Sequence Alignment and Parallel Programming

1st Daniel Vail 2nd Austin Traub 3rd Tony Pham

***Abstract*—...**

I. Introduction

The purpose of this project is to explore the algorithms and related parallel programming techniques used in sequence alignment. Sequence alignment is a process in bioinformatics whereby DNA, RNA, or protein is aligned in way that produces an optimal match between the residues of the molecules being aligned. DNA and RNA are made of nucleotides, and proteins are made of amino acids. Nucleotides and amino acids are represented as letters, and it is the sequences of those letters that are being aligned.

Sequence alignment is typically used to study the functions of biological sequences. Similarity between two biological sequences usually indicate that those sequences perform the same function. Thereby, sequence alignment is a highly useful process when it comes to studying this.

There are two main types of sequence alignments: pairwise alignment and multiple sequence alignment [1]. In a pairwise alignment, two sequences are compared and aligned. In a multiple sequence alignment, multiple sequences are aligned. This project will focus on algorithms for pairwise alignment. The typical application of pairwise alignment is to allow users to search a database for sequences similar to a specified input sequence. The input sequence is compared pairwise to each sequence in the database and the best matches are returned.

Alignments can be categorized further as either global or local. A global alignment is one that finds the best alignment for the entire sequence from beginning to end [2]. This is useful for comparing closely related sequences. A local alignment is one that finds similarities in only local regions of the sequences [2]. This is useful for when maybe only certain regions of the sequences being compared are similar. The project will look at algorithms for both global and local alignments.

What makes the sequence alignment problem difficult is the size of the sequences usually being compared. The typical size of a DNA sequence string in FASTA format, for example, can range from 40KB to 60KB [3]. Using a sequence alignment algorithm such as the Needleman-Wunsch Algorithm, which runs in O(MN), can take some time. In addition, doing this many times over while searching a database makes this process impractical. That is why it is important to look into ways in which sequence alignment could be parallelized.

The following is an example sequence alignment for the sequences TCGATA and AGATC:

TCGATA
–AGATC

The dash before the letter A in the second sequence in the alignment represents a gap. Gaps are indicative of insertions or deletions. Sequences may have been the same at some point in time and ended up diverging through mutations involving substitutions, insertions, or deletions [4]. For example, the second sequence above could have undergone a mutation where T (the first letter) was deleted and C (the second letter) was substituted with an A.

Another possible alignment for the sequences above is as follows:

TCGATA
A–GATC

This example indicates that, in the second sequence, a mutation could have caused T (the first letter) to be substituted with A and C (the second letter) to be deleted. To determine which alignment is best, algorithms use a scoring system to assign a score to each possible alignment. Usually, the alignment with the highest score is the optimal alignment.

The existing algorithms for sequence alignment fall into three categories: dynamic programming based, heuristic-based, or a mixture of the two [1]. The dynamic programming based algorithms breaks the problem into subproblems and finds a solution using the bottom up approach. In order words, the optimal alignments for the subsequences are found, and those are used to find optimal alignment for the overall sequences. The dynamic programming based solutions are computationally demanding, making them impractical when dealing with a great number of sequences [1]. As a result, heuristic approaches are typically used to approximate the optimal alignment. These heuristic solutions typically use dynamic programming for only a portion of the sequence while using approximations to make the search space more manageable. The result is a faster and more efficient algorithm.

The focus of the project is to only explore existing sequential and concurrent algorithms for sequence alignment based on dynamic programming.

II. Goals

As stated before, the goal of this project is the explore existing sequential and concurrent algorithms for sequence alignment based on dynamic programming. We will attempt to implement these algorithms and improve the parallelization of them through experimentation.

III. Existing Algorithms

The following section presents some dynamic programming based algorithms for sequence alignment. The Needleman-Wunsch algorithm is used to find global alignments. The Smith-Waterman algorithm is used to find local alignments. And the Hirschberg algorithm is a space efficient modification of the Needleman-Wunsch algorithm.

A. Needleman-Wunsch

The Needleman-Wunsch algorithm uses dynamic programming to find the optimal global alignment between two sequences [5]. It finds the optimal alignment across the entire sequence. The algorithm is based on the idea that the overall optimal alignment of two sequences can be determined through determining the optimal alignments of their subsequences and building up from there. All possible alignments of the two sequences are determined, and a score is given to each to rank it. The alignment with the highest score is the most optimal one.

The algorithm involves the use of two matrices: the score matrix and the traceback matrix. It can be broken down into three main parts: the initialization of the two matrices, the calculating of the scores and directions in the two matrices, and using the traceback matrix to determine the optimal alignment.

