Parallelized Local Sequence Alignment Using OpenMP
Programming Assignment #2
CS677 High Performance Computing
October 7, 2020
Andrew Weston

OUTSTANDING WORK A

Abstract: Local sequence alignment involves approximate pattern matching to find similar sequences in two (or more) strings of characters. In genomic applications, finding a similarity between a new, unknown nucleotide sequence and a previously known nucleotide sequence can provide biologically significant information about structural, functional or evolutionary relationships. We present three parallel implementations of the Smith-Waterman algorithm for calculating local sequence alignment of the HIV-1 nucleotide sequence and a sample unknown HIV-1 Polymerase sequence. We also introduce a draft proof-of-concept implementation for computing a similarity matrix for the longest decoded DNA sequence (2.3 million bases) on a conventional workstation.

1. Introduction

1.1 Biology Background

The “central dogma of molecular biology” explains the flow of information (represented as a sequence of bases) between information-carrying biopolymers in living organisms. Deoxyribonucleic acid (DNA) is a molecule of two polynucleotide chains. Adenine, Thymine, Guanine, and Cytosine are nucleobases that comprise nucleotides. These nucleotides are grouped into triplet sets called codons which specify one of twenty amino acids. In turn, a sequence of amino acids specify a protein. Proteins are significant because they perform a vast array of biologically important functions including DNA replication, catalyzing metabolic reactions, etc.

1.2 Sequence Alignment

Sequence alignment is sometimes referred to as the “longest common subsequence” problem. Fundamentally, it involves *approximate* pattern matching to find similar sequences in two (or more) strings of characters. Unlike the find-and-replace functionality of a word processor, sequences can be inexact matches (every character in the substrings need not be the same). Any sequence alignment algorithm must handle this complexity by defining a scoring framework for how well different substrings “match”. Calculating approximate matches involves tallying points for matching individual characters, matching sequences of characters, penalizing mismatches or gaps in the strings, etc.

1.3 Smith-Waterman Algorithm

The Smith-Waterman(SW) algorithm is a form of dynamic programming, a technique in which the solution to a problem is recursively expressed as a function of similar subproblems at the preceding level. In this case, the subproblems are matching substrings. The first phase of the SW algorithm calculates the similarity matrix, and the second phase retrieves the local alignments. We focus our analysis and implementation on the first

(computationally-demanding) phase.

Input consists of sequences s and t, with |s| = m and |t| = n. String prefix similarity measures (i.e. matrix entries) are calculated using the equation:

$$F(s[1..i],t[1..j]) = \max\begin{cases} F(s[1..i],t[1..j-1]) - 2, \\ F(s[1..i-1],t[1..j-1]) + (\text{if } s[i] = t[j] \text{ then } 1 \text{ else } -1), \\ F(s[1..i-1],t[1..j]) - 2, \\ 0 \end{cases}$$

Additionally, the first row and first column of the similarity matrix are initialized with zeros to support the “look-back” aspect of the implementation where computing an element is dependent on accessing previously computed elements. Conceptually, at each step the algorithm calculates the optimal alignment for a prefix that ends with a gap in s or a gap in t (and hence incurs a gap penalty), or has a match/mismatch (and incurs a match score or match penalty respectively). A gap results in a score of (previous value – 2); a match is scored as (previous value + 1); and a mismatch is (previous value – 1). The highest score is the optimal alignment for that prefix.

Figure 1 below illustrates the dynamic programming technique as applied by the similarity scoring equation. The recursive nature of the equation is expressed in such a way that the value of a matrix element depends on the values of its neighbors to the “north”, “west”, and “northwest”.

Figure 1

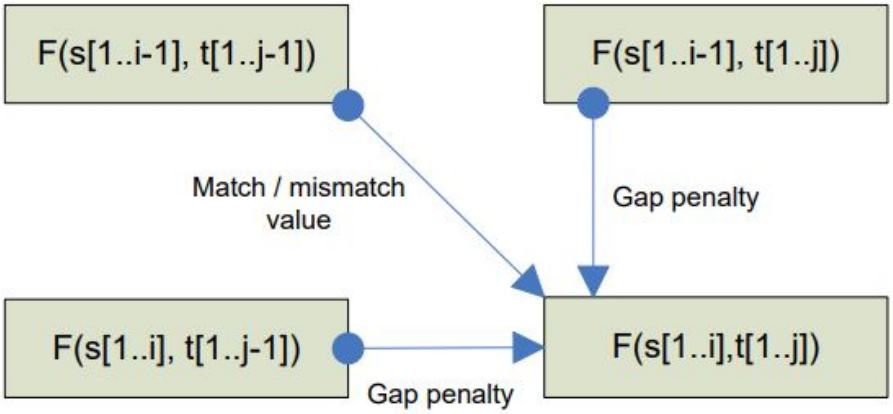
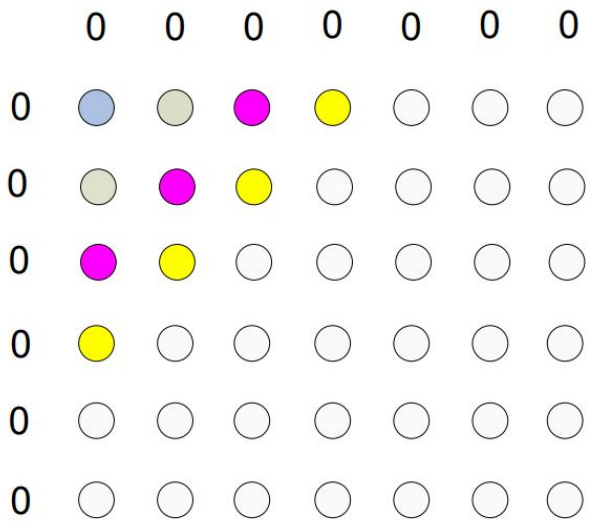


Figure 2 below illustrates how this data dependency creates a “wavefront” as elements become available for computation at each timestep. At the first timestep, only element (1,1) can be computed (the blue sphere). Once that result is available, elements (1,2) and (2,1) can be computed (the grey spheres), as the data they depend on is now available. Then elements (1,3), (2,2), and (3,1) can be computed (the pink spheres), and so on.

Figure 2



2. Solutions using Smith-Waterman Algorithm

2.1 Common Components

First, we present the algorithm components that are common across all of our Smith-Waterman implementations. Then we discuss those implementations individually in detail. Any solution must build the nucleotide sequences, s and t, by reading the source files' text and producing a string of the characters 'A', 'T', 'C', 'G' (representing Adenine, Thymine, Cytosine, and Guanine), and '?' (representing wildcard) . Next, it should create a two-dimensional "similarity matrix" based on the lengths of the input strings. Every element of the matrix is initialized to a sentinel value representing work "to-do" with the first row and column thereafter assigned to 0 for look-back dependencies. Then it should snapshot the start time, call the (implementation-specific) SW function, and snapshot the calculation completion stop time. Optionally, the program can print a representation of the similarity matrix and the program results. See sample output in Appendix A.

Additionally, the calculation of a particular element of the matrix is performed uniformly regardless of the specific Smith-Waterman implementation. Appendix B line 132 shows how the calc_element function performs a busy-wait until the element of interest's north and west neighbors have been computed. This busy-wait is unnecessary for the single-threaded sequential solution, but incurs minimal overhead, and allows all of the implementations to use the same calc_element helper function. This uniformity makes focusing on the structure of the parallel algorithms themselves easier.

Additionally, each element has a data dependency on its north, west, and northwest neighbors, but we can eliminate the check for the northwest dependency because the element's north and west neighbors also have this dependency. Essentially, if the element's north neighbor or west neighbor have been accurately computed then the element's northwest neighbor *must* have been accurately computed. Once all dependent computations are performed, calc_element performs a straightforward maximum calculation based on the point tally description in Section 1.3.

Finally, except for the strictly sequential approach in Section 2.2, all of our solutions leverage the shared-memory multiprocessing application programming interface (API) OpenMP. OpenMP is designed to incrementally enhance a sequential program with parallelism. Accordingly, we explain our solutions beginning with a sequential program, then considering a "parallelized" version of the sequential program, and then considering alternative parallel implementations.

2.2 Sequential Solution

Overview: The sequential solution uses nested loops to iterate through the similarity matrix element-by-element.

Synchronization: Since only one thread is used, no synchronization of updates to variables for tracking the maximum value element is required.

Data Dependencies: Since loop iteration occurs in the same direction as the wavefront ripples toward higher indices, this function automatically handles the SW data dependencies.

Summary: The sequential solution is a simple, intuitive, baseline for comparing parallel implementations.

2.3 Sentinel-Synchronization Solution

Overview: The sentinel synchronization solution is a minimalist modification of the sequential solution to leverage OpenMP. Nested loops are still used to iterate through the similarity matrix element-by-element. However unlike the sequential solution, multiple threads execute the iteration.

Synchronization: At initialization the entire matrix is set to the sentinel value to identify "work to be completed". Then, the first row and column are re-initialized to 0 for the "look-back" attribute explained in Section 1.3. In this way, parallel threads reading an element can deduce that the element has not been computed if it is still set to the sentinel value.

This solution uses private variables for the matrix element computations as well as "omp critical" synchronized access to the variables for tracking the maximum value element.

Data Dependencies: The SW algorithm's wavefront is substantially different from the OpenMP default partitioning of parallel for loops into equal "chunks" (where independent calculations are assumed). Thus, threads assigned to relatively high indices are likely to perform some busy-waiting while the wavefront progresses outward.

Summary: Failing to explicitly tailor the parallelization to the algorithm's data dependencies means the sentinel solution will probably improve performance relative to the sequential solution, but it will probably not produce the Speedup of a carefully tailored parallel algorithm.

2.4 Queue-Synchronization Solution

Overview: The queue synchronization solution is an alternative modification of the sequential solution using a job-queue and a mutex for synchronization. The job queue is filled with tuples representing indices of elements to be computed. For example, inserting the task for computing the first element of the similarity matrix might look like:

```
job_q.push(make_tuple(1, 1));
```

After the job queue is filled with the workload, threads can synchronously 1. Acquire the job queue mutex 2. Pop a tuple representing an element computation "job assignment" and 3. Release the mutex. While the job queue has remaining work, threads repeatedly and synchronously get assignments and then perform computations.

