasing@knights.ucf.edu

Gramajo, Stacy
University of Central Florida
Orlando, FL
SGramajo@knights.ucf.edu

1. Abstract

In Bioinformatics, one of the well-known problem is understanding evolutionary relationships through sequence alignments that can be found in a number set of data: protein, DNA, RNA, or genes. A fundamental method used for global sequence alignment is Needleman-Wunsch algorithm. It solves the problem of finding the optimal alignment among two sequences, based on a simple scoring system. We propose to examine this algorithm and implement a parallelized program that can decrease the execution time for analyzing two sequences without sacrificing correctness and time complexity.

2. Introduction

The Needleman-Wunsch(NW) algorithm is global sequence alignment method that exemplifies the relationship between sequences using a scoring system. Using a dynamic programming approach, there are two parts to the algorithm in order to retrieve the optimal score: creating a matrix and traceback. The traceback method can be accomplished in linear time; however, the creation of the scoring matrix poses a more complex problem that we will be focusing on.

Overall, the NW algorithm achieves a $O(n \cdot m)$ time complexity and $O(n \cdot m)$ space complexity, where n and m are the lengths of the input sequences to be aligned. Although this is not an issue if the sequences are small, as they increase, the size of the scoring matrix as well as computing it grows in a quadratic fashion. For huge datasets that are the norm within the computational biology field, this poses an issue. For this reason, speeding up this algorithm can have great affects within the bioinformatics community, if not at least provide some insight into how to parallelize such an algorithm.

We propose to parallelize the method that computes the scoring matrix within the NW algorithm in order to improve the overall execution time. Our main goal is to achieve reasonable speedup versus the original NW algorithm by utilizing current consumer grade multicore processing architectures.

3. Related Work

The Needleman-Wunsch algorithm is widely known for solving this fundamental biology problem of global RNA sequence alignment. An algorithm proposed by Temple F. Smith and Michael S. Waterman is also very well known for solving a very similar problem, local RNA sequence alignment [3]. Both of these algorithms solve the problem of optimal RNA alignment by dynamic programming, reaching the same time complexity. There are many other implementations and concepts of solving the problem of non-structured optimal RNA alignment, but we will focus on the NW dynamic programming solution.

Some parallelized implementations of NW take advantage of the hardware being used to not only speed up the original non-parallelized algorithm, but also push this hardware to its limits in order to squeeze as much computation as possible out of it with a parallelized algorithm. A group of researchers at the University of Delaware create a highly parallelized implementation based on fine grain multithreaded execution and architecture model [2].

The majority of research around parallelizing the NW algorithm focuses on speeding up the calculation of the scoring matrix, which is where most of the computation takes place. However, some researchers focus on both the method of creating the scoring matrix, as well as the backtracking method [1]. Researchers at the University of Puerto Rico at Mayaguez bring a more complete approach of parallelizing the NW algorithm by offering a parallel implementation of the backtrack-

ing method, in addition to the scoring matrix method. This group of researchers take advantage of properties of symmetry between the matrix D and D', where D is a matrix based on the original sequences being aligned, and D' is a matrix based on the same sequences in reverse order.

Other approaches include dividing the input sequences into several smaller sequences, calculating the matrices, backtracking to find the optimal score, and combining the results from each set of sequence alignments. This method is based on finding local optimal sequence alignment. This implies that it is not guaranteed to find the optimal global sequence alignment score. However, it may provide insight into possible solutions to parallelizing this global optima algorithm [4].

4. Background

4.1. Needleman-Wunsch

To compute the NW, the algorithm needs two sequences as the input. Assuming in this case that sequence 1 is m and sequence 2 is n. Using the sequences, a table is created where the number of rows equal to length of m+1 and the number of columns equal to the length of n+1. After the creation of the matrix S, the first row and column in precomputed. Then the matrix takes each of the remaining cells in the matrix and computes the maximum of three equations:

$$S(i,j) = max \begin{cases} S(i-1,j-1) + c(i,j) \\ S(i-1,j) + c(i,-) \\ S(i,j-1) + c(-,j) \end{cases} \quad (1)$$

Once S has been filled, the final score is taken by the last item that was inputted. In order to print out the string containing gaps, mismatches, or matches, NW uses S to trace back starting at S [m+1, n+1].

The scoring system that is used to determine the overall score consists of three types of variables used: Gap, Mismatch, and Match. Gap is represented as the - character symbol that will appear in either/both of the sequences as the output solution. Match is the number added in the score if the characters of index i equal each other. Mismatch is usually a negative value that decrease the final score.