The score matrix is used to store the score for each alignment. The traceback matrix is used to store the direction from which each score came from. Each score in the score matrix is calculated based on a previously calculated neighboring score. The directions in the traceback matrix are used to trace and determine the highest scoring alignment. Fig. 1 is an example score matrix, and Fig. 2 is its corresponding traceback matrix.

		S	E	N	D
	0	−10	−20	−30	−40
A	−10	1	−9	−19	−29
N	−20	−9	−1	−3	−13
D	−30	−19	−11	2	3

Fig. 1. Example score matrix. [5]

A scoring scheme needs to be picked to help calculate the scores. A basic example scheme is to use +1 for a match, -1 for a mismatch, and 0 for a gap. A match occurs when two letters match. A mismatch is when the letters do not match. And a gap is where the letter from one sequence is lined up with a gap, corresponding to an insertion/deletion. Inserting a gap into a sequence shifts it, causing a letter from the other sequence to match with the gap.

	S	E	N	D
done	left	left	left	left
up	diag	left	left	left
up	diag	diag	diag	left
up	up	diag	diag	diag

Fig. 2. Example traceback matrix. [5]

A substitution matrix can also be used to define the scoring scheme more thoroughly. Using a substitution matrix, different matches/mismatches could be assigned different substitution scores. For example, a C-C match could be assigned a +1 while a T-T match could be assigned a +2. Below in Fig. 3 is an example substitution matrix. Here, all the matches are assigned +1 while all the mismatches are assigned -1. For example, the value 1 in row 2 column 2 indicates that a C-C match should have a substitution score of 1.

	C	T	A	G
C	1	-1	-1	-1
T	-1	1	-1	-1
A	-1	-1	1	-1
G	-1	-1	-1	1

Fig. 3. Example substitution matrix. [5]

To determine the score for a cell in the score matrix, the scores of the neighboring cells to the left, above, or on the top left of the current cell are taken into account. The subscore of the current cell, determined by the scoring scheme, is added to the score of each of those neighboring cells individually. The highest result out of the three is chosen as the score for the current cell. Going to the cell on the left means adding a gap to the letter on the sequence placed on the left side of the matrix. The gap penalty should be added here. Going to the cell above means adding a gap to the letter on the sequence at the top of the matrix. The gap penalty should be added

here in this case. Going diagonally to the top left means that a gap was not added. Here, the match score should be added for a match, and the mismatch score should be added for a mismatch. The scoring system is represented by the recursive relation in Fig. 4, where *s(i, j)* is the substitution score and *g* is the gap penalty. Fig. 5 demonstrates this relation visually on a score matrix.

$$D(i,j) = \max \begin{cases} D(i-1,j-1) + s(x_i, y_j) \\ D(i-1,j) + g \\ D(i,j-1) + g \end{cases}$$

Fig. 4. Recursive relation to determine the score. [5]

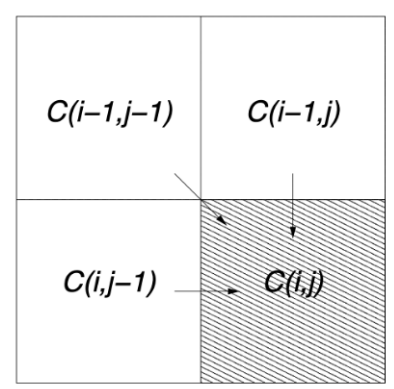

Fig. 5. Determining the score on the matrix. [5]

The directions in the traceback matrix are either Left, Up, or Diagonal (for top left). The direction for each cell depends on which of the three neighboring cells was used to determine the final score for it. For example, if the final and highest score for a cell in a score matrix was calculated using the cell above it, the corresponding cell in the traceback matrix would have the value U for Up.

To demonstrate, in the score matrix below in Fig. 6, the score for row 2 column 2 (A-A) is determined to be 2. This matrix uses -2 for the gap penalty, -3 for a mismatch, and 2 for a match. Starting at row 2 column 2, going left to row 2 column 1 means adding a gap to the sequence on the left side of the matrix. The gap penalty is -2, so the total score for going left is the sum of the left cell and the gap penalty. That is, -2 + -2 = -4. Similarly, going up to row 1 column 2 means adding a gap to the sequence at the top of the matrix. The total score here is also -2 + -2 = -4. Lastly, going diagonally to row 1 column 1 means not adding a gap. The current cell is a match since it represents the pair A-A; both the letters corresponding to the row and column of the cell are A's. The score for a match is 2. Accordingly, the total score here is 0 + 2 = 2. Out of the three scores calculated, 2 is the highest, so it is chosen as the final score for the cell. Since the max score is calculated using the top left diagonal neighboring cell's score, we set the corresponding cell's value in the traceback matrix to D for diagonal. That is, we set the value of row 2 column 2 of the traceback matrix to D.