Synchronization: Shared access to the job queue is protected from concurrent modification by a mutex. A thread must acquire the mutex before popping its job assignment.

Like the sentinel approach, access to the variables for tracking the maximum value element is synchronized with an "omp critical" block.

Data Dependencies: Since the job queue is filled with element indice tuples in sequential order (via nested loops) jobs are guaranteed to be *removed* from the queue in an order compatible with SW data dependencies, but this alone is inadequate to guarantee synchronization of *execution*. The calc_element function's busy-wait check is required for the queue solution because variation in thread scheduling could easily cause an element popped from the queue before an element to be computed after that element.

Summary: A job queue is a natural approach to managing a workload with data dependencies, but shared access to a single data structure is an obvious bottleneck. This approach may have better performance with coarser-grained computations (where the queue synchronization overhead can be amortized across more time computing an element) or where the data dependencies of the problem are so complicated that tailoring individual thread behavior to manage the dependencies is infeasible.

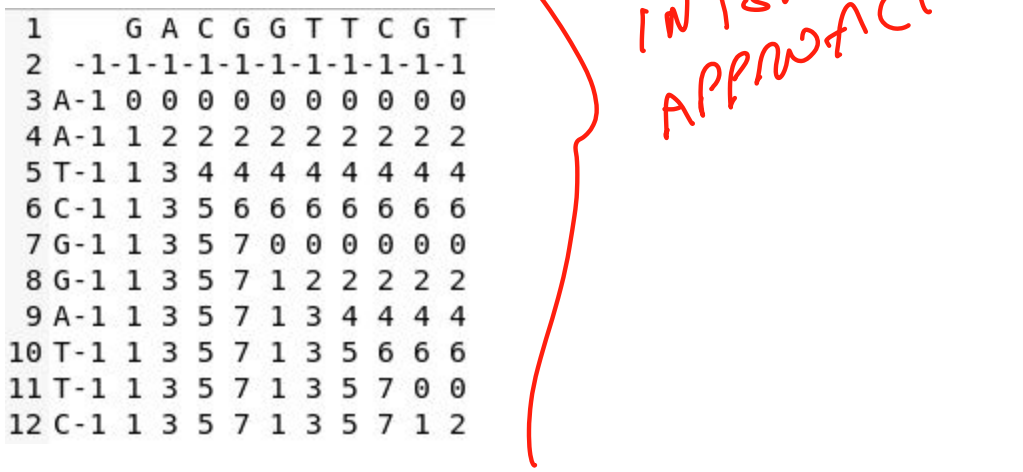
2.5 Strip-Partition Solution

Overview: The strip partition solution strives to tailor the parallel program structure to the problem's data dependencies. A strip is a partial row or column where the only data dependencies for calculating all of the elements in the strip are either 1. From an element in the strip itself, or 2. From an element in a strip earlier in the sequence of strips. Thus, a thread that is assigned a strip can eliminate all of the data dependencies from the first case by computing elements from left to right or top to bottom. Since threads are assigned their strips in strip_index order, the strips are likely to be computed (approximately) in order. In other words as each strip in the sequence of strips is tasked to a thread, that strip is likely to begin computation before any subsequent strip begins computation. Theoretically, this

reduces the likelihood of busy-waiting for data dependencies in the second case as well.

Figure 3 shows eight threads identifying their strip assignments via their thread numbers. Line 3 shows the first strip - all 0's. Likewise the column with the first 'G' shows the second strip - all 1's. Line 4 shows the third strip - all 2's and so on.

Figure 3



```
 1     G A C G G T T C G T
 2  -1-1-1-1-1-1-1-1-1-1
 3 A-1 0 0 0 0 0 0 0 0 0
 4 A-1 1 2 2 2 2 2 2 2 2
 5 T-1 1 3 4 4 4 4 4 4 4
 6 C-1 1 3 5 6 6 6 6 6 6
 7 G-1 1 3 5 7 0 0 0 0 0
 8 G-1 1 3 5 7 1 2 2 2 2
 9 A-1 1 3 5 7 1 3 4 4 4
10 T-1 1 3 5 7 1 3 5 6 6
11 T-1 1 3 5 7 1 3 5 7 0
12 C-1 1 3 5 7 1 3 5 7 1
```


See Appendix C for a sample of the strip assignments for the HIV computation.

Assuming string t is shorter or equal in length to string t, every character of t will start a strip. Likewise, every element of s (except for the one already handled by a horizontal strip) will be a vertical strip. However, since the number of vertical strips is also constrained by the length of string t we require a minimum operation. Altogether the total number of strips is given by:

$$totalStrips = t.length + min(s.length - 1, t.length)$$

Synchronization: The strip partition solution synchronizes threads using a mutex for the access to the maximum value element variables. Appendix B shows how a natural distinction between horizontal and vertical strips precludes the use of one "omp critical" section for updating the variables for tracking the maximum value element. Instead, a simple mutex guarantees threads synchronize their modifications.

Data Dependencies: As with the queue implementation, a busy-wait check in the calc_element function is still necessary. Specifically, strips are likely to be executed in strip_index order, but this is not guaranteed.

Summary: We expect the strip partition solution to improve performance relative to the sequential solution (as well as any of the other parallel solutions) through intentional parallelism with limited contention and overhead.

2.6 Solution Validation

During development, we validated the correctness of all of our SW implementations by manually calculating a test bench and then verifying each solution produced the same results. First, we manually computed two small similarity matrices as a test bench. Appendix D shows the results of our calculations. Then we developed the sequential SW solution explained in Section 2.2. Appendix E shows the sequential program output matches the manually calculated test bench. This gave us confidence that the sequential program was correct. Then, we applied the sequential solution to the larger HIV and HIV-Polymerase problem. An informal inspection of the program results further validated our sequential program. Thereafter, we used the manually computed examples and the results from the sequential program to validate the correctness of the parallel solutions. See Appendix F for a sample BASH script for verifying implementations' similarity matrices match.

3. Experimental Results

3.1 Overview

We determined experimental results on a Grand Valley State University Exploratory Operating System workstation. The machine has an Intel quad-core processor (eight logical cores). We have run ten iterations of each solution and analyze only the average runtime for a given solution. A sample script and full details of every trial's performance are available in Appendix G and H respectively. Additionally, we note that Speedup is defined by the equation:

$$Speedup = Time_{Sequential} / Time_{Parallel}$$

Figures 4 and 5 show that the Strip-Partition solution had the best performance - computing the similarity matrix in approximately 6.3 ms and resulting in a 3.2x speedup. Notably, the Strip-Partition solution was only marginally faster than the Sentinel-Synchronization solution which computed the similarity matrix in 7.4 ms and resulted in a 2.8x speedup.

Figure 4

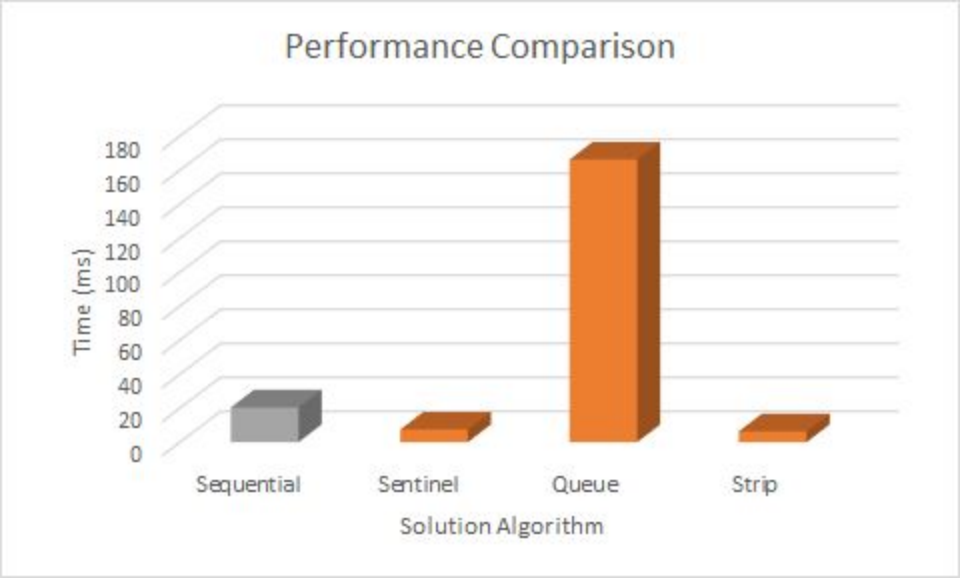
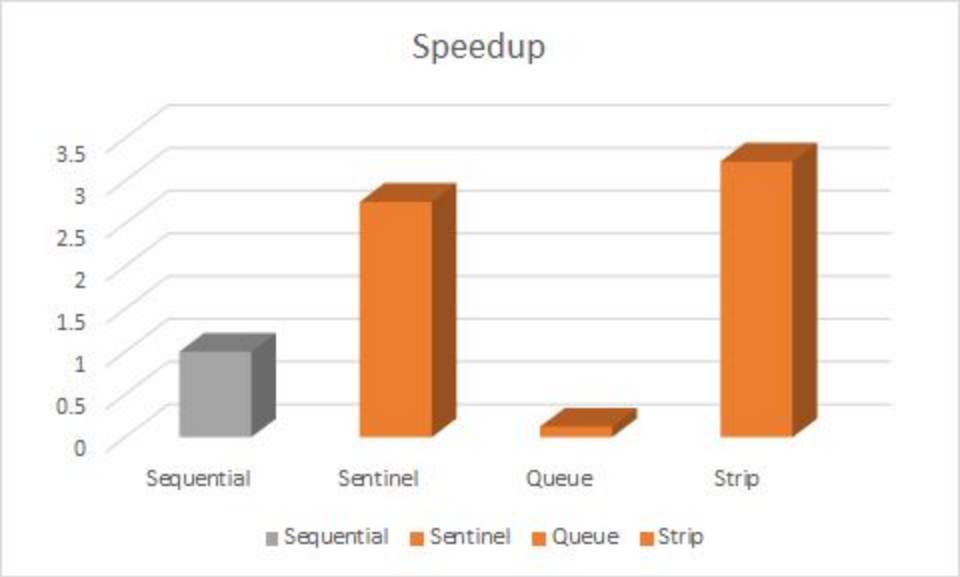


Figure 5



3.2 Sentinel-Synchronization Solution

The Sentinel-Synchronization solution was able to compute the HIV similarity matrix in 7.4 ms and resulted in a 2.8x speedup. While it was marginally slower than the Strip-Partitions solution, we note emphatically that because this implementation was a direct translation of the sequential program, it was substantially easier to write and debug.

