A Comparison of Shared and Distributed Memory Implementations for Global Sequence Alignment

by

Maxwell Farrington

Advised by Dr. Russ Miller

December 2024

A project write-up submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science
Department of Computer Science and Engineering

Abstract

Sequence alignment algorithms like the Needleman-Wunsch algorithm have a key role in areas such as bioinformatics for the calculation of the degree of similarity of nucleotides between genetic sequences. This project involved the implementation of sequential and parallelized versions of the Needleman-Wunsch algorithm using a shared memory system with OpenMP as well as a distributed memory system with MPI with the goal of comparing the scalability of the algorithms and learning more about the different overheads that one must consider for both systems. During this process, the benchmarks were run using SLURM batch scripts at the University at Buffalo Center for Computational Research (CCR) in order to reserve multiple nodes on similar hardware. From the benchmarks, it was found that while the OpenMP implementation had improved scaling over the MPI version relative to a sequential solution, the MPI version had better runtimes for similarly sized problems. While this could have been for a myriad of factors, it is hypothesized that the reason for this is because the Needleman-Wunsch algorithm is very parallelizable, such that the communication overhead for the distributed memory algorithm is significantly lower than the cost of thread synchronization on the shared memory version.

TABLE OF CONTENTS

Introduction

Sequence alignment is a task that is often done in the bioinformatics field to find the similarities when comparing protein and nucleotide sequences. The sequences are represented as strings with the characters representing different amino acids. Related sequences will contain portions of shared characters with differences coming in the form of the insertion, removal, or skipping of characters, often thought of as the mutations which related sequences can undergo as they evolve [1]. The Needleman-Wunsch algorithm is a dynamic programming algorithm which focuses on global alignment, which involves aligning two sequences in their entirety. This is done by finding a score which is the aggregate based on the number of matches and mismatches of characters, as well as insertions and deletions required to align the two sequences. The algorithm can be broken down into multiple components with the work split amongst processors to reduce running time. When this is done, extra consideration must be made to determine the scalability of the parallel implementation; the two most commonly used benchmarks being strong and weak scaling. Strong scaling determines how well a problem can be parallelized while keeping the problem sized fixed and increasing the number of processors while determining the speedup. An algorithm that exhibits good strong scaling will continue to become faster at roughly the rate you are increasing the number of processors. Weak scaling determines how well a problem scales when increasing the problem size alongside the number of processors by calculating the efficiency. This is done to determine if parallelization allows one to compute larger and larger problem sizes without a drop in performance. An algorithm which exhibits

good weak scaling will remain efficient as you increase the problem size and number of processors [4].

Problem Solution

Both the shared and distributed memory algorithms are based on the Needleman-Wunsch algorithm. The vanilla version of this algorithm involves the comparison of two sequences with assigning a score system that includes penalties for gaps such as insertions/deletions (indel) as well as a value for a match and mismatch. This score system can vary depending on the use case. Often, an affine gap penalty is preferred such that the opening of a gap is assigned some value g, and the extension of that gap adds another value, h where $g \geq h$ since consecutive gaps tend to relate to a single insertion/deletion whereas man scattered indels are likely separate occurrences [3]. This project focused on a linear scoring system for the sake of simplicity. After a scoring system is decided a scoring matrix of size $(m + 1) \times (n + 1)$, where m is the length of sequence A, and n is the length of sequence B is filled following that system. Row and column 0 represent if an indel were selected for each element in the row/column. Each element $m[i][j]$ of that matrix representing the maximum value depending on whether the characters in $A[i-1]$ and $B[j-1]$ are matched, mismatched, or should be considered an indel. If the maximum value corresponds to an indel, you look to the value to the left or top of the current element if it exists, $m[i-1][j]$ or $m[i][j-1]$ respectively, and add the associated penalty. Otherwise, you look to the top left value to the current element, $m[i-1][j-1]$ and add the associated value depending on whether the characters are matched or mismatched according to your scoring system. In the

vanilla version of the algorithm, you can then loop through the rows, columns, or anti-diagonals of the matrix to fill it and then retrieve the maximum score [2].