4.2. Multiprocessor Programming Concepts

In both the lock-based and lock-free implementation, it uses an unbounded pool as one of its architecture design. A bounded or unbounded item determines if the structure is limited to a specific capacity. If the object is unbounded, then it is not confined to a size; whereas, bounded is visa-versa. A pool is an object that contains methods where producer threads is able to store items and consumer threads are able to retrieve items to work on.

The lock-based method will contain a few mutexes. Mutexes are locks used to ensure mutual exclusion on critical sections. The lock-free does not use synchronization primitives, such as the mutex. It is a program that must be thread-safe when accessing a shared data. When constructing a lock-free program, primitives like the CAS (compare-and-swap) is used. The CAS is a type of instruction that has three parameters: the address of object x, value it is expected e, and value that it needs to change to v. When it is executed, CAS checks if x is equal to e. If true, then the value is updated to v and returns true; otherwise, x is not altered and returns false.

5. Implementation Overview

5.1. Lock Based Implementation 1

The first implementation of a parallel NW algorithm makes use of a single lock and a pool, implemented as an vector array. Before the algorithm starts spinning up threads to do work, the first row/column of the scoring matrix is computed. These computations are only based on the previous single calculation. The main idea of this implementation is as follows: The first thread begins to calculate the first cell. When it is finished, the thread adds two items into the pool. These two items are the cells directly below and directly to the right. Adding these items to the pool indicates that they have at least one of the two scores needed to calculate its score. This cycle continues until the matrix has been complete. There are a few protocols added to ensure that out-of-bound cells are not being added to the pool and ensures that the values needed to compute the current cell value are complete.

In the pseudocode below, each cell is set to INT_MIN, minimum value for int, and once it computes the maximum score, the algorithm replaces INT_MIN with the new score. The pseudocode depicts two methods: thread_function and findvalue. thread_function is where each of the threads tries to dequeue an item from the pool, if possible. findvalue is where the algorithm finds the value of the cell and adds the two other values.

23452345PSUEDOCODE3523452345

The lock in this implementation allows us to ensure no issues arise when accessing the pool. For instance, if a thread takes an element from the pool to work on, it

may be possible that another thread attempts to access the same element. We must lock each time we enqueue an element into the pool, as well as whenever we check or dequeue an element.

This implementation uses a coarse grained locking scheme where this single lock becomes a bottleneck, especially with the addition of many threads. This implementation's performance will suffer greatly as it is scaled up with multiple CPUs(threads). As more CPUs are performing operations in this implementation, a higher number of threads must be stopped for one single thread whether it is enqueueing, dequeueing, or simply checking the pool.

5.2. Lock Based Implementation 2

In this lock-based implementation, we use the same approach of how to solve the problem. However, we use two different locks, one for enqueueing and one for dequeueing from the pool. Unlike the last method, there are two pointers used for the pool, since the pool is implemented as a linked list, in which we use head and tail pointers. This allows us to access the pool with less constraint. As long as the pool is not empty, threads will experience less time waiting for a lock to be released. This means we can asynchronously dequeue and enqueue while there is something in the pool. This is an attempt to speed up our first lock-based implementation and provide a more fine grained locking approach.

5.3. Lock Free Implementation

In the lock-free implementation, we use compare-and-swap operations when accessing the pool. Just as the lock-based #2 implementation, our pool is a linked list of nodes. However, each node's next pointer is an atomic reference in order to use compare-and-swap. Instead of locking all enqueueing threads when one thread is enqueueing, multiple threads can potentially enqueue at the same time if they are not enqueueing the same item. The enqueue operation traverses the list until it finds NULL and attempts to enqueue. If the item is already in the pool, nothing happens and the enqueue fails. Otherwise, the item is added to the pool. The dequeue operation follows similar logic to the enqueue. We attempt to dequeue an item with compare-and-swap. If it fails, meaning an item about to be dequeued has already been dequeued by another thread, the dequeue will retry. This continues until an item is successfully dequeued or the pool is exhausted, in which case the dequeue fails.

6. Experimental Results

The results gathered from our experimentation include running each algorithm on 5 different benchmarks. The benchmarks used are from real genomes of several animals. These genomes that were used are from the National Center of Biotechnology Information (NCBI). The benchmarks are substrings of the genome for each respective animal. The animals and genomic substrings that were used include: bison vs hamster(length of 500), Rabbit and Polar Bear(length of 974), Fish and Dolphin(length of 1200), Goat and Giraffe(length of 2000). We ran tests to quantify how each implementation performs against each other, as well as how they perform when varying the number of threads.