Figure 6 shows the change in performance as we added more threads to the computation. The best average runtime improved until we reached the seventh thread when adding additional threads began to degrade performance.

Figure 6

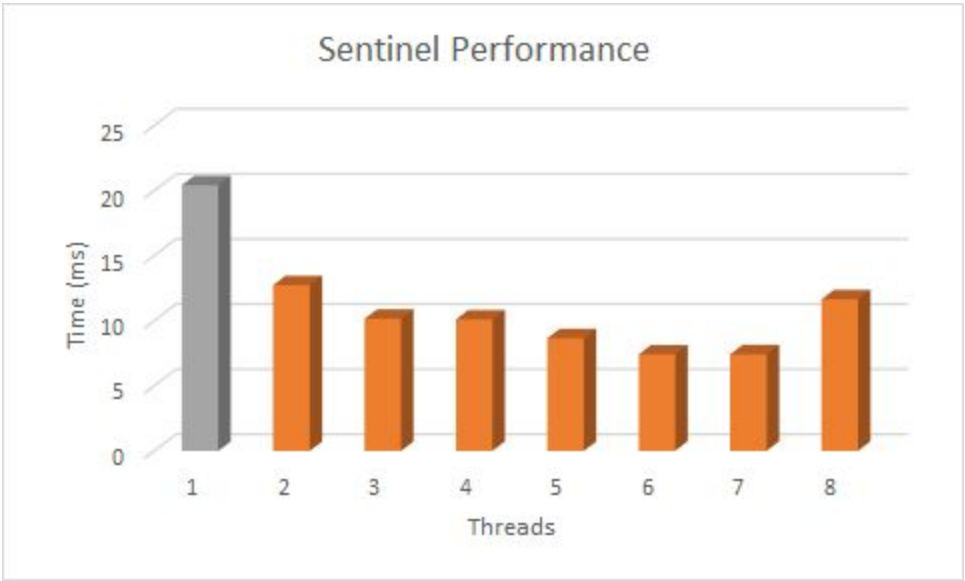
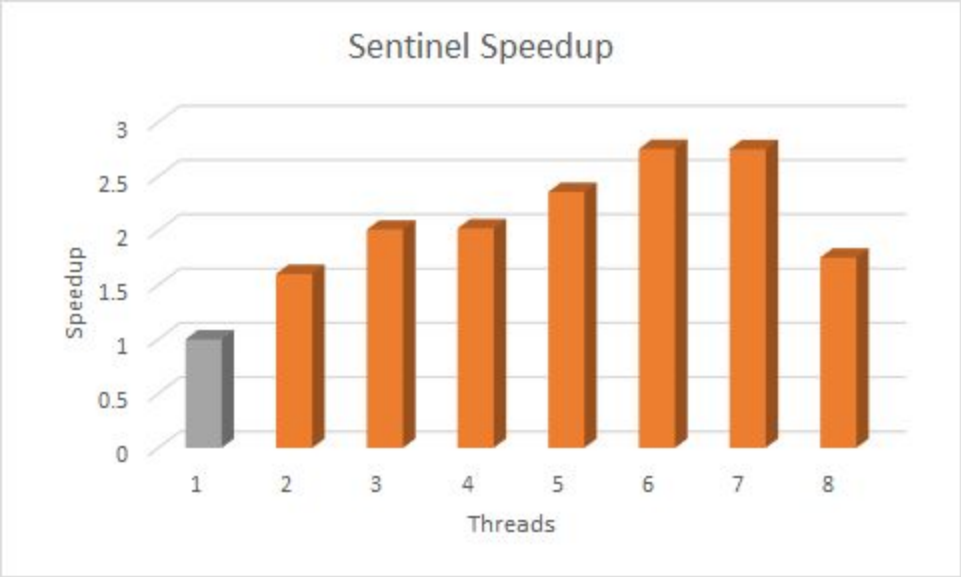


Figure 7 shows the commiserate increase in Speedup. Six threads produced the best Speedup - approximately 2.8x.

Figure 7



3.3 Queue-Synchronization Solution

The Queue-Synchronization solution substantially worsened performance regardless of the number of threads employed. Figure 8 shows that the queue implementation was at least 8x *slower* than simply using the sequential solution. This produced the atypical "Speedup" chart in Figure 9.

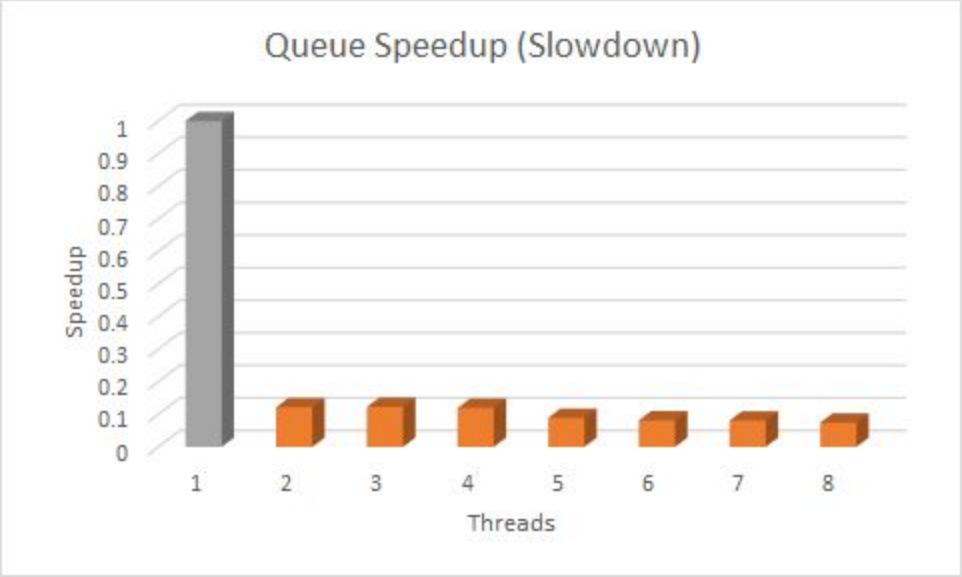
This decrease in performance is probably the result of several factors. First and foremost, the shared job queue is an obvious bottleneck. We needed to use a synchronization mechanism to ensure multiple threads could not access the queue concurrently. Unfortunately, computing an element of the matrix is a relatively fine-grained workload. Thus, threads have to frequently return to the queue to get the indices for their next job assignment. Ultimately, this probably means that a thread would pop an element assignment while the other threads waited on the mutex queue. Then, after quickly performing its computation the thread would join the back of the mutex queue and wait behind the other threads. One solution to this problem is for the queue to store larger "job assignments" than calculating a single element. For example, the queue could store the row and column strips from the Strip-Partition Solution.

Another issue with the Queue-Synchronization solution enqueueing job assignments of individual elements is that assigning elements in the same row or column to different threads undermines the cache architecture. In contrast, the sequential solution computes element (1,1) first. Then it advances to element (1,2). During the calculation of (1,2), the thread needs the north, west, and northwest neighbors to perform the calculation: elements (0,2) (0,1), and (1,1) respectively. Conveniently, (1,2)'s northwest and west neighbors would already be cached from the previous computation. With the job queue implementation, different threads would perform these computations. Thus, we would expect two additional

memory accesses for most elements. Like the queue access bottleneck, this problem could be mitigated with enqueuing larger “job assignments” than calculating a single element. Threads assigned an entire row or column would benefit from caching as with the sequential and strip partition solutions.

Figure 8

Figure 9



3.4 Strip-Partition Solution

The Strip-Partition solution achieved the best performance of all of our implementations. Six threads computed the similarity matrix in approximately 6.3 ms and resulted in a 3.2x speedup. As with the Sentinel-Synchronization solution, Figure 10 shows that adding a seventh thread begins to degrade performance.