Distributed Memory Algorithm

The parallel algorithm for the distributed memory portion of the project was based on an algorithm by Aluru et. all [1] with modifications to support a fixed gap penalty. This algorithm is specifically designed to fill the matrix row-wise or column-wise. This is so the number of processors being used and work per processor remains constant. Starting from element $m[1][1]$ the calculation of each element of the score matrix is split into two parts. For each element you first calculate two values, $w[j]$ and $x[j]$ which follow figures 1 and 2 respectively. $w[j]$ represents the maximum of the values to the top and top left of the current element being computed while adding the corresponding values for indels or a match/mismatch just like in the vanilla algorithm. These are able to be calculated independently in parallel within each processor because the row above any given row has already been computed. Then the value $x[j]$ represents taking the maximum of $w[j]$and the value of $x[j-1]$ from the element to the left of the current element being computed. When calculating $x[0]$, the value of $x[j-1]$ can be substituted with $-\infty$. The reason that this value needs to be separated is because the sequences are split between processors, and each processor is unaware of the values from processors of lower rank. To get these values, there is a parallel prefix communication step with the maximum operator where the processors send their local maximums for $x[j]$. After this step, each processor will have the running maximum of *x* and can finish their computations by doing another round of maximum operations, and each element in the row will have their final value.

$$w[j] = max \begin{cases} m\,[i-1]\,[j-1] \;+\; match\,(A\,[i-1]\,,\; B\,[j-1]) \\ m\,[i]\,[j-1] \;-\; gap \end{cases}$$

Figure 1. equation for *w[j]*

$$x[j] = max \begin{cases} w\,[j] + \; jg \\ x\,[j-1] \end{cases}$$

Figure 2. equation for *x[j]*

The algorithm was implemented using MPI in C++ since the standard includes all the communication patterns required by this algorithm. The algorithm was then able to be benchmarked using resources from CCR. This involved writing a SLURM batch script and submitting jobs to the queue in order to reserve the resources needed to test the scalability.

Shared Memory Algorithm

The implementation of the shared memory version of the Needleman-Wunsch algorithm did not require any changing of the core part of the algorithm such as the distributed memory version. The implementation for this portion of the project was an anti-diagonal implementation of Needleman-Wunsch with added OpenMP pragmas to distribute the work to multiple cores in a shared memory machine. Implementing the anti-diagonal algorithm is especially useful in this case because of the lack of data dependencies and data races between cores. When computing a given element in the scoring matrix, you need the value to the left, top, and top left of the

element being calculated. As such, these values can all be obtained by the two previous diagonals which are guaranteed to have already been calculated as seen in figure 3. This property gets rid of any potential data races when splitting the work along the antidiagonal, but the downside of the anti-diagonal method is that the amount of work as you fill the matrix and will lead to an inefficient balancing of work scheduling among processing units [1]. This algorithm was implemented using OpenMP and C++ and was benchmarked similarly to the distributed memory algorithm.

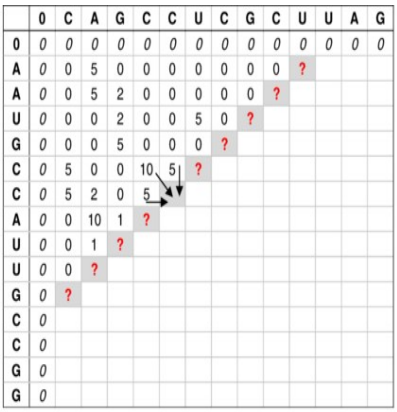

Figure 3. Example of the anti-diagonal method for Needleman-Wunsch [5]

