On the Efficacy of Haskell for High-Performance Computational Biology

Jacqueline Addesa
Academic Advisors: Jeremy Archuleta, Wu-chun Feng

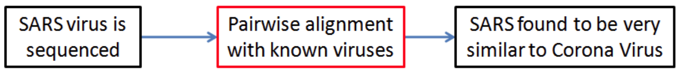
1. Problem and Motivation

Biologists can leverage the power of computers to expedite discoveries about everything from how evolution occurred to how to target an emerging virus. In particular, geneticists would like to determine how and which mutations and changes occur in organisms over time by comparing the genetic information. Unfortunately, there exists a chasm between the knowledgeable domain scientist and the knowledgeable computer scientist whereby creating efficient and correct computer programs is difficult.

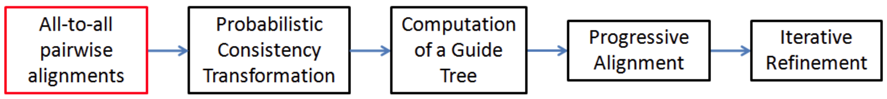
Furthermore, each and every day the amount of digital genetic information substantially increases thanks in large part to the new technology of next-generation sequence machines and the new area of metagenomics. Fully analyzing this "deluge of data" is imperative in order to understand how organisms are similar (compare) and which organisms are present (classify) and can only be accomplished efficiently with computers.

Solving this problem efficiently requires an interdisciplinary knowledge of both biology and the computer science. This work attempts to apply known computer science to the biological problem of pairwise sequence alignment (the comparison of only two sequences) in an effort to make finding the alignment faster and require less of a resource overhead.

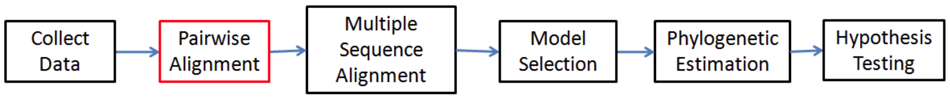
The primary method to perform this computational comparison and classification is sequence alignment. Sequence alignment is the comparison of two or more genetic sequences to determine areas of similarity. Identifying these similarities is important for several reasons. First, these similarities can be used to aid the creation of drug treatments: if an emerging virus (e.g. SARS) is found to be closely similar to an existing virus (e.g., Coronavirus), then perhaps existing drug regimens can be effective to combat and treat the emerging virus (see Figure 1A). Second, determining the similarity of two sequences is one of the first steps in multiple sequence alignment which is used to create phylogenetic trees (see Figure 1B and 1C). Phylogenetic trees can be used to determine how evolution progressed as well as determining which species are closely related enough to serve as models for other species in genetic experiments.


SARS virus is sequenced
Pairwise alignment with known viruses
SARS found to be very similar to Corona Virus

(A)


All-to-all pairwise alignments
Probabilistic Consistency Transformation
Computation of a Guide Tree
Progressive Alignment
Iterative Refinement

(B)



(C)

Figure 1. (A) An example use of pairwise alignment. (B) The cycle of computing multiple sequence alignment using ProbCons [1]. (C) The simplified phylogenetic process.

The necessary process to perform this sequence alignment often consists of evaluating the unknown digital DNA sequence against a very large database of known sequences, such as those found at the National Center of Biotechnology Information (NCBI). The size and growth of these databases requires high-performance computing in order to determine a solution in a reasonable amount of time and has thus far focused predominantly on using MPI and C to create parallel tools (mpiBLAST [2], mpiHMMER [3]).

This work evaluates a technology typically not used for computational biology. Haskell, a functional programming language that is frequently used for web services, is rarely used for high performance computational biology. A key feature of Haskell for this work is that it uses lazy-evaluation to determine the parameters of a function. It is also under active development and is inherently parallel due to its functional nature (i.e. functions have no side-effects). The fact that Haskell is functional can lead to shorter, more concise code. This reduction in the code base creates code which is easier to maintain. Also, functional languages allow for deeper recursion than what is supported by C.

Given these benefits, this work was undertaken to analyze whether or not Haskell should be used for high performance computational biology, specifically for the problem of pairwise sequence alignment.