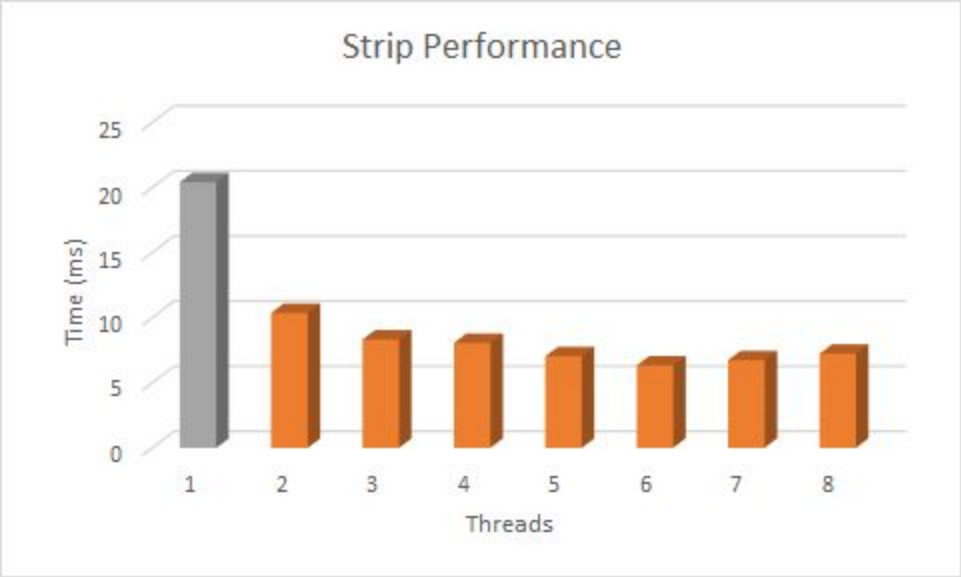
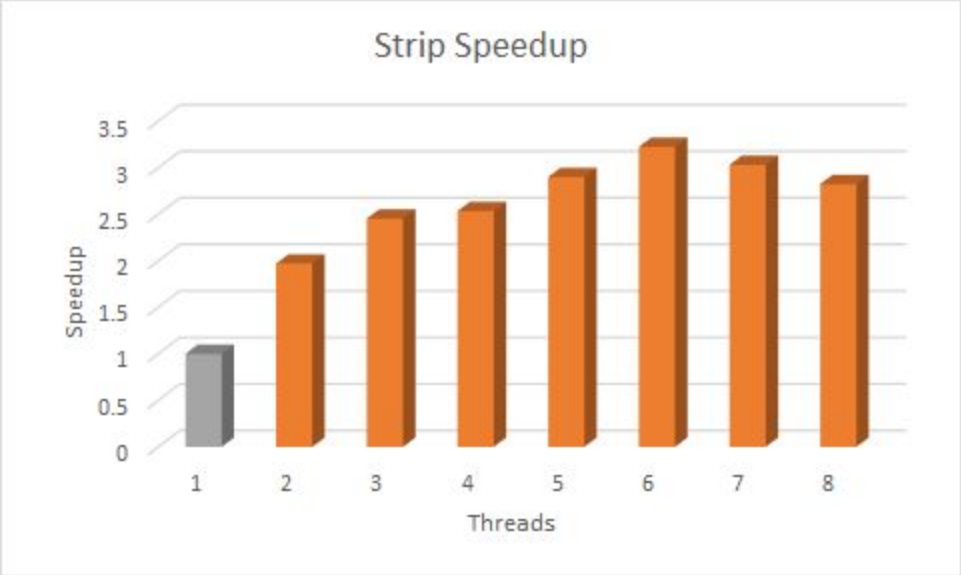


Figure 11 shows the strip solution Speedup with varying numbers of threads. Using between five-to-eight threads achieved more Speedup than even the Sentinel-Synchronization solution.

Figure 11



As explained in Section 2.5, the Strip-Partition solution was designed to tailor the parallel algorithm to the SW data dependencies. Strip partitioning divided the work into nearly independent strip "job assignments". Where dependencies between strips were unavoidable, strips were assigned in an order compatible with those dependencies. This reduced the likelihood of one thread waiting for another to compute an element.

The parallel algorithm used relatively coarse-grained job assignments to leverage the cache architecture and minimize overhead. The result was our highest performance solution.

4. Parallelizing Computation of a Piecemeal Similarity Matrix

The SW Algorithm is a well-established approach for calculating local sequence alignment. However, a major drawback of the algorithm is the memory requirements of supporting the similarity matrix. Specifically, the program must allocate $(s + 1) * (t + 1)$ integers for the matrix alone. As strings s and t grow to even moderate size, the size of the similarity matrix exceeds the physical memory of most machines.

An interesting question with practical significance is whether the longest known gene sequence (approximately 2.3 million bases) could be compared to an equally long sequence on a conventional workstation[1]. The capability to perform such computations on an affordable workstation that is nearly universally accessible to biology researchers would enable them to nimbly identify, compare, and analyze the results of similarity matrices without external support and special hardware.

We present a simple draft implementation that modifies our Sentinel-Synchronization approach from Section 2.3. At the scale of a 2.3 million x 2.3 million integer matrix, we must find a way to partition the array in a way that allows for parallel computation but can be held in physical memory. Our simple approach is to hold seven rows of the matrix in memory at any time. Based on the empirical results in Section 3, we use six threads to compute six rows of the matrix. Then we write the results out to disk. Next, we copy the bottom row to the top (to support the next six rows of computation) and repeat.

Appendix I shows our draft implementation. While the effect of "rotating windows" does work in this implementation, it is only a proof of concept. The implementation does not properly handle the last window of six rows. Moreover, it has only been preliminarily validated.

Further study to dynamically tailor the amount of memory used to that which is available, to concurrently write to disk while performing computation, and to implement our higher performance Strip-Partition solution is required.

INTERESTING EXERCISE

5. Problems and Solutions

5.1 Correctness - As explained in Section 2.6, we made a concerted effort to ensure our program was correct. By manually calculating two sample similarity matrices independently and then checking our answer with a solution, we are reasonably confident they are correct.

---

[1] Davison, A.

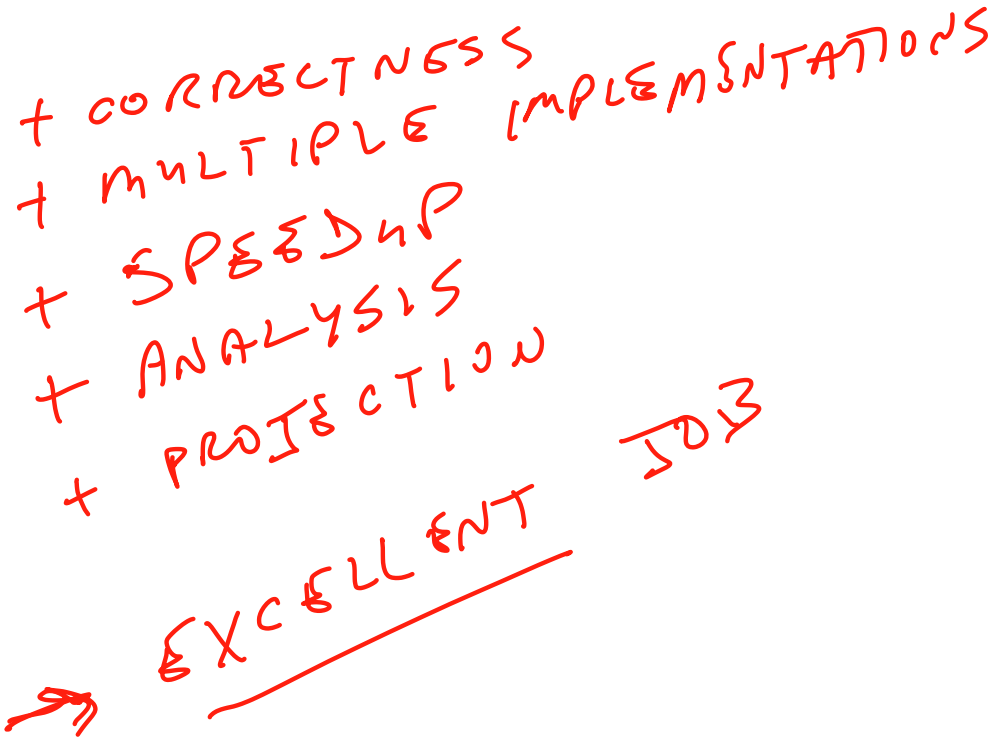
There is some risk in assuming that the sequential solution is correct for larger problems. We manually inspected a small subset of the computations (the most we could feasibly do) to further ensure correctness. This is probably just part of the nature of High Performance Computing (HPC). Problems that are large enough to require HPC are too large to truly ensure correctness. We simply need a reasonable validation strategy.

5.2 OpenMP - This was my first project using OpenMP beyond simple exercises to understand the API's constructs. I realized my Integrated Development Environment (IDE) was executing my parallel functions sequentially because I had not configured it properly. Ultimately this was somewhat constructive because my intuition was that I should have been getting better performance which led me to identify and resolve the problem.

5.3 Parallel Synchronization - The Sentinel-Synchronization solution was fairly straightforward to write after developing the Sequential solution. However, writing and debugging the Queue-Synchronization, the Strip-Partition, and an incomplete Wave solution (where each color of wave in Figure 2 is assigned to a thread) were substantially more complex and error-prone. In some cases using synchronization primitives like mutexes was necessary. In the future this may be a clue that a much more complicated solution is only going to return a marginal increase in performance.

References

[1] Davison, A. (2018, December 28). Record for decoding the longest DNA sequence is impressive – here's what to expect next. Retrieved October 07, 2020, from https://phys.org/news/2018-12-decoding-longest-dna-sequence.html.