Results

Distributed Memory Algorithm

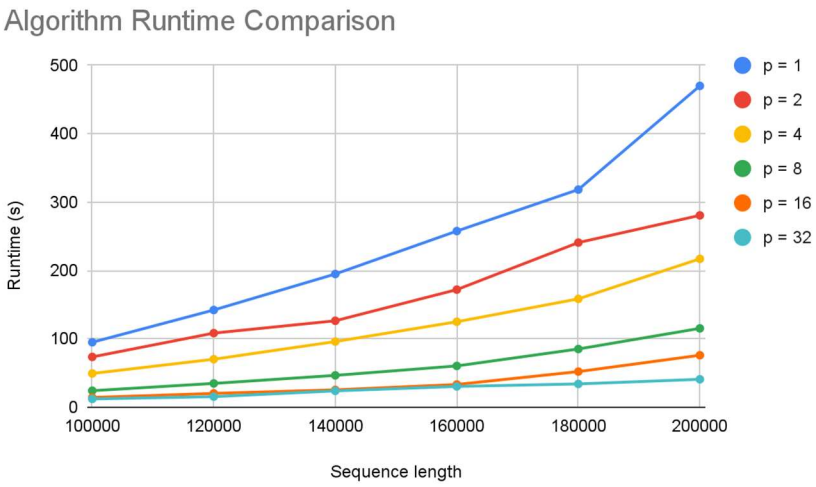

Figure 4. Runtime of MPI implementation for each processor orientation and problem size with 1 processor per node.

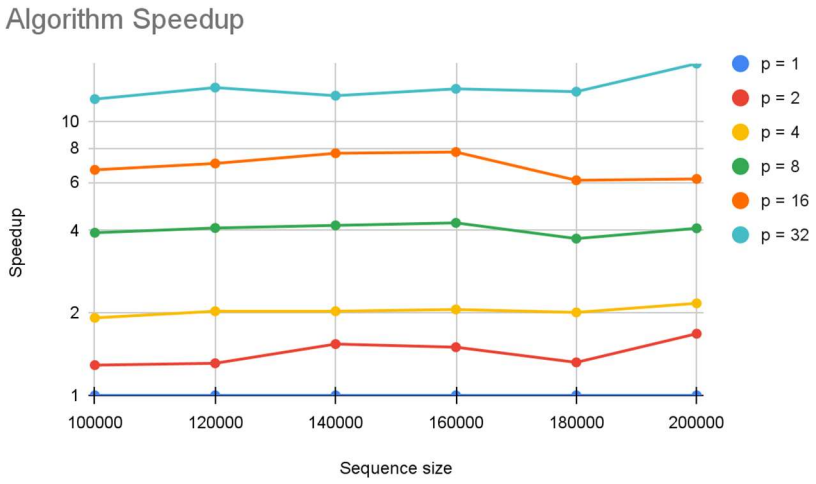

Figure 5. Speedup of MPI implementation for each processor orientation and problem size relative to $p = 1$ with 1 processor per node.

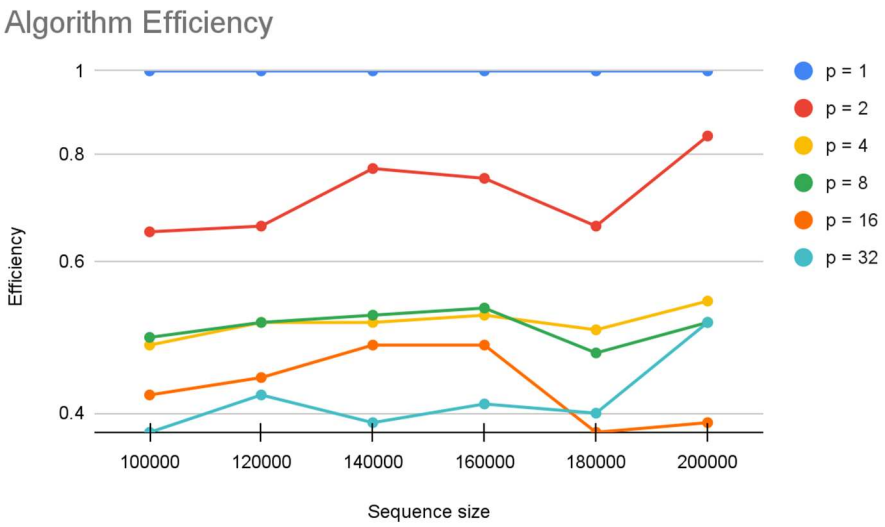

Figure 6. Efficiency of MPI implementation for each processor orientation and problem size relative to p = 1 with 1 processor per node.