In Figure 1, we can see the performance of each implementation measured in milliseconds while varying the length of sequences that are aligned. We can see the first lock-based algorithm performs the worst out of all implementations. For the longest length sequence alignment of 2000 characters, this implementation takes 107,214 ms. The second lock-based algorithm takes 7182.29ms. Compared to the first lock-based algorithm, this is a significant improvement. Using two separate locks substantially increased the performance, due to less contention of the single lock. The lock-free algorithm performs the best out of our parallel implementations, however is not much of an improvement over our fine-grained locking approach. However, we see that as we increase the length of the sequences to be aligned, the performance begins to diverge. The sequential implementation of the Needleman-Wunsch algorithm outperforms all of the parallel implementations. We believe this is due to the overhead of synchronization taking a substantial amount of time compared to the computation that is allowed to happen in parallel.

Figures 2-4 show the effect of adding more threads to the parallel implementations. Each figure displays 4 groupings which represent a pair of aligned sequences sorted by length: 500, 974, 1200, 2000. Within each grouping, the 3 graphs represent 2, 3, and 4 threads, respectively. In each implementation, we notice that there aren't any significant performance increase with the addition of more threads. In fact, in some cases, the runtimes were worse with more threads added. However, this is most likely due to the randomness of the scheduler between executions.

7. Conclusion and Future Work

Parallelizing the Needleman-Wunsch algorithm to obtain a significant speedup is not a trivial task when attempting to optimally align two sequences. Concep-

tually, this algorithm is not very computationally intensive between synchronization operations. Because of this, the overhead brought on from the synchronization protocols within our proposed implementations substantially slows down this algorithm. However, employing different fundamental synchronization protocols does offer us speedup where possible.

Some future work that we would like to explore includes re-conceptualizing our approach to parallelizing the Needleman-Wunsch algorithm. One possibility is reversing the data that is stored within the pool. Instead of storing items that are ready to be calculated, store items that have already been calculated to only be dequeued once instead of twice.

8. Appendix

8.1. Challenges

The first challenge was to analyze the NW algorithm and figure out a way to generate a concurrent program. With myraid of methods already generated, we wanted to implement the lessons taught in class into the project. The first implementation was a lock-based method using a vector array pool. Although this algorithm provided a new insight to the NW, there were numerous of problems that occurred when testing larger test cases. Not only was it was going slow, it was also showing segmenation faults in some instances. After researching the problem and figuring out the solution, it was due to not locking in some of the places within the algorithm. After the changes were made, the segmentation fault dissolved; however, the bottleneck situation was still an issue.

In the second implementation, we immediately addressed the lock issued from the last method. Instead of using a vector array, a linked-list was used for the pool. In this way, two pointers can be used to enqueue/dequeue. Additionally, instead of one lock, two locks were used. Implementation two was able to decrease by more than sixty percent of the first one. However, this was still a lock-based method. We wanted to see if we can further decrease the time with a lock-free way.

Although, Implementation three was similar to the other three, it was a bit convoluted in where CAS should be placed; however, once it was understood where it should go, everything else fell into place.

References

[1] Carlos Torres Jaime Seguel. Paralellization of needleman-wunsch string alignment method. *BIOCOMP*, 1:239–244, 2011.

[2] W. S. Martins, J. B. Del Cuvillo, F. J. Useche, K. B. Theobald, and G. R. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. *Pacific Symposium on Biocomputing*, 6:311322, 2001.

[3] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195197, 1981.

[4] F. Zhang, XZ Qiao, and ZY Liu. Parallel divide and conquer bio-sequence comparison based on smith-waterman algorithm. *Science in China Series F-Information Sciences*, 47(2):221231, 2004.