Appendix A - Sample Output of Similarity Matrix for HIV1 and HIV Polymerase

```
    T T T T T T A G G G A A A A T T T G G C C T T C C A ? C A A G G G G A G G C C A G G ? A A T T T
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 0 1 1 0 0 0 1 1 1 0 0 0 0
G 0 0 0 0 0 0 0 0 1 2 2 0 0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 1 0 0 0 1 2 2 2 0 1 2 0 0 0 1 2 2 0 0 0 0
T 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0 0 0 0 1 1 1 0 0 1 0 0 0 0 3 1 0 1 1 1
C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 2 1 0 1 2 0 0 0 0 0 0 0 0 0 1 2 0 0 0 1 2 0 0 0 0
T 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 3 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 1
C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 1 0 1 2 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0
T 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 2 2 1 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1
C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 3 2 0 1 2 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0
T 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 2 1 1 2 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1
G 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 2 1 0 0 0 1 0 0 1 2 0 0 0 1 1 1 1 0 1 1 0 0 0 1 1 1 0 0 0 0 0
G 0 0 0 0 0 0 0 0 1 2 2 0 0 0 0 0 0 0 1 3 1 0 0 0 0 0 0 2 1 0 0 1 2 2 2 0 1 2 0 0 0 1 2 2 0 0 0 0 0
T 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 1 1 0 1 2 0 1 1 0 0 0 1 1 0 0 0 0 1 1 1 0 0 1 0 0 0 0 3 1 0 1 1 1
T 0 1 2 2 2 2 2 2 0 0 0 0 0 0 0 0 1 2 2 0 0 0 1 1 2 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 1 2 2
A 0 0 0 1 1 1 1 3 1 0 0 1 1 1 1 0 0 1 1 0 0 0 0 0 1 0 1 1 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 1 2 3 1 0 1
G 0 0 0 0 0 0 0 0 1 4 2 1 0 0 0 0 0 0 0 2 2 0 0 0 0 0 0 0 2 0 0 0 2 1 1 1 0 2 1 0 0 0 2 1 1 0 1 2 0 0
A 0 0 0 0 0 0 0 0 1 2 3 1 2 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 2 0 1 0 0 1 0 1 2 2 1 0 1 0
C 0 0 0 0 0 0 0 0 0 0 0 1 2 0 1 0 0 0 0 0 0 0 0 2 2 0 0 1 1 0 2 2 0 0 0 0 0 0 0 1 0 2 1 0 0 0 2 1 1 0 0 0
C 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 3 1 0 1 2 0 1 3 1 0 0 0 0 0 0 0 0 1 3 1 0 0 1 1 0 0 0 0
A 0 0 0 0 0 0 0 0 1 0 0 0 1 2 1 1 0 0 0 0 0 0 0 1 2 0 0 0 3 1 1 4 2 0 0 0 0 1 0 0 0 1 4 2 0 1 2 2 0 0 0
.
.
.
A 0 0 0 0 0 0 0 2 0 0 0 2 1 1 1 0 0 0 1 2 0 0 0 2 3 4 4 2 0 3 2 0 0 0 0 2 0 0 0 0 1 0 0 3 2 1 0 0 0
G 0 0 0 0 0 0 0 0 0 3 1 1 0 1 0 0 0 0 0 0 1 2 1 0 0 0 1 2 3 5 3 1 2 3 1 1 1 0 3 1 0 0 0 2 1 1 2 1 0 0 0
T 0 1 1 1 1 1 1 1 0 1 2 0 0 0 0 0 1 1 1 1 0 0 1 0 1 1 1 0 0 1 4 4 2 0 1 2 0 0 0 1 2 0 0 0 0 1 2 0 1 2 1 1
G 0 0 0 0 0 0 0 0 0 1 2 3 1 0 0 0 0 0 0 0 0 2 1 0 0 0 0 0 0 0 0 2 3 3 1 1 1 2 3 1 0 1 2 1 0 0 1 1 2 1 0 0 1 0
C 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 0 0 0 0 0 1 2 1 0 0 1 1 0 1 3 2 2 0 0 1 2 0 0 0 3 2 0 0 0 2 1 0 0 0 0
T 0 1 1 1 1 1 1 1 0 0 0 0 0 1 0 0 1 1 1 0 0 0 1 2 1 0 0 0 1 1 2 1 1 0 0 0 1 0 0 1 2 1 0 0 1 1 0 1 1 1
T 0 1 2 2 2 2 2 2 0 0 0 0 0 0 0 0 1 2 2 0 0 0 0 2 3 1 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 2 2
C 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 4 2 0 1 2 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1
A 0 0 0 0 0 0 0 2 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 2 3 3 1 0 3 1 0 0 0 0 1 0 0 0 0 2 0 0 1 2 1 0 0 0
Time: 16131.2 microseconds
max = 42, i = 1685, j = 48
```

Appendix B - SW C++ Implementations

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <vector>
 4 #include <string>
 5 #include <fstream>
 6 #include <algorithm>
 7 #include <chrono>
 8 #include <omp.h>
 9 #include <queue>
10 #include <mutex>
11
12 using namespace std;
13
14 /* Function to parse input file into compatible string */
15 string parse_input(const string &in_file);
16
17 /* Helper function to profile memory usage */
18 int get_memory_snapshot();
19
20 /* Helper function to print similarity matrix state */
21 void print_matrix(string &s, string &t,
22                   const vector<vector<int>> &sim_matrix);
23
24 /* Helper function to print similarity matrix results */
25 void print_results(int matrix_max, int max_i, int max_j,
26                    chrono::duration<double, micro> micro_secs);
27
28 /* Helper function to perform the max operation from SW algorithm */
29 int calc_element(string &s, string &t, int &i, int &j,
30                  vector<vector<int>> &sim_matrix);
31
32 /* Single-threaded, nested loop "baseline" implementation of SW
33  * algorithm */
34 void sequential(int &matrix_max, int &max_i, int &max_j,
35                 string &s, string &t, vector<vector<int>> &sim_matrix,
36                 int threads);
37
38 /* Multi-threaded, implementation of SW algorithm using "incomplete"
39  * sentinels */
40 void sentinel_sync(int &matrix_max, int &max_i, int &max_j, string &s,
41                    string &t, vector<vector<int>> &sim_matrix,
42                    int threads);
43
44 /* Multi-threaded, implementation of SW algorithm using "job queue" */
45 void queue_sync(int &matrix_max, int &max_i, int &max_j, string &s,
46                 string &t, vector<vector<int>> &sim_matrix,
47                 int threads);
48
49 /* Multi-threaded, implementation of SW algorithm using
50  * dependency-based "strips" */
51 void strip_partitions(int &matrix_max, int &max_i, int &max_j,
52                       string &s, string &t,
53                       vector<vector<int>> &sim_matrix, int threads);
54
55 /* The SW algorithm assigns 0 to the first row and column making the
56  * first calculated element at index (1,1)*/
57 const int BASE = 1;
58
59
60 /*****************************************************************
61  * This function serves as the program "driver" for an implementation
62  * of the Smith-Waterman Local Sequence Alignment algorithm using
63  * OpenMP. It reads in a genetic sequence file, builds a
```

```
 64  * "similarity matrix" using dynamic programming, and prints results
 65  * and performance data.
 66  *
 67  * @param argc is the number of command-line arguments.
 68  * @param v is the array of strings representing the command-line args.
 69  ********************************************************************/
 70 int main(int argc, char **argv) {
 71     if (argc != 5) {
 72         cerr << "Invalid Use\n Syntax is: file_name_str_s ";
 73         cerr << "file_name_str_t num_threads algorithm" << endl;
 74         cerr << "Algorithms are sequential sentinel queue or strip"
 75              << endl;
 76     }
 77
 78     int matrix_max = 0;
 79     int max_i = 0;
 80     int max_j = 0;
 81     string s = parse_input(argv[1]);
 82     string t = parse_input(argv[2]);
 83
 84     vector<vector<int>> sim_matrix(s.length() + BASE,
 85                                    vector<int>(t.length() + BASE, -1));
 86     for (int i = 0; i < s.length() + BASE; i++) {
 87         sim_matrix[i][0] = 0;
 88     }
 89     for (int j = 0; j < t.length() + 1; j++) {
 90         sim_matrix[0][j] = 0;
 91     }
 92
 93     int threads = stoi(argv[3]);
 94     auto start_time = chrono::high_resolution_clock::now();
 95     string algorithm(argv[4]);
 96
 97     if (algorithm == "sequential")
 98         sequential(matrix_max, max_i, max_j, s, t,
 99                    sim_matrix, threads);
100     else if (algorithm == "sentinel")
101         sentinel_sync(matrix_max, max_i, max_j, s, t,
102                       sim_matrix, threads);
103     else if (algorithm == "queue")
104         queue_sync(matrix_max, max_i, max_j, s, t,
105                    sim_matrix, threads);
106     else if (algorithm == "strip")
107         strip_partitions(matrix_max, max_i, max_j, s, t,
108                          sim_matrix, threads);
109
110     auto end_time = chrono::high_resolution_clock::now();
111     chrono::duration<double, micro> fp_micro = end_time - start_time;
112     print_matrix(s, t, sim_matrix);
113     print_results(matrix_max, max_i, max_j, fp_micro);
114     return 0;
115 }
116
117
118 /*******************************************************************************
119  * This helper function serves as the program implementation
120  * of the Smith-Waterman Local Sequence Alignment algorithm "max"
121  * operation where an element is calculated based on the calculated
122  * based on its "North", "West", and "Northwest" neighbors.
123  *
124  * @param s is the first string for local sequence alignment.
125  * @param t is the first string for local sequence alignment.
126  * @param i is the row index of the similarity matrix.
```

```
127  * @param j is the col index of the similarity matrix.
128  * @param sim_matrix is the 2D vector similarity matrix.
129  *
130  * @return is the maximum value for the specified matrix element.
131  ************************************************************/
132 int calc_element(string &s, string &t, int &i, int &j,
133                  vector<vector<int>> &sim_matrix) {
134     //busy wait for N and W
135     // (note: both N and W have a transitive dependency for NW so no
136     // need to check)
137     while (sim_matrix[i - 1][j] == -1 || sim_matrix[i][j - 1] == -1);
138
139     const int i_str_index = i - 1;
140     const int j_str_index = j - 1;
141     const int NORTH = i - 1;
142     const int WEST = j - 1;
143     const int GAP_PENALTY = -2;
144     const int MATCH = 1;
145     const int MISMATCH = -1;
146
147     int cell_max = 0;
148     //west
149     if ((sim_matrix[i][WEST] + GAP_PENALTY) > cell_max)
150         cell_max = sim_matrix[i][WEST] + GAP_PENALTY;
151     //north
152     if ((sim_matrix[NORTH][j] + GAP_PENALTY) > cell_max)
153         cell_max = sim_matrix[NORTH][j] + GAP_PENALTY;
154     //northwest +1 for match -1 if not (also check for '?' wildcards)
155     if ((s.at(i_str_index) == t.at(j_str_index) ||
156          (s.at(i_str_index) == '?') ||
157          (t.at(j_str_index) == '?')
158     )) {
159         if ((sim_matrix[NORTH][WEST] + MATCH) > cell_max)
160             cell_max = sim_matrix[NORTH][WEST] + MATCH;
161     } else {
162         if ((sim_matrix[NORTH][WEST] + MISMATCH) > cell_max)
163             cell_max = sim_matrix[NORTH][WEST] + MISMATCH;
164     }
165     return cell_max;
166 }
167
168
169 /*****************************************************************
170  * This helper function serves as the program implementation
171  * of the Smith-Waterman Local Sequence Alignment algorithm using
172  * a single-threaded "sequential" algorithm.
173  *
174  * @param matrix_max is the maximum element in the similarity matrix.
175  * @param max_i is the row index of the similarity matrix max.
176  * @param max_j is the col index of the similarity matrix max.
177  * @param s is the first string for local sequence alignment.
178  * @param t is the first string for local sequence alignment.
179  * @param sim_matrix is the 2D vector similarity matrix.
180  * @param threads is the number of parallel thread tasks to use.
181  ************************************************************/
182 void sequential(int &matrix_max, int &max_i, int &max_j, string &s,
183                 string &t, vector<vector<int>> &sim_matrix,
184                 int threads) {
185     for (int i = BASE; i < s.length() + BASE; i++) {
186         for (int j = BASE; j < t.length() + BASE; j++) {
187             sim_matrix[i][j] = calc_element(s, t, i, j, sim_matrix);
188             if (sim_matrix[i][j] >= matrix_max) {
189                 matrix_max = sim_matrix[i][j];
```