Shared Memory Algorithm

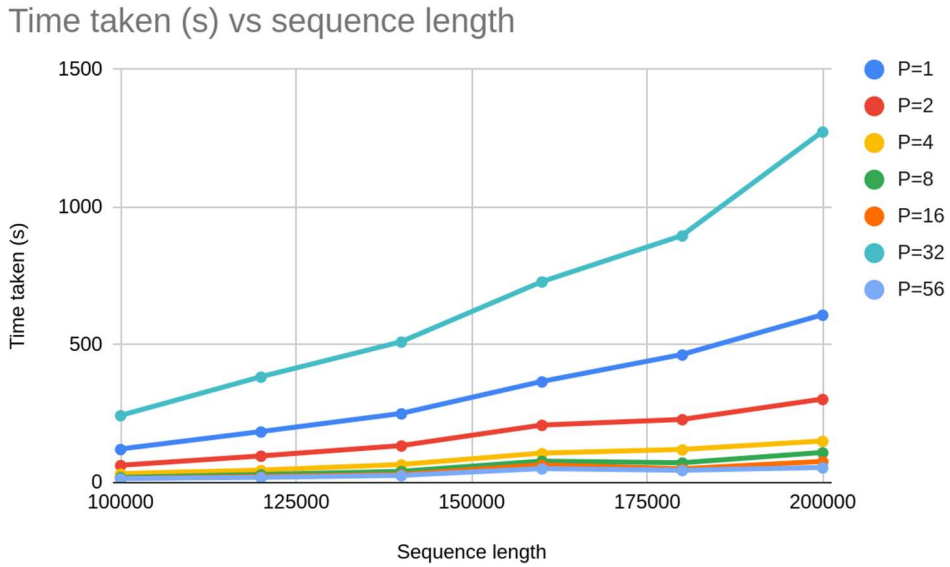

Figure 7. Runtime of OpenMP implementation for each processor orientation and problem size.

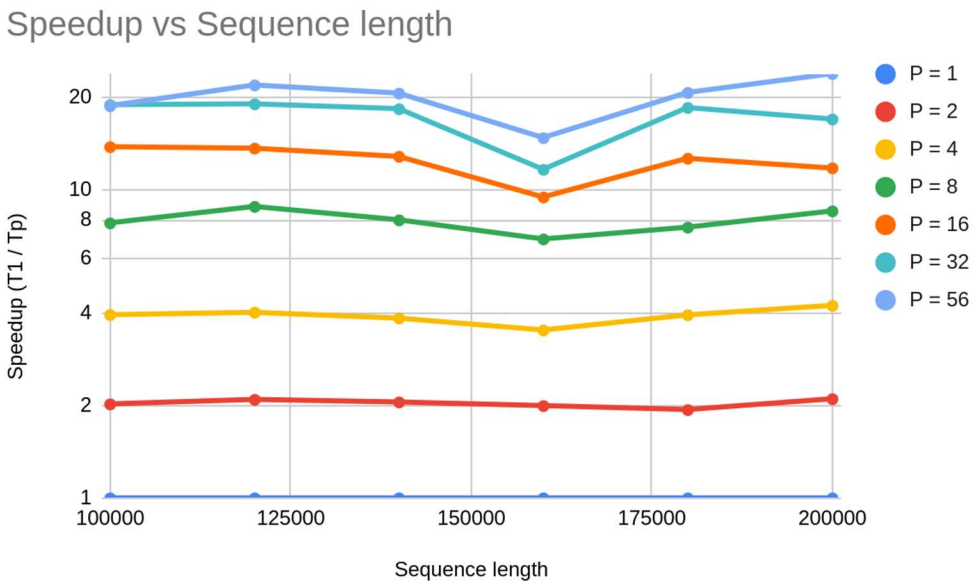

Figure 8. Speedup of OpenMP implementation for each processor orientation and problem size relative to $p = 1$.