		A	C	T	G	A	T	T	C	A
	0	-2	-4	-6	-8	-10	-12	-14	-16	-18
A	-2	2	0	-2	-4	-6	-8	-10	-12	-14
C	-4	0	4	2	0	-2	-4	-6	-8	-10
G	-6	-2	2	1	4	2	0	-2	-4	-6
C	-8	-4	0	-1	2	1	-1	-3	0	-2
A	-10	-6	-2	-3	0	4	2	0	-2	2
T	-12	-8	-4	0	-2	2	6	4	2	0
C	-14	-10	-6	-2	-4	0	4	2	6	4
A	-16	-12	-8	-4	-5	-2	2	1	4	8

Fig. 6. Another example score matrix. [6]

The scoring matrix is initialized by setting the top uppermost leftmost cell to 0. The gap penalty is then added incrementally both along the first row and first column. For example, in Fig. 6, where the gap penalty is -2, the scores along the first row are 0, -2, -4, -6, -8, and so on. The scores along the first column are the same. These scores indicate the overall score for appending gaps to the beginning of each sequence. For instance, adding two gaps to the beginning of the left sequence above produces --ACGCATCA. The corresponding score for this alignment is -4 since row 3 column 1 is -4.

Once all of the scores in the score matrix and the directions in the traceback matrix have been determined, the traceback matrix is used to produce the most optimal alignment between the two sequences. The traceback begins at the bottommost rightmost cell. The trace then moves either left, up, or diagonally from that point on. Movement is based on the direction stored in the traceback matrix. Moving left means adding a gap to the sequence on the left of the matrix. Moving up means adding a gap to the sequence at the top of the matrix. And moving diagonally means that the two letters for that cell are aligned and no gap is to be added. The traceback ends when the topmost leftmost cell is reached. Fig. 7 shows an example traceback matrix and its final path. The corresponding optimal alignment for the sequences SEND and AND is as follows:

SEND
A–ND

B. Parallel Version of Needleman-Wunsch

Naveed, Siddiqui, and Ahmed describes a method for parallelizing the Needleman-Wunsch algorithm [3]. This method involves two parts: the parallel initialization of the matrices used in the algorithm and the parallel calculation of the scores and directions that fill the matrices.

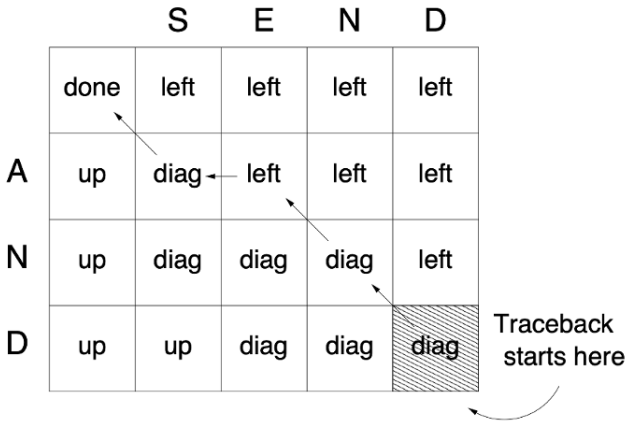

Fig. 7. Another example traceback matrix. [5]

The score and traceback matrices each can be initialized by two separate threads. One thread can set the initial row column values (-1, -2, -3, … or Left) while the other thread can set the initial first column values (-1, -2, -3, … or Up). If the sequences are stored in the matrices as headers, they also can be stored in a similar parallel manner. One thread sets the first sequence across the top and the other sets the second sequence along the side.

As for the calculating the scores, the scores can be calculated anti-diagonally. This way, the calculations can be parallelized within each anti-diagonal. The scores in the first anti-diagonal are calculated first, then the second, the third, and so on. Fig. 8 below shows the anti-diagonals used in this technique. Calculating the scores in an anti-diagonal order is possible since each score is determined based on the cells left, above, or top left of it. The scores for the left, top, and top left cells for each cell will be already determined by the time we attempt to fill in the next anti-diagonal.

In Fig. 8 below, row 2 column 2 would be calculated first. Then row 2 column 3 and row 3 column 2 would be calculated next in parallel. Here, one thread can be used to calculate row 2 column 3 and another can be used to calculate row 3 column 2. The next calculations are row 2 column 4, row 3 column 3, and row 4 column 2 together in parallel. This pattern continues until the matrix is completely filled in.

The limitation of this method is that the parallelization can be done only within each anti-diagonal. To start filling in the next anti-diagonal, the preceding anti-diagonal must already be all completed. Otherwise, there would not be enough information to calculate all of the scores for that anti-diagonal. The scores in each anti-diagonal is dependent on the scores in the preceding anti-diagonal.