2. Related Work

The most fundamental building block of life is DNA, which controls the individual cells of every living organism. One way of analyzing DNA is to convert the physical object into a digital representation using sequencing machines. The resulting digital genetic sequence is composed of the characters "A", "C", "G", and "T" (representing the physical components adenine, cytosine, guanine, and thymine respectively), and can be analyzed in depth using computers.

The process of determining how similar two sequences are is called sequence alignment. Needleman-Wunsch [4] is an algorithm to compute the similarity between two genetic sequences, which is $O(n^2)$ in time and space. A global alignment is determining the minimum number of changes to transform one entire sequence into another. A local alignment, on the other hand, will find the most closely related subsequence. In this work, we focus on finding the global alignment.

The global alignment of the two sequences is computed by filling in a score matrix, using dynamic programming where each cell is computed by adding a penalty to the minimum of three other cells as shown graphically in Figure 2. Mathematically, if X is the target cell to compute, X = min(NW, N, W) + match_penalty. If the two characters of the sequence match for this cell, then the match_penalty will be zero, and no penalty will be incurred and the minimum of the surrounding cells will be used. If the two characters do not match, the penalty is incurred. This way of computing X means that the cells that X is dependent on must be calculated first. Since the goal is to determine global alignment, the bottom right hand corner of the matrix must be computed. A brute force approach results in the filling in of the entire matrix, even though most of it is in fact unneeded.

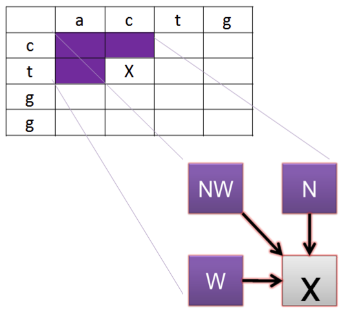


Figure 2. Score matrix and dependencies for Needleman-Wunsch

Some of the matrix will not need to be computed, since the lower scores tend to remain along the main diagonal, it is not necessary to compute all of the corners. For example, in Figure 3, it would not be necessary to compute the cells in the upper right and lower left corners since these scores are not going to affect the score in the bottom right corner. Not evaluating these cells increases performance in two ways. First, since the cells to do not have to evaluated, the time to evaluate them will be saved. Second, as the cells are not evaluated, they do have to be created in memory. Therefore, this slight change will improve both the running time and decrease the resource overhead.

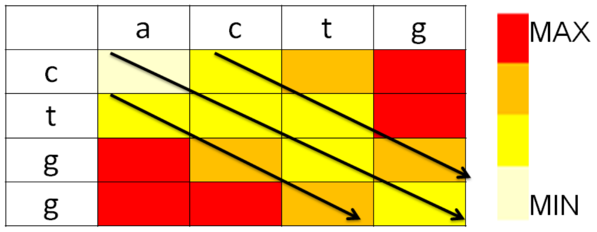


Figure 3. A sample score matrix.

Allison's paper, "Lazy Dynamic-Programming can be Eager" [5] outlines an algorithm for functional programming languages to evaluate only the neccessary cells in the score matrix for the alignment of two sequences. The banded algorithm computes along diagonals from the upper left hand corner to the bottom right hand corner. The banded final algorithm proposed by Allison uses a special minimum to prevent computation of unnecessary cells.

The objective of the algorithm is to follow the path with the lowest score from the top left corner to the bottom right corner; therefore, eliminating the computation of unnecessary cells, the ones with the highest scores. The cells are computed on an as needed basis with the special minimum that takes into account the nature of how scores are computed to bias the search along the diagonals closes to the main diagonal.

3. Uniqueness of Approach

While the biological and computer science knowledge used in this project existed in a fragmented form, our contribution is the blending all of this scattered knowledge together to solve real-world, computationally-intensive problems. Instead of creating a completely new approach, this work focuses on leveraging existing techniques to facilitate biological discovery. Specifically, we evaluated the same algorithm in both functional and imperative languages. The purpose is to understand the effects of merging theoretical knowledge about functional programming languages and existing knowledge of biological algorithms in the context of the efficiency and maintenance of these algorithms.