```
190                 max_i = i;
191                 max_j = j;
192             }
193         }
194     }
195 }
196 
197 
198 /*****************************************************************
199  * This helper function serves as the program implementation
200  * of the Smith-Waterman Local Sequence Alignment algorithm using
201  * a simple parallelization of the sequential algorithm.
202  *
203  * @param matrix_max is the maximum element in the similarity matrix.
204  * @param max_i is the row index of the similarity matrix max.
205  * @param max_j is the col index of the similarity matrix max.
206  * @param s is the first string for local sequence alignment.
207  * @param t is the first string for local sequence alignment.
208  * @param sim_matrix is the 2D vector similarity matrix.
209  * @param threads is the number of parallel thread tasks to use.
210  *****************************************************************/
211 void sentinel_sync(int &matrix_max, int &max_i, int &max_j, string &s,
212                    string &t, vector<vector<int>> &sim_matrix,
213                    int threads) {
214 
215     int i, j, cell_max;
216 #pragma omp parallel for schedule(dynamic) num_threads (threads) \
217 default(none) shared(s, t, sim_matrix, matrix_max, max_j, max_i) \
218 private( i, j, cell_max)
219     for (i = BASE; i < s.length() + BASE; i++) {
220         for (j = BASE; j < t.length() + BASE; j++) {
221             sim_matrix[i][j] = calc_element(s, t, i, j, sim_matrix);
222             if (sim_matrix[i][j] >= matrix_max) {
223 #pragma omp critical
224                 {
225                     //recheck in case max changed while we were waiting
226                     if (sim_matrix[i][j] >= matrix_max) {
227                         matrix_max = sim_matrix[i][j];
228                         max_i = i;
229                         max_j = j;
230                     }
231                 }
232             }
233         }
234     }
235 }
236 
237 
238 /*****************************************************************
239  * This helper function serves as the program implementation
240  * of the Smith-Waterman Local Sequence Alignment algorithm using
241  * a "job queue" parallelization algorithm.
242  *
243  * @param matrix_max is the maximum element in the similarity matrix.
244  * @param max_i is the row index of the similarity matrix max.
245  * @param max_j is the col index of the similarity matrix max.
246  * @param s is the first string for local sequence alignment.
247  * @param t is the first string for local sequence alignment.
248  * @param sim_matrix is the 2D vector similarity matrix.
249  * @param threads is the number of parallel thread tasks to use.
250  *****************************************************************/
251 void queue_sync(int &matrix_max, int &max_i, int &max_j, string &s,
252                 string &t, vector<vector<int>> &sim_matrix,
```

```
253                 int threads) {
254     int i, j;
255     queue<tuple<int, int>> job_q;
256     //fill job queue
257     for (i = BASE; i < s.length() + BASE; i++) {
258         for (j = BASE; j < t.length() + BASE; j++) {
259             job_q.push(make_tuple(i, j));
260         }
261     }
262     mutex q_mutex;
263     tuple<int, int> el;
264 #pragma omp parallel num_threads (threads) default(none) shared(s, t, \
265 sim_matrix, matrix_max, max_j, max_i, q_mutex, job_q) private(el, i, j)
266     {
267         while (true) {
268             {
269                 lock_guard<mutex> lock(q_mutex);
270                 if (!job_q.empty()) {
271                     el = job_q.front();
272                     job_q.pop();
273                 } else break;
274             }
275             i = get<0>(el);
276             j = get<1>(el);
277             sim_matrix[i][j] = calc_element(s, t, i, j, sim_matrix);
278             if (sim_matrix[i][j] >= matrix_max) {
279 #pragma omp critical
280                 {
281                     //recheck in case max changed while we were waiting
282                     if (sim_matrix[i][j] >= matrix_max) {
283                         matrix_max = sim_matrix[i][j];
284                         max_i = i;
285                         max_j = j;
286                     }
287                 }
288             }
289         }
290     }
291 }
292
293
294 /*****************************************************************************
295  * This helper function serves as the program implementation
296  * of the Smith-Waterman Local Sequence Alignment algorithm using
297  * a "strip partitions" parallelization algorithm that divides the
298  * similarity matrix calculations into strips based on dependencies.
299  *
300  * @param matrix_max is the maximum element in the similarity matrix.
301  * @param max_i is the row index of the similarity matrix max.
302  * @param max_j is the col index of the similarity matrix max.
303  * @param s is the first string for local sequence alignment.
304  * @param t is the first string for local sequence alignment.
305  * @param sim_matrix is the 2D vector similarity matrix.
306  * @param threads is the number of parallel thread tasks to use.
307  ****************************************************************************/
308 void strip_partitions(int &matrix_max, int &max_i, int &max_j,
309                       string &s, string &t,
310                       vector<vector<int>> &sim_matrix, int threads) {
311     //assuming t <= s; every character of t will be start a strip
312     //every element of s (except for the first one already handled
313     // by a horizontal strip) will be a vertical strip but the number
314     // of vertical strips in the matrix is also constrained by len of t
315     int total_strips = t.length() + min(s.length() - 1, t.length());
```

```
316     int strip_index = 0;
317     int i, j, k, l, cell_max;
318     //can't use omp critical because different blocks update max vars
319     mutex max_mutex;
320 #pragma omp parallel for schedule(dynamic) num_threads (threads) \
321 default(none) shared(total_strips, s, t, sim_matrix, matrix_max, \
322 max_j, max_i, max_mutex) private(strip_index, i, j, k, l, cell_max)
323     for (strip_index = 0; strip_index < total_strips; strip_index++) {
324         // even strips = compute a row
325         if (strip_index % 2 == 0) {
326             for (j = strip_index / 2; j < t.length(); j++) {
327                 //use k and l as local i and j to avoid recalc
328                 k = BASE + (strip_index / 2);
329                 l = BASE + j;
330                 cell_max = calc_element(s, t, k, l, sim_matrix);
331                 sim_matrix[k][l] = cell_max;
332                 if (cell_max >= matrix_max) {
333                     {
334                         lock_guard<mutex> lock(max_mutex);
335                         // check in case max has changed while waiting
336                         if (cell_max >= matrix_max) {
337                             matrix_max = cell_max;
338                             max_i = k;
339                             max_j = l;
340                         }
341                     }
342                 }
343             }
344         }
345             //odd strips = compute a col
346         else {
347             for (i = (strip_index / 2) + 1; i < s.length(); i++) {
348                 //use k and l as local i and j to avoid recalculating
349                 k = BASE + i;
350                 l = BASE + (strip_index / 2);
351                 cell_max = calc_element(s, t, k, l, sim_matrix);
352                 sim_matrix[k][l] = cell_max;
353                 if (cell_max >= matrix_max) {
354                     lock_guard<mutex> lock(max_mutex);
355                     // check in case max has changed while waiting
356                     if (cell_max >= matrix_max) {
357                         matrix_max = cell_max;
358                         max_i = k;
359                         max_j = l;
360                     }
361                 }
362             }
363         }
364     }
365 }
366
367
368 /*****************************************************************
369  * This helper function prints the similarity matrix representation
370  * of strings s and t.
371  *
372  * @param s is the first string for local sequence alignment.
373  * @param t is the first string for local sequence alignment.
374  * @param sim_matrix is the 2D vector similarity matrix.
375  *****************************************************************/
376 void print_matrix(string &s, string &t,
377                   const vector<vector<int>> &sim_matrix) {
378     for (int k = 0; k < t.length() + 1; k++) {
```

```
379         if (k == 0) cout << "    ";
380         else cout << t.at(k - 1) << " ";
381     }
382     cout << endl;
383     for (int i = 0; i < s.length() + 1; i++) {
384         if (i == 0)
385             cout << ' ';
386         else
387             cout << s.at(i - 1);
388         for (int j = 0; j < t.length() + 1; j++) {
389             if (sim_matrix[i][j] != -1)
390                 cout << " " << sim_matrix[i][j];
391             else cout << sim_matrix[i][j];
392         }
393         cout << endl;
394     }
395 }
396
397
398 /*****************************************************************
399  * This helper function prints the SW algorithm results.
400  *
401  * @param matrix_max is the maximum element in the similarity matrix.
402  * @param max_i is the row index of the similarity matrix max.
403  * @param max_j is the col index of the similarity matrix max.
404  * @param micro_secs is the time to calculate the matrix.
405  *****************************************************************/
406 void print_results(int matrix_max, int max_i, int max_j,
407                    chrono::duration<double, micro> micro_secs) {
408     cout << "Time: " << micro_secs.count() << " microseconds" << endl;
409     cout << "max = " << matrix_max << ", ";
410     cout << "i = " << max_i << ", ";
411     cout << "j = " << max_j << endl;
412 }
413
414 /*****************************************************************
415  * This helper function opens the file specified by the parameter
416  * string and then returns a "cleaned" version where all non-alphabetic
417  * characters have been removed.
418  *
419  * @param in_file is the name if the input data source file.
420  * @return is the string representing the "cleaned" data.
421  *****************************************************************/
422 string parse_input(const string &in_file) {
423     string temp_line;
424     string cleaned_input;
425     ifstream in(in_file);
426
427     //parse the input (assumes only A,T,C,G,?)
428     char c;
429     if (in.is_open()) {
430         while ((in.get(c))) {
431             switch (c) {
432                 case 'A':
433                 case 'T':
434                 case 'C':
435                 case 'G':
436                 case '?':
437                     cleaned_input.append(1, c);
438                     break;
439                 case ' ':
440                 case '\t':
441                 case '\n':
```