According to the authors, the parallel version of the Needleman-Wunsch algorithm has a runtime of O(N+M) [3]. This is an improvement over the O(MN) runtime of the original algorithm.

DataArray

		A	C	G	T	A	A	G	T
	0	-1	-2	-3	-4	-5	-6	-7	-8
T	-1	X	X	X	X	X	X	X	X
G	-2	X	X	X	X	X	X	X	X
C	-3	X	X	X	X	X	X	X	X
C	-4	X	X	X	X	X	X	X	X
A	-5	X	X	X	X	X	X	X	X
G	-6	X	X	X	X	X	X	X	X
T	-7	X	X	X	X	X	X	X	X
G	-8	X	X	X	X	X	X	X	X

Fig. 8. Sections that can be calculated in parallel. [3]

C. Hirschberg

Created in the mid 1970's by Dan Hirschberg, Hirschberg's algorithm is an improvement to the Needleman-Wunsch algorithm created at the beginning of the decade [7]. The goal of the algorithm is to find the optimal sequence alignment of two strings [7]. This is particularly useful in DNA and protein sequencing in the field of bioinformatics/computational biology. Hirschberg's improvement on the Needleman-Wunsch algorithm comes from it being more spatially efficient. As stated in the above section, the runtime efficiency of Needleman-Wunsch is O(nm), where n and m are the lengths of the two strings respectively, which is the same as Hirschberg's. The difference is that Needleman-Wunsch uses O(nm) space, since it creates an n by m matrix, where n and m are the lengths of the sequences, whereas Hirschberg's algorithm only takes O(m) space by using binary recursion [8].

IV. Experiments

This section describes in detail our attempts to parallelize the algorithms described above. All of our experiments are coded using Java.

A. Needleman-Wunsch

Naveed, Siddiqui, and Ahmed described a method to parallelize the Needleman-Wunsch algorithm in their paper but did not show an actual implementation [3]. We attempted to implement parallelized versions of the Needleman-Wunsch algorithm based on the anti-diagonal method described in their paper. The results of our experiments are described in this section.

For each of our experiments, we implemented and ran a sequential version of the algorithm alongside the parallelized version to compare the runtimes between the two. The sequential version that we implemented is the dynamic programming based algorithm described in the prior section. The main part of our sequential implementation is the fillMatrices() method, which fills up both the scores matrix and traceback matrix for the two input sequences. After the initialization of the first row and first column of each matrix in the constructor, the matrices are filled from left to right starting from the top left based on the scoring system described in the prior section.

Fig. 9 shows the code for our fillMatrices() method. As can be seen in the figure, three scores are calculated for each cell: one based on the adjacent left cell, one based on the adjacent top cell, and one based on the adjacent top left diagonal cell. The scores are stored in the variables *left*, *up*, and *diagonal*. The score based on the diagonal cell depends on whether the two letters representing the current cell is a match. For example, if the letters A and A represents the cell, then we add the match score to the diagonal score. The highest score out of the three calculated scores is chosen and stored into the cell. Next, the corresponding cell in the traceback matrix is filled based on which score was chosen. For example, the character 'd' for diagonal would be stored in the cell if the max calculated score is based on the diagonal cell.

```java
public void fillMatrices() {
    int left, up, diagonal;

    for (int i = 1; i <= lastRowIndex; i++) {
        for (int j = 1; j <= lastColIndex; j++) {
            left = scores[i][j - 1] + gap;
            up = scores[i - 1][j] + gap;

            if (s1.charAt(j - 1) == s2.charAt(i - 1)) {
                diagonal = scores[i - 1][j - 1] + match;
            } else {
                diagonal = scores[i - 1][j - 1] + mismatch;
            }

            scores[i][j] = Math.max(left, Math.max(up, diagonal));

            if (diagonal == scores[i][j]) {
                traceback[i][j] = 'd';
            } else if (up == scores[i][j]) {
                traceback[i][j] = 'u';
            } else {
                traceback[i][j] = 'l';
            }
        }
    }
}
```

Fig. 9. The fillMatrices() method of our sequential Needleman-Wunsch implementation.

The scoring scheme for each implementation is defined at the start of the program. We decided to assign +1 for each match, -1 for each mismatch, and -1 for each gap. These values are stored in variables at the start of the program.

All implementations of the algorithm (both sequential and concurrent) contain the same printOptimal() method that prints the optimal alignment to the screen. Fig. 10 shows the code for this method. The method recursively traces the completed traceback matrix starting from the bottom rightmost cell of the matrix and inserts gaps into the sequences accordingly. It then recursively prints the results to the screen. Fig. 11 shows an example output. The sequences are displayed vertically. For example, the first sequence in the example output is —GT–TA.