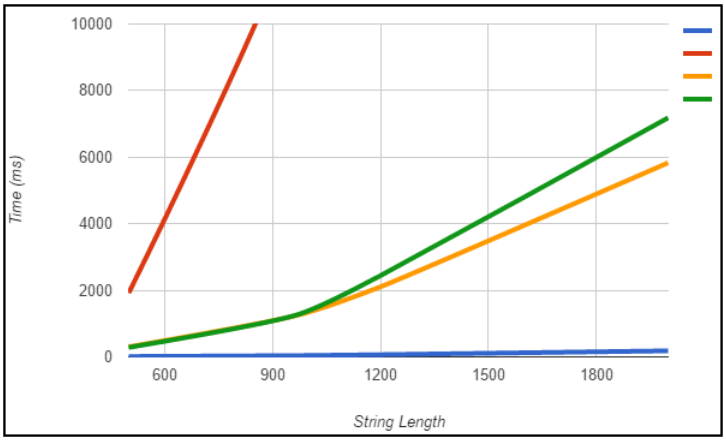

Figure 1. Implementation Runtimes

Red: Lock-Based #1
Green: Lock-Based #2
Orange: Lock-Free
Blue: Sequential

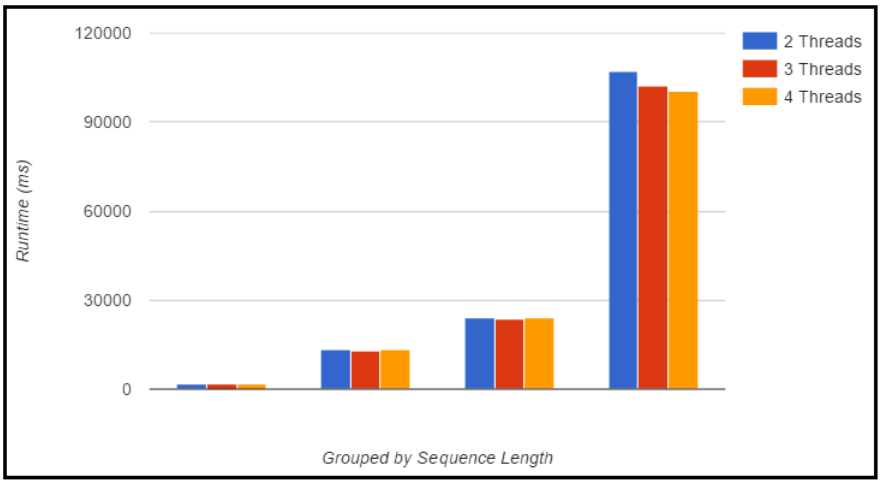

Figure 2. Lock-Based #1 Thread Scaling

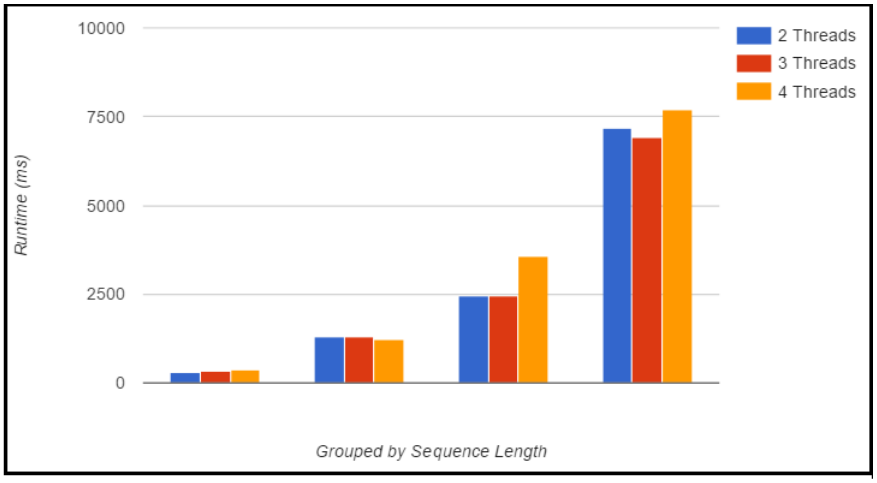

Figure 3. Lock-Based #2 Thread Scaling

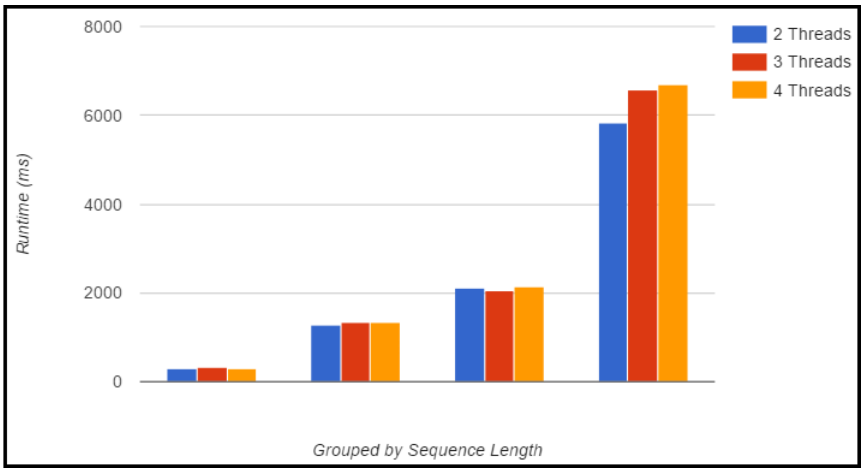

Figure 4. Lock-Free Thread Scaling