```
442                 case '\r':
443                     //do nothing to exclude whitespace
444                     break;
445                 default:
446                     cerr << "Failed to parse input due to character "
447                          << c << endl;
448             }
449         }
450     } else {
451         cerr << "Failed to open file for parsing " << endl;
452         exit(1);
453     }
454     in.close();
455     return cleaned_input;
456 }
457 
458 /*************************************************************************
459  * This helper function returns memory snapshots of the program during
460  * execution.
461  * I modified my code based on the concept and sample at:
462  * https://hpcf.umbc.edu/general-productivity/checking-memory-usage/
463  *
464  * @return is the number of KBs in use.
465  ************************************************************************/
466 //Note: this value is in KB!
467 int get_memory_snapshot() {
468     ifstream in;
469     string temp_line;
470     in.open(
471             "/proc/self/status");
472     if (in.is_open()) {
473         while (getline(in, temp_line, '\n')) {
474             if (temp_line.substr(0, 7) == "VmSize:") {
475                 in.close();
476                 return stoi(
477                         temp_line.substr(7, temp_line.length() - 7));
478             }
479         }
480     }
481     in.close();
482     cerr << "Memory Snapshot Failed" << endl;
483     exit(1);
484 }
485 
```

Appendix C - Sample Strip Assignments for HIV1 and HIV Polymerase

(-1 represents the default value when the 2D matrix was allocated; the other numbers represent threads reporting their strip assignments of partial rows or columns)

```
    T T T T T T A G G G A A A A T T T G G C C T T C C A ? C A A G G G G A G G C C A G G ? A A T T T
 -1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
G-1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
G-1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
T-1 1 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
C-1 1 3 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
T-1 1 3 5 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
C-1 1 3 5 7 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
T-1 1 3 5 7 1 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
C-1 1 3 5 7 1 3 5 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
.
.
.
C-1 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7
T-1 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7
T-1 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7
C-1 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7
A-1 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7 1 3 5 7
```

Appendix D - Manually Computed Similarity Matrix "Test Bench"

Sequence (*s*): G A T A T G C A
Unknown (*t*): A T A G C T

		A	T	A	G	C	T
	0	0	0	0	0	0	0
G	0	0	0	0	1	0	0
A	0	1	0	1	0	0	0
T	0	0	2	0	0	0	1
A	0	1	0	3	1	0	0
T	0	0	2	1	2	0	1
G	0	0	0	1	2	1	0
C	0	0	0	0	0	3	1
A	0	1	0	1	0	1	2

max = 3, i = 7, j = 5

s:	G	A	T	A	T	G	C	A
t:		A	T	A	-	G	C	
		1	1	1	-2	1	1	

$\Sigma = 3$

Sequence (s): A A T C G G A T T C
Unknown (t): G A C G G T T C G T

		G	A	C	G	G	T	T	C	G	T
	0	0	0	0	0	0	0	0	0	0	0
A	0	0	1	0	0	0	0	0	0	0	0
A	0	0	1	0	0	0	0	0	0	0	0
T	0	0	0	0	0	0	1	1	0	0	1
C	0	0	0	1	0	0	0	0	2	0	0
G	0	1	0	0	2	1	0	0	0	3	1
G	0	1	0	0	1	3	1	0	0	1	2
A	0	0	2	0	0	1	2	0	0	0	0
T	0	0	0	1	0	0	2	3	1	0	1
T	0	0	0	0	0	0	1	3	2	0	1
C	0	0	0	1	0	0	0	1	4	2	0

max = 4, i = 10, j = 8

Appendix E - Sequential Program Output

(The Sequential Program Output matches the manual computations in Appendix D)

```
  ATAGCT
 0000000
G0000100
A0101000
T0020001
A0103100
T0021201
G0001210
C0000031
A0101012


  GACGGTTCGT
 00000000000
A00100000000
A00100000000
T00000011001
C00010000200
G01002100031
G01001310012
A00200120000
T00010023101
T00000013201
C00010001420
```

Appendix F - Sample Validation “Test Bench” BASH Script

(Ensure the output of the queue implementation produces the same output as the sequential program regardless of the number of threads; ignore differences in time which are expected to vary)

```
 1 g++ -Wall -fopenmp main.cpp -o SW
 2 if [ $? -eq 0 ]
 3 then
 4   clear;
 5   for i in {1..8}
 6         do
 7                 ./SW /home/westoand/hpc/project2/HIV-1_db.fasta /home/westoand/hpc/project2/-
   HIV-1_Polymerase.txt "$i" queue > queue.txt
 8                 diff -I '^Time*' sequential.txt queue.txt
 9         done
10 fi
```

Appendix G - Sample BASH Script for Parallel Algorithm Experiments

(10 trials with varying thread counts; filter output to only time results)

