Parallel Needlemam-Wunsch Algorithm for Sequence Alignment

Final Report - COMS 4995 Parallel Functional Programming

Phillip Le (pnl2111@columbia.edu) & Emily Lo (ej2192@barnard.edu)

1 INTRODUCTION

The Needleman-Wunsch (NW) Sequence Alignment algorithm is vital in bioinformatics for aligning nucleotide sequences. It enables the observation of genetic similarities, differences, and patterns, crucial for understanding ancestry, evolution, and diseases. However, due to the considerable length of nucleotide sequences, optimizing the algorithm is an ongoing focus in Computer Science [1].

In our project, we've delved into current optimization strategies, particularly in Haskell. We've created and evaluated four distinct NW algorithm implementations, assessing their individual strengths and limitations.

2 GENERAL ALGORITHM

This algorithm takes as input two sequences and seeks to maximize their alignment based on a scoring system, doing so through inserting gaps in either sequence or accepting misalignments.The algorithm makes the decision between either option based on the defined scoring schema and the scores of the neighboring cells. The output of the algorithm is an optimal alignment between the two sequences through the introduction of gaps.

Finally, there is a 'traceback' operation which traces through the matrix and assembles the optimal alignment based on the scores. From the bottom right, the traceback looks at the scores and determines path of alignment based on whether there was a match, mismatch, or gap according to the scoring and penalty rules. This happens until the it reaches the top left of the matrix.

3 LIMITATIONS

Any implementation of the Needleman-Wunsch algorithm is bound by the face that the scores of the individual cells in the matrix are computed based on the maximum score of applying the scoring scheme with the cells directly atop, to the left, and top-diagonal of the current cell. Therefore, every implementation is limited in that when beginning the calculation of the score of the current cell, the calculations for the previous cells must be complete.

4 SEQUENTIAL APPROACH

The sequential implementation simply operates within nested for loops, computing the scores of the cells top to bottom starting from the left. This works simply because the implementation delivers that guarantee that the computation of the cells that are dependencies for the score of the current cells already have their score.

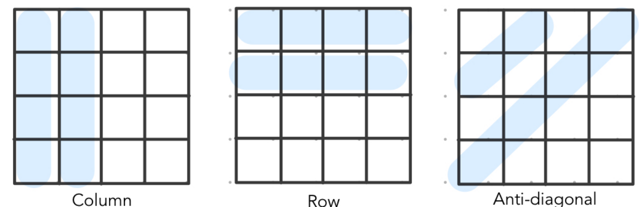
Here is the pseudocode for a sequential approach:

```
func nw-alignment (seq1: [char], seq2: [char], score: ScoreScheme)

    rows = len(seq1) + 1
    columns = len(seq2) + 1

    scores = create 2d array (rows, columns)

    //populate the first row and column
    for i from 0 to rows:
          score = score[i][0] = i * score.gapPenalty
    for j from 0 to columns
          score = score[0][j] = j * score.gapPenalty

    ** Iterate through each cell from, row by row
    for i from 1 to rows:
          for j from to 1 to columns:
                if seq2[i-1] == seq1[j-1]:
                      match = scores[i-1][j-1] + score.matchScore
                else:
                      match = scores[i-1][j-1] + score.mismatchPenalty
                gap1 = scores[i][j-1] - score.gapPenalty
                gap2 = scores[i-1][j] - score.gapPenalty

                score[i][j] = max(match , gap1, gap2)
```

5 PARALLEL APPROACH

In considering the parallelization of the Needleman-Wunsch algorithm, we encountered limitations with the dependencies on the previous cells in the calculation of the current cell's score. Additionally, because the score computation involves minimal processing, we couldn't fragment the operation to run in parallel.

So our parallel approaches were instead informed by how we can divide the matrix into sections. There are three ways to divide the matrix such that we still maintain the dependency of the score of the previous cell in the score of the current cell. 1) Defining sections by columns of matrix 2) Sections based on row, and 3) defining sections based on the anti-diagonals in the matrix [2]. Refer to the illustration,



6 HASKELL IMPLEMENTATIONS

6.1 Developing Infrastructure

We developed four Haskell implementations of the NW-algorithm and its optimizations. Though we mentioned that the algorithm is two-part, populating the matrix and performing the traceback, we set our focus on the former. Other than that, everything stayed the same such that we can see the optimization of just the matrix computation.