In our parallelized versions of the algorithm, we fill in the matrices anti-diagonally. This was achieved by using a nested loop. The outer loop traverses each anti-diagonal starting from the top left and ending at the bottom right. The inner loop,

```java
public void printOptimal() {
    printOptimalHelper(lastColIndex, lastRowIndex);
}

private void printOptimalHelper(int i, int j) {
    if (traceback[i][j] == 'x') {
        return;
    } else if (traceback[i][j] == 'l') {
        printOptimalHelper(i, j - 1);
        System.out.println(s1.charAt(j - 1) + "-");
    } else if (traceback[i][j] == 'u') {
        printOptimalHelper(i - 1, j);
        System.out.println("-" + s2.charAt(i - 1));
    } else if (traceback[i][j] == 'd') {
        printOptimalHelper(i - 1, j - 1);
        System.out.println(s1.charAt(j - 1) + "" + s2.charAt(i - 1));
    }
}
```

Fig. 10. The printOptimal() method used in all Needleman-Wunsch implementations.

```
-T
-C
GG
TT
-G
TT
A-
```

Fig. 11. Example optimal alignment output.

meanwhile, traverses each cell in an anti-diagonal starting from the topmost cell and ending in the leftmost cell.

We ran a sequential version of the algorithm where the matrices are filled in anti-diagonally and found that there was no significant difference in runtime when compared to filling in the matrices in order from left to right starting from the top. Thus, the order of traversal should not affect the runtime.

For all implementations, we tested the results using a pair of short DNA sequences and a pair of long DNA sequences. The short sequences each consists of 15 characters, and the long sequences each consists of 8,191 characters. These sequences are stored in a text file and are read into strings at the beginning of each implementation.

In each parallel version of the algorithm, we run a loop to check the resulting scores matrix against the one produced in the sequential version to make sure that the correct output is produced.

1) Experiment 1: In our first attempt to parallelize the Needleman-Wunsch algorithm, we initially tried using locks. Our initial thought was to lock each anti-diagonal. We attempted to use locks to prevent the next anti-diagonal from being processed until the current one is completely filled in. After some time, we realized that this was the wrong approach because of the nature of locks. Locking access to each anti-diagonal meant that only one thread could enter the anti-diagonal at a time. We want all threads to be able to process the anti-diagonal at the same time. As for when processing each cell, there is no need to lock each cell since we are just

calculating and filling in the score for the cell. Doing this does not affect the scores of any of the other cells within the same anti-diagonal. We realized that there is no need for locks at all.

This led us to experimenting with atomic variables instead. Knowing that atomic variables are expensive, we tried to come up with an implementation that only required the use of one atomic variable. In order to do this, however, we had to create new threads for each anti-diagonal, which we thought might be even more costly than using multiple atomic variables. Nevertheless, we proceeded with the experiment.

In the implementation, we use an atomic variable to keep track of the current row in each anti-diagonal that still needs to be calculated. The score for each cell is calculated the same way it is in the sequential implementation. The main steps are as follows. First, we traverse to the next anti-diagonal that needs to be processed. We then create 8 new threads and have the main process wait until all 8 threads terminate before looping and traversing to the next anti-diagonal. Each of the created threads enter another loop that traverses the rows in the anti-diagonal. This is equivalent to traversing each cell of the anti-diagonal since each of those cells is on a different row. Every time a thread loops, it calls getAndIncrement() on the current row atomic variable to get the row of the cell it needs to process. It processes the cell assigned to it by calculating the cell's score and updating the scores and traceback matrices. This inner loop continues until there are no more cells to process for the current anti-diagonal. At this point, all threads terminate and the main process proceeds to loop to the next anti-diagonal. Eight new threads are created for this anti-diagonal and the steps are repeated until the matrices are filled. Fig. 12 shows the outer loop in the implementation. The inner loop is within the run() method, which gets called when each thread runs.

```
for (antiDiagonal = 1; antiDiagonal < 2 * numRows - 1; antiDiagonal++) {
    currRow = (antiDiagonal < numRows) ? 0 : antiDiagonal - numRows + 1;
    tempCurrRow.set(currRow);
    int numThreads = 8;
    Thread[] threads = new Thread[numThreads];

    for (int i = 0; i < threads.length; i++) {
        threads[i] = new Thread(this);
    }

    for (int i = 0; i < threads.length; i++) {
        threads[i].start();
    }

    for (int i = 0; i < threads.length; i++) {
        try {
            threads[i].join();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
```

Fig. 12. Portion of Experiment 1 code.