```
g++ -Wall -fopenmp main.cpp -o SW
if [ $? -eq 0 ]
then
  clear;
  for i in {1..10}
        do
                for j in {2..8}
                        do
                                ./SW /home/westoand/hpc/project2/HIV-1_db.fasta /home/westoand/hpc/-
project2/HIV-1_Polymerase.txt $j strip | grep "Time" >> performance_strip$j.txt
                        done
        done
fi
```

Appendix H - Performance Data by Solution

Sequential

Trial	Time (microseconds)
1	20050.1
2	21436.6
3	19423.2
4	20279.5
5	20878.3
6	19354.8
7	24427.3
8	19372
9	19432.6
10	19385.8
Average	**20404.02**

Sentinel-Synchronization

Threads	2	3	4	5	6	7	8	
Trial	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	
1	15621.1	10986.6	10268.2	9930.92	7663.76	13764.4	8366.24	
2	12200.3	9813.05	10309.3	8227.13	7157.67	6314.8	6160.31	
3	12454.4	12815.9	10015.5	9051.66	7737.86	6920.44	11261.8	
4	12070	9493.21	9972.32	8188.76	7179.2	6631.67	26957.4	
5	13952.4	9484.36	9927.15	8682.58	7432.59	6711.85	19430.4	
6	12314.9	9470.07	10241.8	8575.82	7790.99	6824.06	8201.9	
7	12161	10385	9983.61	8400.93	7526.77	6815.13	9212.17	
8	12416.9	9748.44	10162.6	8371.61	7323.3	6906.03	6664.99	
9	12030.9	9600.14	9963.31	8707.23	7138.95	6951.71	6283.09	
10	12069	9652.68	10115.8	8420.15	7142.62	6293.9	13912.4	Min
Average (microsec)	**12729.09**	**10144.945**	**10095.959**	**8655.679**	**7409.371**	**7413.399**	**11645.07**	**7409.371**

Queue-Synchronization

Threads	2	3	4	5	6	7	8	
Trial	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	
1	142391	175385	170664	217527	257622	250480	302937	
2	174204	161889	157222	231431	257757	250027	276156	
3	179784	162654	179241	231047	247429	261761	255916	
4	174017	164053	207682	227048	256158	254750	261757	
5	160409	160705	160752	224863	253158	247816	372571	
6	179851	169211	162138	243406	251861	252175	239052	
7	177402	163998	161254	235138	233426	251944	256318	
8	170467	178637	169192	219226	241051	250853	258233	
9	149267	160282	177138	221157	247622	241388	254988	
10	169250	165478	163184	238094	256625	257614	261540	Min
Average (microsec)	**167704.2**	**166229.2**	**170846.7**	**228893.7**	**250270.9**	**251880.8**	**273946.8**	**166229.2**

Strip-Partition Synchronization

Threads	2	3	4	5	6	7	8	
Trial	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	Time (microseconds)	
1	10021.7	10303.2	8207.16	7402.6	6163.02	5705.87	5360.46	
2	10527.5	7574.08	8389.35	6947.28	6502.97	5538.88	7010.68	
3	10068.7	10592.8	8458.3	7229.69	6222.83	5560.38	6421.62	
4	10403.8	6880.34	8474.62	7006.41	6326.67	5604.56	7518.43	
5	10280.4	7466.47	8139.6	6350.47	6352.55	8076.18	12214.8	
6	10450.4	7862.5	5909.1	7213.2	6188.98	5506.73	5061.28	
7	10063.4	6910.24	7978.77	7293.31	6390.11	7413.19	6824.6	
8	10779.4	11537.7	8585.55	7158.16	6470.71	5554.19	7065.58	
9	10291.5	7134.71	8282.39	6772.9	6494.23	7417.26	9017.77	
10	10659.4	6932.77	8136.38	6992.83	6159.21	10939.8	5754.42	Min
Average	**10354.62**	**8319.481**	**8056.122**	**7036.685**	**6327.128**	**6731.704**	**7224.964**	**6327.128**

Appendix I - SW C++ Piecemeal Similarity Matrix - Enhancement Draft Implementation

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <vector>
 4 #include <string>
 5 #include <fstream>
 6 #include <algorithm>
 7 #include <chrono>
 8 #include <omp.h>
 9 #include <queue>
10 #include <mutex>
11
12 using namespace std;
13
14 /* Function to parse input file into compatible string */
15 string parse_input(const string &in_file);
16
17 /* Helper function to print similarity matrix state */
18 void print_matrix(string &s, string &t,
19                   const vector<vector<int>> &sim_matrix);
20
21 /* Helper function to print similarity matrix results */
22 void print_results(
23         chrono::duration<double, micro> micro_secs);
24
25 /* Helper function to simulate a long base sequence */
26 void generate_long_sequence();
27
28 /* Helper function to write-out partial matrix */
29 void write_out_helper(string &s, string &t, vector<vector<int>> &sim_matrix,
30                       int threads);
31
32 /* Helper function to perform the max operation from SW algorithm */
33 int calc_element(string &s, string &t, int i, int i_factor, int j,
34                  vector<vector<int>> &sim_matrix);
35
36 /* Multi-threaded, implementation of SW algorithm using "incomplete"
37  * sentinels */
38 void parallel_piecemeal(string &s,
39                         string &t, vector<vector<int>> &sim_matrix,
40                         int threads);
41
42
43 /* The SW algorithm assigns 0 to the first row and column making the
44  * first calculated element at index (1,1)*/
45 const int BASE = 1;
46
47
48 /*********************************************************************
49  * This function serves as the program "driver" for an extension
50  * of the Smith-Waterman Local Sequence Alignment assignment using
51  * OpenMP. It reads in a genetic sequence file, builds a
52  * "similarity matrix" in a piecemeal fashion and writes results
53  * to disk.
54  *
55  * @param argc is the number of command-line arguments.
56  * @param v is the array of strings representing the command-line args.
57  *********************************************************************/
58 int main(int argc, char **argv) {
59     if (argc != 4) {
60         cerr << "Invalid Use\n Syntax is: file_name_str_s ";
61         cerr << "file_name_str_t num_threads algorithm" << endl;
62         cerr << "Algorithms are sequential sentinel queue or strip"
63              << endl;
```

```
 64     }
 65
 66     string s = parse_input(argv[1]);
 67     string t = parse_input(argv[2]);
 68     int threads = stoi(argv[3]);
 69
 70     vector<vector<int>> sim_matrix(threads + BASE,
 71                                    vector<int>(t.length() + BASE, -1));
 72     for (int i = 0; i < threads + BASE; i++) {
 73         sim_matrix[i][0] = 0;
 74     }
 75     for (int j = 0; j < t.length() + 1; j++) {
 76         sim_matrix[0][j] = 0;
 77     }
 78
 79     auto start_time = chrono::high_resolution_clock::now();
 80
 81     parallel_piecemeal(s, t,
 82                        sim_matrix, threads);
 83
 84     auto end_time = chrono::high_resolution_clock::now();
 85     chrono::duration<double, micro> fp_micro = end_time - start_time;;
 86     return 0;
 87 }
 88
 89
 90 /*****************************************************************
 91  * This helper function serves as the program implementation
 92  * of the Smith-Waterman Local Sequence Alignment algorithm "max"
 93  * operation where an element is calculated based on the calculated
 94  * based on its "North", "West", and "Northwest" neighbors.
 95  *
 96  * @param s is the first string for local sequence alignment.
 97  * @param t is the first string for local sequence alignment.
 98  * @param i is the row index of the similarity matrix.
 99  * @param j is the col index of the similarity matrix.
100  * @param sim_matrix is the 2D vector similarity matrix.
101  *
102  * @return is the maximum value for the specified matrix element.
103  *****************************************************************/
104 int calc_element(string &s, string &t, int i, int i_factor, int j,
105                  vector<vector<int>> &sim_matrix) {
106     //busy wait for N and W
107     // (note: both N and W have a transitive dependency for NW so no
108     // need to check)
109     while (sim_matrix[i - 1][j] == -1 || sim_matrix[i][j - 1] == -1);
110
111     const int i_str_index = i - 1 + i_factor;
112     const int j_str_index = j - 1;
113     const int NORTH = i - 1;
114     const int WEST = j - 1;
115     const int GAP_PENALTY = -2;
116     const int MATCH = 1;
117     const int MISMATCH = -1;
118
119     int cell_max = 0;
120     //west
121     if ((sim_matrix[i][WEST] + GAP_PENALTY) > cell_max)
122         cell_max = sim_matrix[i][WEST] + GAP_PENALTY;
123     //north
124     if ((sim_matrix[NORTH][j] + GAP_PENALTY) > cell_max)
125         cell_max = sim_matrix[NORTH][j] + GAP_PENALTY;
126     //northwest +1 for match -1 if not (also check for '?' wildcards)
```

```
127     if ((s.at(i_str_index) == t.at(j_str_index) ||
128          (s.at(i_str_index) == '?') ||
129          (t.at(j_str_index) == '?')
130     )) {
131         if ((sim_matrix[NORTH][WEST] + MATCH) > cell_max)
132             cell_max = sim_matrix[NORTH][WEST] + MATCH;
133     } else {
134         if ((sim_matrix[NORTH][WEST] + MISMATCH) > cell_max)
135             cell_max = sim_matrix[NORTH][WEST] + MISMATCH;
136     }
137     return cell_max;
138 }
139 
140 
141 /*********************************************************************
142  * This helper function serves as the program implementation
143  * of the Smith-Waterman Local Sequence Alignment algorithm using
144  * a simple parallelization of the sequential algorithm.
145  *
146  * @param matrix_max is the maximum element in the similarity matrix.
147  * @param max_i is the row index of the similarity matrix max.
148  * @param max_j is the col index of the similarity matrix max.
149  * @param s is the first string for local sequence alignment.
150  * @param t is the first string for local sequence alignment.
151  * @param sim_matrix is the 2D vector similarity matrix.
152  * @param threads is the number of parallel thread tasks to use.
153  *********************************************************************/
154 void parallel_piecemeal(string &s, string &t,
155                         vector<vector<int>> &sim_matrix,
156                         int threads) {
157     int i, j;
158     for (int sub_matrix = 0; sub_matrix < (s.length() / threads) + 1;
159          sub_matrix++) {
160 #pragma omp parallel for schedule(dynamic) num_threads (threads) \
161 default(none) shared(s, t, sim_matrix, threads, sub_matrix) \
162 private( i, j)
163         for (i = BASE; i < threads + BASE; i++) {
164             for (j = BASE; j < t.length() + BASE; j++) {
165                 sim_matrix[i][j] =
166                         calc_element(s, t, i,
167                                      sub_matrix * threads,
168                                      j, sim_matrix);
169             }
170         }
171         write_out_helper(s, t, sim_matrix, threads);
172     }
173 }
174 
175 void write_out_helper(string &s, string &t, vector<vector<int>> &sim_matrix,
176                       int threads) {
177     int i, j;
178     for (i = BASE; i < threads + BASE; i++) {
179         for (j = BASE; j < t.length() + BASE; j++) {
180             cout << sim_matrix[i][j] << " ";
181         }
182         cout << endl;
183     }
184     for (i = 0; i < threads + BASE; i++) {
185         for (j = BASE; j < t.length() + BASE; j++) {
186             // cp bottom row to top
187             if (i == 0 && j > 0) {
188                 sim_matrix[i][j] = sim_matrix[threads + BASE - 1][j];
189             } else {
```

```
190                 //reset # threads rows
191                 sim_matrix[i][j] = -1;
192             }
193         }
194     }
195 }
196 
197 
198 /*****************************************************************
199  * This helper function prints the similarity matrix representation
200  * of strings s and t.
201  *
202  * @param s is the first string for local sequence alignment.
203  * @param t is the first string for local sequence alignment.
204  * @param sim_matrix is the 2D vector similarity matrix.
205  *****************************************************************/
206 void print_matrix(string &s, string &t,
207                   const vector<vector<int>> &sim_matrix) {
208     for (int k = 0; k < t.length() + 1; k++) {
209         if (k == 0) cout << "    ";
210         else cout << t.at(k - 1) << " ";
211     }
212     cout << endl;
213     for (int i = 0; i < s.length() + 1; i++) {
214         if (i == 0)
215             cout << ' ';
216         else
217             cout << s.at(i - 1);
218         for (int j = 0; j < t.length() + 1; j++) {
219             if (sim_matrix[i][j] != -1)
220                 cout << " " << sim_matrix[i][j];
221             else cout << sim_matrix[i][j];
222         }
223         cout << endl;
224     }
225 }
226 
227 
228 /*****************************************************************
229  * This helper function prints the SW algorithm results.
230  *
231  * @param matrix_max is the maximum element in the similarity matrix.
232  * @param max_i is the row index of the similarity matrix max.
233  * @param max_j is the col index of the similarity matrix max.
234  * @param micro_secs is the time to calculate the matrix.
235  *****************************************************************/
236 void print_results(int matrix_max, int max_i, int max_j,
237                    chrono::duration<double, micro> micro_secs) {
238     cout << "Time: " << micro_secs.count() << " microseconds" << endl;
239 }
240 
241 /*****************************************************************
242  * This helper function opens the file specified by the parameter
243  * string and then returns a "cleaned" version where all non-alphabetic
244  * characters have been removed.
245  *
246  * @param in_file is the name if the input data source file.
247  * @return is the string representing the "cleaned" data.
248  *****************************************************************/
249 string parse_input(const string &in_file) {
250     string temp_line;
251     string cleaned_input;
252     ifstream in(in_file);
```

```
253
254     //parse the input (assumes only A,T,C,G,?)
255     char c;
256     if (in.is_open()) {
257         while ((in.get(c))) {
258             switch (c) {
259                 case 'A':
260                 case 'T':
261                 case 'C':
262                 case 'G':
263                 case '?':
264                     cleaned_input.append(1, c);
265                     break;
266                 case ' ':
267                 case '\t':
268                 case '\n':
269                 case '\r':
270                     //do nothing to exclude whitespace
271                     break;
272                 default:
273                     cerr << "Failed to parse input due to character "
274                          << c << endl;
275             }
276         }
277     } else {
278         cerr << "Failed to open file for parsing " << endl;
279         exit(1);
280     }
281     in.close();
282     return cleaned_input;
283 }
284
285
286 /*********************************************************************
287  * This helper function generates a string simulating the longest
288  * known gene sequence.
289  *********************************************************************/
290 void generate_long_sequence() {
291     const int LONGEST_GENE_SEQ = 2600000;
292     srand(time(NULL));
293     for (int i = 0; i < LONGEST_GENE_SEQ; i++) {
294         int base = rand();
295         if (base % 4 == 0)
296             cout << "A";
297         else if (base % 4 == 1)
298             cout << "T";
299         else if (base % 4 == 2)
300             cout << "C";
301         else cout << "G";
302     }
303 }
304
```