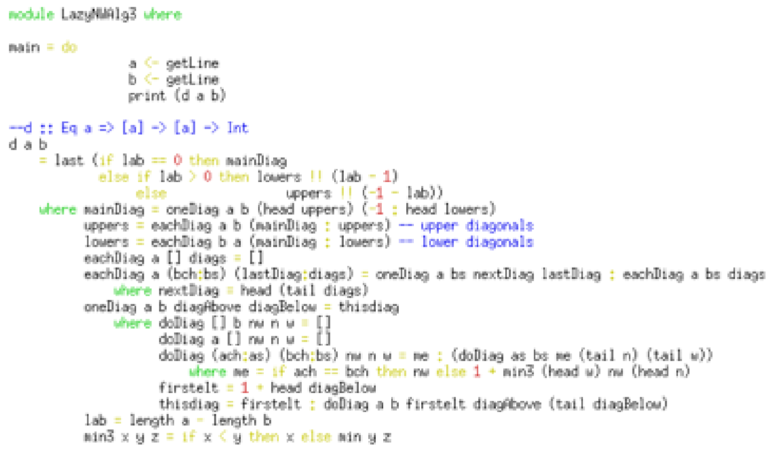
The end result of pulling all of these techniques together is a significant performance improvement coupled with a drastically reduced code base. The details of working in a language not well suited to deep recursion as compared to a language with lazy-evaluation and the resulting performance effects will be discussed in the results section below.

4. Results and Contribution

4.1 Method of Evaluation

An implementation of the banded algorithm was implemented in the purely functional Haskell programming language. We decided to compare this implementation with an imperative implementation, using the C programming language, that mirrored the Haskell implementation as closely as possible. This allowed us to compare the how each language was able to compute the solution using this banded algorithm as opposed to studying the differences in algorithmic approaches. Since Haskell is rarely used in high performance computational biology, this research provided us with an opportunity to determine if this lack of use was justified.

Two C algorithms were implemented, a banded version that computes exactly the same thing as the Haskell implementation and a non-banded, brute force version which fills in the entire matrix in the same order as the banded version. The Haskell algorithm is purely recursive (as can be seen by Figure 4), but this was not possible in C as the level of recursion need to run the larger sequences we tested was not supported. Therefore, a modified recursive model had to be made for the banded C implementation. This implementation has a stack data structure which keeps track of the recursion. The non-banded, brute force C implementation is the same as the banded C implementation, except that the implementation of the minimum is the standard minimum instead of the special minimum for the banded version.



```haskell
module LazyNWAlg3 where

main = do
                a <- getLine
                b <- getLine
                print (d a b)

--d :: Eq a => [a] -> [a] -> Int
d a b
    = last (if lab == 0 then mainDiag
            else if lab > 0 then lowers !! (lab - 1)
                 else                   uppers !! (-1 - lab))
    where mainDiag = oneDiag a b (head uppers) (-1 : head lowers)
          uppers = eachDiag a b (mainDiag : uppers) -- upper diagonals
          lowers = eachDiag b a (mainDiag : lowers) -- lower diagonals
          eachDiag a [] diags = []
          eachDiag a (bch:bs) (lastDiag:diags) = oneDiag a bs nextDiag lastDiag : eachDiag a bs diags
              where nextDiag = head (tail diags)
          oneDiag a b diagAbove diagBelow = thisdiag
              where doDiag [] b nw n w = []
                    doDiag a [] nw n w = []
                    doDiag (ach:as) (bch:bs) nw n w = me : (doDiag as bs me (tail n) (tail w))
                        where me = if ach == bch then nw else 1 + min3 (head w) nw (head n)
                    firstelt = 1 + head diagBelow
                    thisdiag = firstelt : doDiag a b firstelt diagAbove (tail diagBelow)
          lab = length a - length b
          min3 x y z = if x < y then x else min y z
```

Figure 4. Haskell source code.

In order to evaluate the Haskell and C implementations of the banded Needleman-Wunsch code, random sequences of lengths varying between 10 and 1800 characters were used. Each pair of sequences consisted of two sequences of equal length. .The algorithms were compiled using GCC (GNU Compiler Collection) 4.3.2 using the -O3 optimization level and GHC (Glasgow Haskell Compiler) 6.10.1 with no added optimization flags. The serial tests were then run on a Quad-core 2.8GHz Intel E5462 computer with 6MB cache with 8 GB RAM.

4.2 Results

The Haskell implementation significantly outperformed both C implementations for larger sequences, as can be seen by Figure 5. Haskell provides significant improvement over both the banded and non-banded, brute force C implementations. For smaller sequences, the C implementations perform better than the Haskell code, as can be seen in Figure 6. This is because the Haskell code is compiled at runtime, so there is a startup costs associated with using the interpreted Haskell. We believe that the runtime compilation penalty can be removed by performing the compilation prior to execution.