Table I shows the results of Experiment 1 using both the short and long DNA sequence pairs. As can be seen, the runtimes for the sequential implementation is significantly better than the parallel implementation in all cases. To isolate the issue, we removed the multi-threading part of the code and ran the program again to test the impact of only the atomic variable. The execution time for the sequential version here was 1973885 ns, while the execution time for the version with the atomic variable was 2329790 ns. There was a decrease in performance here, but not as nearly significant as before. This shows that the constant creating and terminating of new threads in the Experiment 1 implementation contributed significantly to its slow runtime.

TABLE I
EXPERIMENT 1 RESULTS.

		Sequential	**Parallel**	***Speed Up***
Short	*Run 1*	462574 ns	18238115 ns	0.025363036
	Run 2	487001 ns	18840366 ns	0.025848808
	Run 3	469271 ns	20105765 ns	0.023340123
Long	*Run 1*	629235385 ns	7483184535 ns	0.08408658
	Run 2	585775817 ns	7559493370 ns	0.077488765
	Run 3	619996746 ns	7559426547 ns	0.08201637

2) Experiment 2: In our second attempt to parallelize the algorithm, we moved the thread creation portion of the code to the outside of the loop so that it occurs only once. To prevent the outer loop from going to the next anti-diagonal before it's ready to be processed, we had to use two atomic booleans and another atomic integer in addition to the atomic integer used in the previous experiment that was keeping track of the current row. That's a total of 4 atomic variables being used. Despite atomic variables being costly, we thought that the parallelization would offset this cost.

The first atomic boolean, *waitToLoopAgain*, is used to prevent the threads from moving on to the next anti-diagonal before the current one is processed. It is initially set to true. As seen in Fig. 13, we placed a while loop that spins on *waitToLoopAgain* to prevent the threads from moving on until the last cell in the current anti-diagonal is processed. The point at which all the cells in the anti-diagonal is processed is the point when the last thread exits the inner loop. To keep track of this, we used the atomic integer *threadCount* to keep count of the number of threads that has exited the inner loop. Once the last thread has exited the inner loop, we have that thread update some variables to prepare for the processing of the next anti-diagonal. The last thread also sets *waitToLoopAgain* to false to free the other threads from the spin wait. If we loop again directly from this point, a potential problem is that the last thread might become stuck at this point when we reset *waitToLoopAgain* to true again some point before calculating the next anti-diagonal. For instance, if we were to reset *waitToLoopAgain* at the beginning right before the inner loop, one of the other threads might reach the reset point before that last thread is able to get past the spin wait, causing it to be stuck in the last loop iteration. To solve this problem, we introduced another atomic boolean *waitToExecuteLoop*, which guarantees that last thread will always move on to the next iteration. This allows the last thread to be the thread that reset *waitToLoopAgain*. The main downside of this method is

that we have to call incrementAndGet() for each thread again in order to keep count of the number of threads that has exited the spin wait on *waitToLoopAgain*. This adds to the overall cost.

```
while (antiDiagonal < 2 * numRows - 1) {
    int row = currRow.getAndIncrement();

    while (row <= lastRowInAntiDiagonal) {
        int col = antiDiagonal - row;

        if (row == 0 || col == 0) {
            row = currRow.getAndIncrement();
            continue;
        }

        calculateCell(row, col);
        row = currRow.getAndIncrement();
    }

    int loopAgainCount = threadCount.incrementAndGet();

    if (loopAgainCount == NUM_THREADS) {
        antiDiagonal++;
        firstRowInAntiDiagonal = getFirstRowInAntiDiagonal();
        lastRowInAntiDiagonal = antiDiagonal - firstRowInAntiDiagonal;
        currRow.set(firstRowInAntiDiagonal);
        threadCount.set(0);
        waitToExecuteLoop.set(true);
        waitToLoopAgain.set(false);
    }

    while (waitToLoopAgain.get()) {
    }

    int executeLoopCount = threadCount.incrementAndGet();

    if (executeLoopCount == NUM_THREADS) {
        threadCount.set(0);
        waitToLoopAgain.set(true);
        waitToExecuteLoop.set(false);
    }

    while (waitToExecuteLoop.get()) {
    }
```

Fig. 13. Portion of Experiment 2 code.

Table II shows the results of Experiment 2. Again, in all cases, the sequential runtimes are much better than the parallel runtimes. The results here are definitely an improvement over those from Experiment 1. The use multiple atomic variables over constantly creating new threads did have a positive effect on runtimes. However, parallelization did not offset the cost of using all of those atomic variables. We performed a small experiment by testing the effects of atomic variable initialization on runtime and found that just initialization alone slowed down runtime. The more atomic variables we initialized, the more the runtime suffered. So this along with all the getAndIncrement()'s we used within the loops definitely contributed to the dismal results.