6.1.2 Test Cases

We compiled three test cases from the genetic sequences of human and rat mitochondria with each case varying in length of the input strings. The first is testing with two

strings each of length 100, 500, 1000, then 2000. Though current research of the optimization of the NW-algorithm test with input sequences of lengths much longer, since we are developing and testing on Macs we found that these were the most appropriate sizes.

6.1.3 Scoring Scheme and Functions

Our next steps were establish a scoring criteria that underpinned our score calculations in the matrix. Introducing a gap in either sequence incurred a 1 point deduction, while a sequence mismatch resulted in a 2 point deduction. To implement this, we defined the *Scoring* data type and corresponding function:

```
data Scoring = Scoring
  { matchScore :: Int
  , mismatchPenalty :: Int
  , gapPenalty :: Int
  }

score :: Scoring -> Char -> Char -> Int
score scoring a b
  | a == b     = matchScore scoring
  | otherwise = -mismatchPenalty scoring
```

6.1.4 Calculate Score Function

The calculation of the score is consistent across all implementations. The input is a tuple, which represents the row and column of the current cell. There are 'default scores' for the 0th row and column which are the base-case of the matrix, otherwise, the score is the max of different options between applying the score adjustment for mismatch/match and the insertion of gap into either sequence. Here is that implementation:

```
calculateScore (i, j)
  | i == 0     = -j * gapPenalty scoring
  | j == 0     = -i * gapPenalty scoring
  | otherwise = maximum
                [ scores ! (i-1, j-1) + score scoring (s1 !! (i-1)) (s2 !! (j-1))
                , scores ! (i, j-1) - gapPenalty scoring
                , scores ! (i-1, j) - gapPenalty scoring
                ]
```

6.1.5 Traceback Function

We developed the following traceback function which is called on the resulting score-matrix and the original input sequences. The output is the most-aligned sequences according to the scores-matrix, and scoring scheme.

```
traceback :: Array (Int, Int) Int -> String -> String -> (String, String)
traceback scores s1 s2 = go (n, m) ("", "")
  where
    (n, m) = snd $ bounds scores

    go (i, j) (align1, align2)
      | i == 0 && j == 0 = (align1, align2)
      | i > 0 && scores ! (i, j) == scores ! (i-1, j) - 1 =
        let newAlign1 = s1 !! (i-1) : align1
            newAlign2 = '-' : align2
        in go (i-1, j) (newAlign1, newAlign2)
      | j > 0 && scores ! (i, j) == scores ! (i, j-1) - 1 =
        let newAlign1 = '-' : align1
            newAlign2 = s2 !! (j-1) : align2
        in go (i, j-1) (newAlign1, newAlign2)
      | otherwise =
        let newAlign1 = s1 !! (i-1) : align1
            newAlign2 = s2 !! (j-1) : align2
        in go (i-1, j-1) (newAlign1, newAlign2)
```

6.2 NW-Algorithm Four Ways

We implemented the NW-Algorithm four different ways which we go into detail and describe in the following sections. Each function accept as input the scoring scheme, the two sequences, and returns the 2D-array which represent the scores-matrix.

6.2.1 Sequential Algorithm Implementation

The base implementation for the NW-algorithm follows the same concept of the pseudocode provided above. We simply generated the list of indices between 0 and length of sequence 1 and sequence 2 and generated our matrix (2D array) using Haskell's `map` function to calculate the scores of all the cells provided by our list of indices. The calculating the scores of the cells starts from the top-left

```
sequential :: Scoring -> String -> String -> Array (Int, Int) Int
sequential scoring s1 s2 =
    let n = length s1
        m = length s2
        indexes = [(i, j) | i <- [0..n], j <- [0..m]]

        ...

        scores = listArray ((0, 0), (n, m)) $ map calculateScore indexes
    in scores
```

cell to the bottom-right cell. Here is that code:

6.2.2 Naive Parallel Approach, Partition Score-Matrix by Column and Row

The first two approaches to parallelize the NW-algorithm aimed to minimize the number of generated sparks and the consequent drawbacks of their excessive utilization (GC Time). That is, create a spark per column or rows, and compute the scores of those columns or rows in parallel. We can call these, 'Parallel Column Approach' and 'Row Column Approach'.