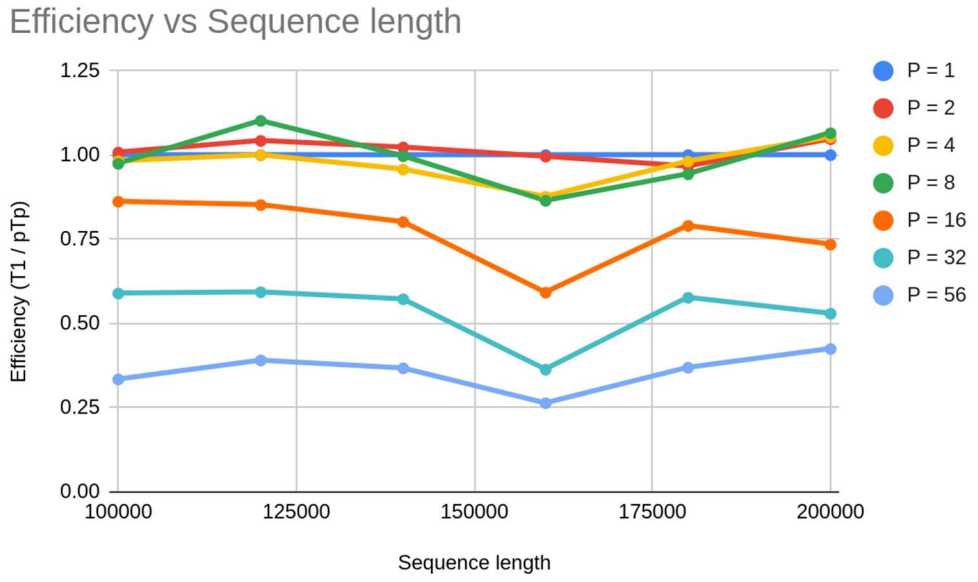

Figure 9. Efficiency of OpenMP implementation for each processor orientation and problem size relative to p = 1.

Conclusions

The results showed that while scaling for the shared memory implementation of the Needleman-Wunsch algorithm was better than that of the distributed memory version, the runtimes were still significantly better for the distributed memory implementation. For strong scaling, the distributed memory implementation hardly showed speedups on-par with the increases in number of processors as per Figure 5. For weak scaling, the only times the distributed memory implementation exhibited an efficiency above 0.75 was for select problem sizes with two processors as shown by Figure 6. One of the reasons this could be the case is because the benchmarks were run with 1 process per node to ensure the parallelism was being shown via actual communication instead of any simulated on-chip parallelism (i.e running a job with 1 node and 32 processes per node with MPI). Further testing fully utilizing on-chip parallelism would likely give better results in terms of runtime and scaling while also simulating conditions closer to what would be done in real world benchmarks. The shared memory algorithm showed clear strong and weak scaling up to 16 processors for almost all sequence lengths from 100000 to 20000 with OpenMP. Strong and weak scaling began to fall off around 32 processes likely due to the overhead of additional processor scheduling. Further testing on larger problem sizes for the shared and distributed memory algorithm could also be useful to test how these algorithms perform when needing to retrieve information from disk. In the real world, these biological sequences are often millions of characters long, and more creativity is necessary in optimizing the data retrieval for the filling of the score matrix.

Works Cited

[1] Srinivas Aluru, Natsuhiko Futamura, Kishan Mehrotra, Parallel biological sequence comparison using prefix computations, Journal of Parallel and Distributed Computing, Volume 63, Issue 3, 2003, Pages 264-272, ISSN 0743-7315

[2] Saul B. Needleman, Christian D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology, Volume 48, Issue 3, 1970, Pages 443-453,

[3] Aluru, S. (Ed.). (2005). Handbook of Computational Molecular Biology (1st ed.). Chapman and Hall/CRC. https://doi.org/10.1201/9781420036275

[4] Scaling - HPC Wiki, https://hpc-wiki.info/hpc/Scaling

[5] https://www.researchgate.net/figure/Anti-diagonal-method-and-dependency-of-the-cells_fig11_222408669