3) Experiment 3: In our third attempt to parallelize the algorithm, we attempt to make use of Java's ExecutorService. Java's ExecutorService creates and manages a pool of threads so that we don't have to manage them ourselves [9]. It then allows us to set tasks to it that are to be run. We thought

TABLE II
EXPERIMENT 2 RESULTS.

		Sequential	**Parallel**	***Speed Up***
Short	*Run 1*	572938 ns	4841669 ns	0.118334815
	Run 2	500059 ns	3022491 ns	0.16544598
	Run 3	484493 ns	2832178 ns	0.17106728
Long	*Run 1*	667113765 ns	2022222281 ns	0.3298914
	Run 2	591880250 ns	1708562826 ns	0.34641996
	Run 3	600636087 ns	1812491761 ns	0.33138692

TABLE III
EXPERIMENT 3 RESULTS.

		Sequential	**Parallel**	***Speed Up***
Short	*Run 1*	472872 ns	33430366 ns	0.014144984
	Run 2	504970 ns	29210923 ns	0.017287025
	Run 3	474302 ns	28252854 ns	0.016787756
Long	*Run 1*	650375441 ns	22830851413 ns	0.028486691
	Run 2	603525660 ns	21543061968 ns	0.02801485
	Run 3	596988249 ns	21576963582 ns	0.02766785

that perhaps ExecutorService would manage the parallelization better than we did in Experiment 2. To use ExecutorService, it first needs to be instantiated with the number of threads passed in as an argument. We can then use the submit() method to pass in a function that is to be run. Once done, we need to call the shutdown() method to terminate the threads. Otherwise, it will run indefinitely.

Initially, we instantiated the executor outside of the outer loop that traverses the anti-diagonals. We placed the submit() method call inside the inner loop that traverse the cells within each anti-diagonal. We pass to it a lambda function that calculates the cell scores. After a few runs the program, we got the an incorrect output. The matrix produced by the parallel implementation did not match the one produced by the sequential one. This means that the executor isn't always filling in the cells in the order that we expect it to. To remedy this, we moved the instantiation of the executor to the inside of the outer loop. Fig. 14 shows this. Moving the instantiation fixed the issue. However, now threads are being created every iteration of the outer loop. This is a situation similar to that of Experiment 1.

Table III shows the results of Experiment 3. As expected, the results are similar to those of Experiment 1 where the parallel runtimes are significantly higher than the sequential ones due to the constant creation of new threads. Our attempt to use Java's ExecutorService to manage threads did not lead to an improvement over Experiment 2.

4) Experiment 4: In our forth attempt to parallelize the algorithm, we try to make use of Java's IntStream parallel() method. Java streams allows us to work more easily with collections of objects by providing functions such as min(), max(), filter(), sum(), and more [10]. An IntStream allows us to work with integers. Using an IntStream, we can run a parallel for-each loop using the range(), forEach(), and parallel() methods. To do this, we specify the range to iterate through in range() and the code to loop through in forEach(). We also call parallel() to request that the loop runs concurrently.

```java
for (int i = 0; i < 2 * n - 1; i++) {
    int z = (i < n) ? 0 : i - n + 1;
    var executor = Executors.newFixedThreadPool(8);

    for (int j = z; j <= i - z; j++) {
        final int row = j;
        final int col = i - j;

        if (row == 0 || col == 0) {
            continue;
        }

        executor.submit(() -> {
            int left = scores[row][col - 1] + gap;
            int up = scores[row - 1][col] + gap;
            int diagonal;

            if (s1.charAt(col - 1) == s2.charAt(row - 1)) {
                diagonal = scores[row - 1][col - 1] + match;
            } else {
                diagonal = scores[row - 1][col - 1] + mismatch;
            }

            scores[row][col] = Math.max(left, Math.max(up, diagonal));

            if (scores[row][col] == diagonal) {
                traceback[row][col] = 'd';
            } else if (scores[row][col] == up) {
                traceback[row][col] = 'u';
            } else {
                traceback[row][col] = 'l';
            }
        });
    }

    executor.shutdown();

    try {
        boolean finished = executor.awaitTermination(1, TimeUnit.MINUTES);
        while (!finished) {
        }
```

Fig. 14. Portion of Experiment 3 code.