Both implementations use `parMap` with `rpar` strategy to attempt the calculation of scores in entire columns in parallel rather than individual cells since regardless, the computation of cell-scores amongst these partitions need to be serial.

The difference between the sequential implementation these parallel approaches is the method in which the score-matrix is populated. For column,

And for row, we switch the variable we iterate through.

```
scores = array ((0, 0), (n, m)) $ do
  let columnCompute j = [((i, j), calculateScore i j) | i <- [0..n]]
  concat $ parMap rpar columnCompute [0..n]
```

6.2.3 Optimal Parallel Approach, Partition Score-Matrix by Anti-Diagonal

The anti-diagonal alignment strategy presents a unique advantage amongst the row by row, column by column, and cell by cell evaluation strategy [3]. This strategy divides the matrix into anti-diagonal stripes. The algorithm must serially compute the scores of the anti-diagonals as a whole. However, the individual cells in the anti-diagonal can all be computed in parallel. This strategy maximizes parallelization because given a current

cell in the anti-diagonal, all the cells necessary for the deriving the score of the current cell have been computed in the previous anti-diagonal. Refer to this illustration:

First, we generate an array of arrays of tuples, each inner-array includes the indices of an anti-diagonal stripe in the matrix given its size:

```
antidiagonalIndices :: Int -> [[(Int, Int)]]
antidiagonalIndices n =
  [ [(i, k - i) | i <- [0..k], k - i < n, k - i >= 0 && i < n] | k <- [0..2*(n-1)] ]
```

We first experimented with creating the score-matrix using `listArray` and initializing all cells to 0. Upon completion of generating the scores of a anti-diagonal , we made 'updates' to the matrix. However, since `listArray` is immutable, these updates meant we had to create an entirely new `listArray` which was incurred extensive memory usage and overhead from the copying operation. Therefore, we used an immutable array, IOUArray, from the Data.IO and initialized all values to 0, like so:

```
scores <- newArray ((0, 0), (n - 1, m - 1)) 0 :: IO (IOUArray (Int, Int) Int)
```

We then are able to use the mapM function on each diagonal in the generated list of anti-diagonals and call the function, computeDiagonal, passing to it the IOUArray and the input strings. Inside we define an inline function which calculates the score at the index and writes the score in the IOUArray at that index. We are able to compute the score of the cell and write it concurrently because of the structure of the code.

Specifically, we computeDiagonal serially through the list of all diagonals, ensuring that before computation of the next anti-diagonal, the calculations of the prior ones are complete, therefore, eliminating any race conditions and also enabling parallel computation of the individual scores of the cells.

```
computeDiagonal :: IOUArray (Int, Int) Int -> [(Int, Int)] -> Scoring -> Array Int Char -
computeDiagonal scores stripe scoring s1 s2 = do
    let calculateAndUpdate (i, j) = do
            calculatedScore <- calculateScoreinDiagonal scores scoring s1 s2 i j
            writeIndicesToScores scores (i, j) calculatedScore
    -- Concurrent computation for each cell in the stripe
    _ <- mapConcurrently calculateAndUpdate stripe
    return ()
```

Since we want to make sure that we are able to really isolate the results of our testing to the the efficiency/deficiency of our score-matrix calculation methods, we needed to use the same traceback function. Therefore we had to use the `freeze` function in order to return a matrix of type IO(Array(Int, Int) Int).

```
frozenScores <- freeze scores
return frozenScores
```

6.2.4 REVISED Optimal Parallel Approach, Partition Score-Matrix by Anti-Diagonal

We noticed that our implementation never used more than two cores, so we went back to the drawing board. We thought more algorithmically (with Professor Edwards) and looked at each individual cell, what information was really needed to compute the score.

Building upon this observation we decided to use the two previous diagonals as input to calculate the next diagonal. Specifically we stored the calculated scores from previous diagonals as arrays. Additionally, we also used vectors to represent the sequences since we were indexing frequently. We noticed that when used strategies like `parMap` with `rpar` we were incurring so much GC time that made the tradeoff for more parallelization one that was not fair. Instead, we implemented a spark by chunk strategy. This is the entry point of our function:

```
antidiagonal :: String -> String -> [Array Int Int]
antidiagonal firstStr secondStr =
  let n = length firstStr
      m = length secondStr
      seq1 = stringToVector firstStr
      seq2 = stringToVector secondStr
      base = [array (0, 1) [(0, -1), (1, -1)], array (0, 0) [(0, 0)]]
  in foldl' (createDiagonal seq1 seq2 n m) base [2..n+m]
```

We used foldl' to apply the function `createDiagonal` as many times as there were diagonals. Create diagonal function takes the two first anti-diagonals, which are default values.

```
createDiagonal :: V.Vector Char -> V.Vector Char -> Int -> Int -> [Array Int Int] -> Int -> [Array Int Int]
createDiagonal seq1 seq2 n m diagonals i = case diagonals of
    (currentDiagonal:prevDiagonal:_) ->
        let newDiagonal = calcForDiagonal seq1 seq2 prevDiagonal currentDiagonal i n m
        in newDiagonal `seq` (newDiagonal : diagonals)
    _ -> error "createDiagonal: Less than two diagonals provided"

calcForDiagonal :: V.Vector Char -> V.Vector Char -> Array Int Int -> Array Int Int -> Int -> Int -> Int -> Array Int Int
calcForDiagonal sec1 sec2 prevDiag1 prevDiag2 counter n m =
  let indices = if counter <= m then [0..counter] else [counter - m..n]
      chunkSize = 10
      scoreList = map (calculateValue sec1 sec2 prevDiag1 prevDiag2 counter n m) indices `using` parListChunk chunkSize rpar
  in listArray (0, length scoreList - 1) scoreList
```

CreateDiagonal is called with the initial parameters, it then calculates a new diagonal using the calcForDiagonal function, which is then added to the list of diagonals. We are adding to the list through prepending, since it is a cheaper operation. This process occurs iteratively, ensuring that by the time we compute the next anti-diagonal, the information that it needs is available.

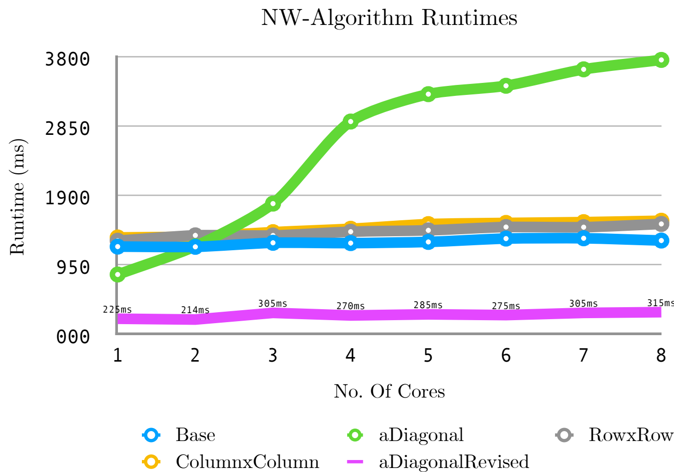
Also, notice that in calcForDiagonal, we address the issue of over-sparking. We defined the chunkSize to 10, which we found optimal between all cores. This function will divide the computation and compute it in parallel between 10 chunks. This really decreased our GC Time and increased our overall efficiency.

Lastly, since we wanted to use the same traceback function. But, the anti-diagonal function returned a different type. We had to provide a function to convert the [Array Int Int] to Array(Int, Int) Int by simply iterating through all the diagonals populating the matrix

from left to right by taking the last item in the anti-diagonal list. Then, we are able to call traceback.