Fig. 15 shows our implementation of the algorithm using IntStream parallel(). Here, the outer loop traversing through each anti-diagonal operates as normal. The inner loop is implemented using IntStream parallel(), allowing it to be run concurrently. So the calculations for the cells within each anti-diagonal is done concurrently. In doing this, we were hoping that the stream would do a better job at parallelizing the calculations than we did in Experiment 2.

```java
for (antiDiagonal = 2; antiDiagonal < 2 * numRows - 1; antiDiagonal++) {
    firstRowInAntiDiagonal = getFirstRowInAntiDiagonal();
    lastRowInAntiDiagonal = antiDiagonal - firstRowInAntiDiagonal;

    IntStream.range(firstRowInAntiDiagonal, lastRowInAntiDiagonal + 1)
        .parallel().forEach(row -> {
        int col = antiDiagonal - row;

        if (row == 0 || col == 0) {
        } else {
            calculateCell(row, col);
        }
    });
}
```

Fig. 15. Portion of Experiment 4 code.

Table IV shows the results of Experiment 4. The results

TABLE IV
EXPERIMENT 4 RESULTS.

		Sequential	Parallel	*Speed Up*
Short	*Run 1*	471986 ns	11916292 ns	0.039608464
	Run 2	505499 ns	12174915 ns	0.041519716
	Run 3	516278 ns	12457773 ns	0.041442238
Long	*Run 1*	782632610 ns	1298789099 ns	0.6025864
	Run 2	594413865 ns	1169940500 ns	0.5080719
	Run 3	619870475 ns	1166825815 ns	0.53124505

are an improvement over those of Experiment 2 for both the short and long DNA sequence pairs. So the stream did indeed do a better job at parallelizing the calculations. However, the sequential runtimes are still at least 2 times faster than the parallel runtimes. The parallel implementation still has not yet won over the sequential one.

These results are the best we have gotten so far. At this point, we considered that maybe we would see improvements if we increased the sizes of the input sequences. Perhaps there is a certain minimum size where we would start seeing the benefits of the parallelized version. We doubled the length of long sequences to be 16,382 characters each and tested the code again. The runtime did not improve. It would stay around the same as before (around 0.5 speed up) and sometimes would even be worse (down to 0.45 speed up).

Next, we attempted to make the code non-blocking by parallelizing both the outer and inner loops to see what would happen. First, we initialized each cell of the scores matrix with Integer.MIN_VALUE. We then implemented the two loops using IntStream parallel() to have them run concurrently. In the calculateCell() method, which determines the score for a cell and gets called in the inner loop, we placed a spin wait on the left, top, and top left diagonal neighbor cells. The spin wait runs until the neighboring cells' values are calculated. That is, it spins until they are not equal to Integer.MIN_VALUE.

We soon found that the problem with this implementation was that it would almost always lead to a deadlock, especially with the longer sequence. And the reason for this is due to the number of threads spawned by the stream being limited. If all threads land on a cell whose neighbors have not been calculated, they will all spin forever and lead to a deadlock.

REFERENCES

[1] V. Chaudhary, F. Liu, V. Matta, and L. T. Yang, "Parallel implementations of local sequence alignment: Hardware and software," *Parallel Computing for Bioinformatics and Computational Biology*, p. 233–264, 2005.

[2] "Global alignment of two sequences - needleman-wunsch algorithm," *Amrita*. [Online]. Available: https://vlab.amrita.edu/index.php?sub=3&brch=274&sim=1431&cnt=1

[3] T. Naveed, I. S. Siddiqui, and S. Ahmed, "Parallel needleman-wunsch algorithm for grid."

[4] B. Settles, "Lecture 2: Sequence alignment," *University of Wisconsin-Madison*, 2008. [Online]. Available: http://pages.cs.wisc.edu/ bsettles/ibs08/lectures/02-alignment.pdf

[5] L. Vladimir, "The needleman-wunsch algorithm for sequence alignment," *SJSU*. [Online]. Available: https://www.cs.sjsu.edu/ aid/cs152/NeedlemanWunsch.pdf

[6] G. Musso, "Sequence alignment (needleman-wunsch, smith-waterman)," *University of Toronto*. [Online]. Available: https://www.cs.utoronto.ca/ brudno/bcb410/lec2notes.pdf

[7] D. S. Hirschberg, “A linear space algorithm for computing maximal common subsequences,” *Communications of the ACM*, vol. 18, no. 6, p. 341–343, 1975.

[8] “Hirschberg’s algorithm: Edit-distance revisited,” *Monash University*. [Online]. Available: http://users.monash.edu/ lloyd/tildeAlgDS/Dynamic/Hirsch/

[9] baeldung, “A guide to the java executorservice,” *Baeldung*, Feb 2021. [Online]. Available: https://www.baeldung.com/java-executor-service-tutorial

[10] ——, “Primitive type streams in java 8,” *Baeldung*, Oct 2019. [Online]. Available: https://www.baeldung.com/java-8-primitive-streams