7 RESULTS

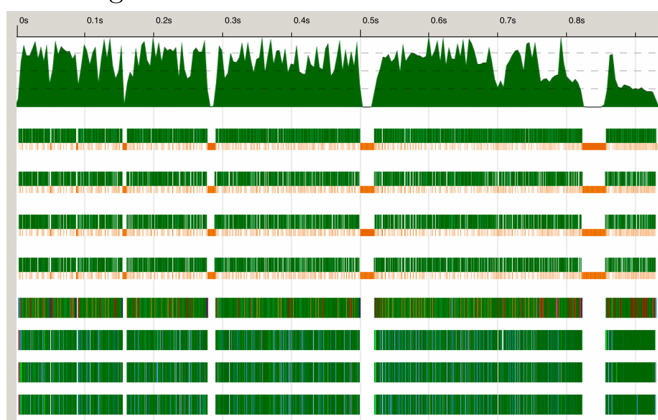
All the rests ran were performed on a 2022 MacBook Pro with 8 cores. We ran the tests over the course of 3 days and found the results to be consistent. The results in the graph below are with an two-input sequences of length 1000. Here are the runtimes:



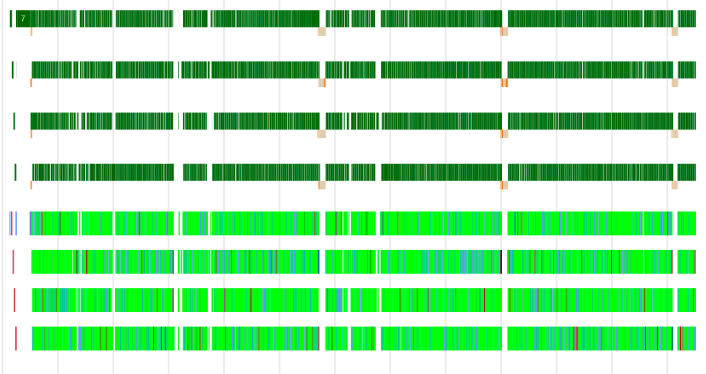
The results of the base, row by row, and column by column approach are unsurprising in that they are quite similar. However, the base (sequential) implementation beats them by a smaller GC time contribution.

But, is clear that the anti-diagonal approach was much faster. We were surprised at how flat the line was when increasing the number of cores. The fastest time was for the 2-core test. We conclude this to be because our chunkSize was constant between the tests and in theory should be changed for optimal performance based on the number of cores available.

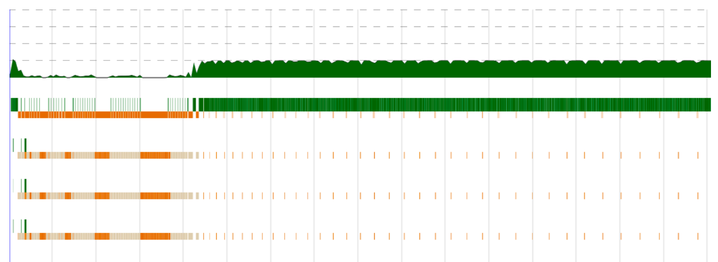
Lastly, we noticed that between all implementations the activity was the most balanced for the revised anti-diagonal approach and it most efficiently took advantage of runs with multi-core:



In comparison to our first attempt at anti-diagonal implementation (same test, and bright green indicates waking up threads a.k.a a lot of sitting and waiting and maintaining concurrently rather than being concurrent):



And finally, the column by column implementation. Notice, activity all on at most one core and large chunk of time I the middle of just GC:



8 Conclusion

For our final project, we implemented five different implementations of the Needleman-Wunsch Sequence Alignment Algorithm in Haskell. We developed code infrastructure and instituted extensive testing on the results and were able to put what we learned in our Parallel Functional Programming course with Professor Stephen Edwards into practice.

In the future, we hope to return to the implementation of the anti-diagonal approach with different parallel strategies that may lead to results in its favor.

10 REFERENCES

[1] T. Roughgarden. 2021. Algorithms Illuminated (Part 3): Greedy Algorithms and Dynamic Programming.

[2] Y. S. Lee, Y. S. Kim, R. L. Uy. “Serial and parallel implementation of Needleman-Wunsch algorithm, ” 2020 International Journal of Advances in Intelligent Informatics. doi: 10.26555/ijain.v6i1.361.

[3] V. Gancheva and I. Georgiev, "Multithreaded Parallel Sequence Alignment Based on Needleman-Wunsch Algorithm," 2019 IEEE 19th International Conference on Bioinformatics and Bioengineering (BIBE), Athens, Greece, 2019, pp. 165-169, doi: 10.1109/BIBE.2019.00037.