The crossover point between sequence length of 400 and 500 bases represents the point at which this penalty does not make the performance of the Haskell implementation worse than the performance of the C implementation. Also, the runtime curves for both of the C implementations appear to be exponential, whereas the runtime curve for the Haskell implementation appears to be near linear.

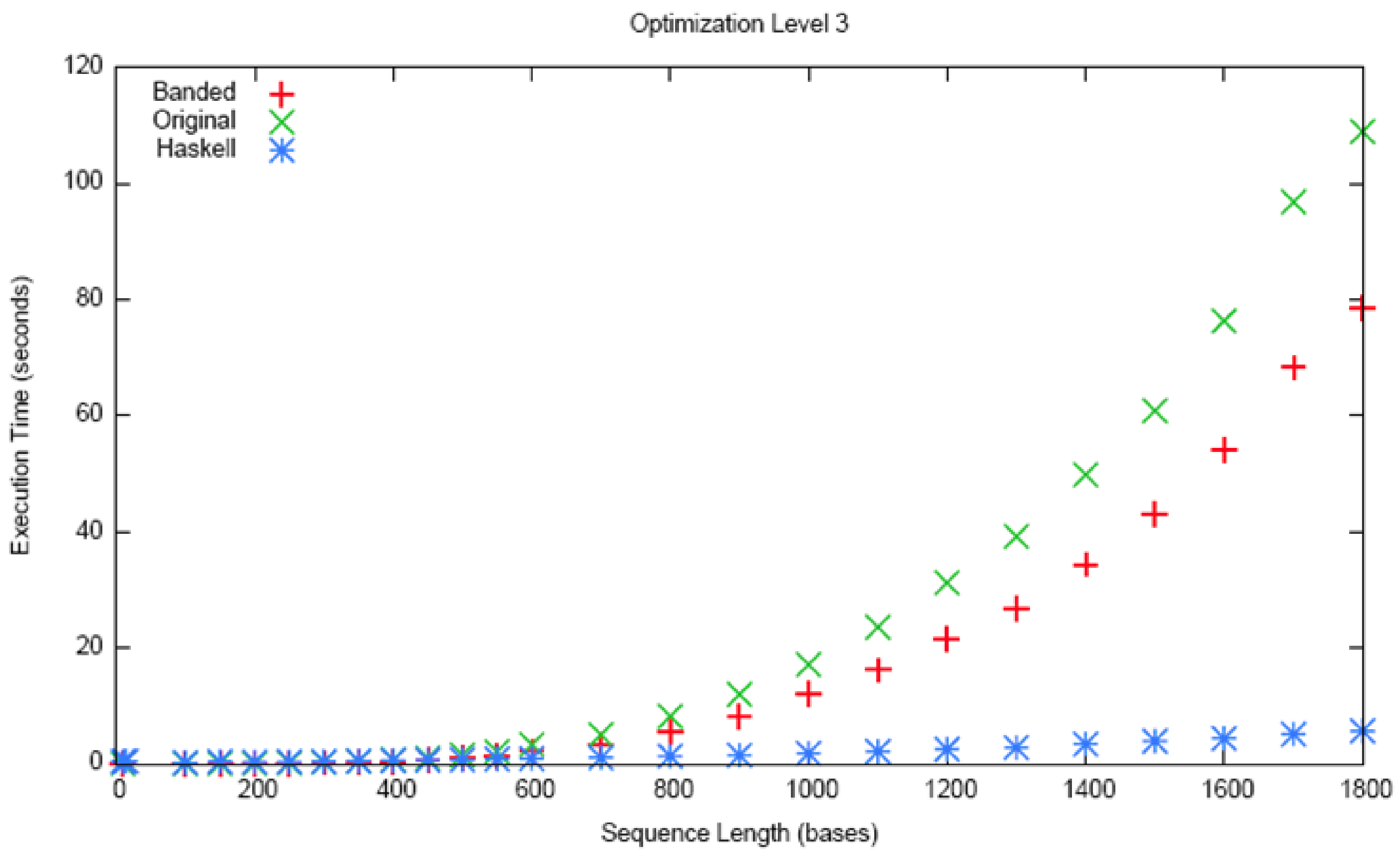


Figure 5. Execution time of implementations.

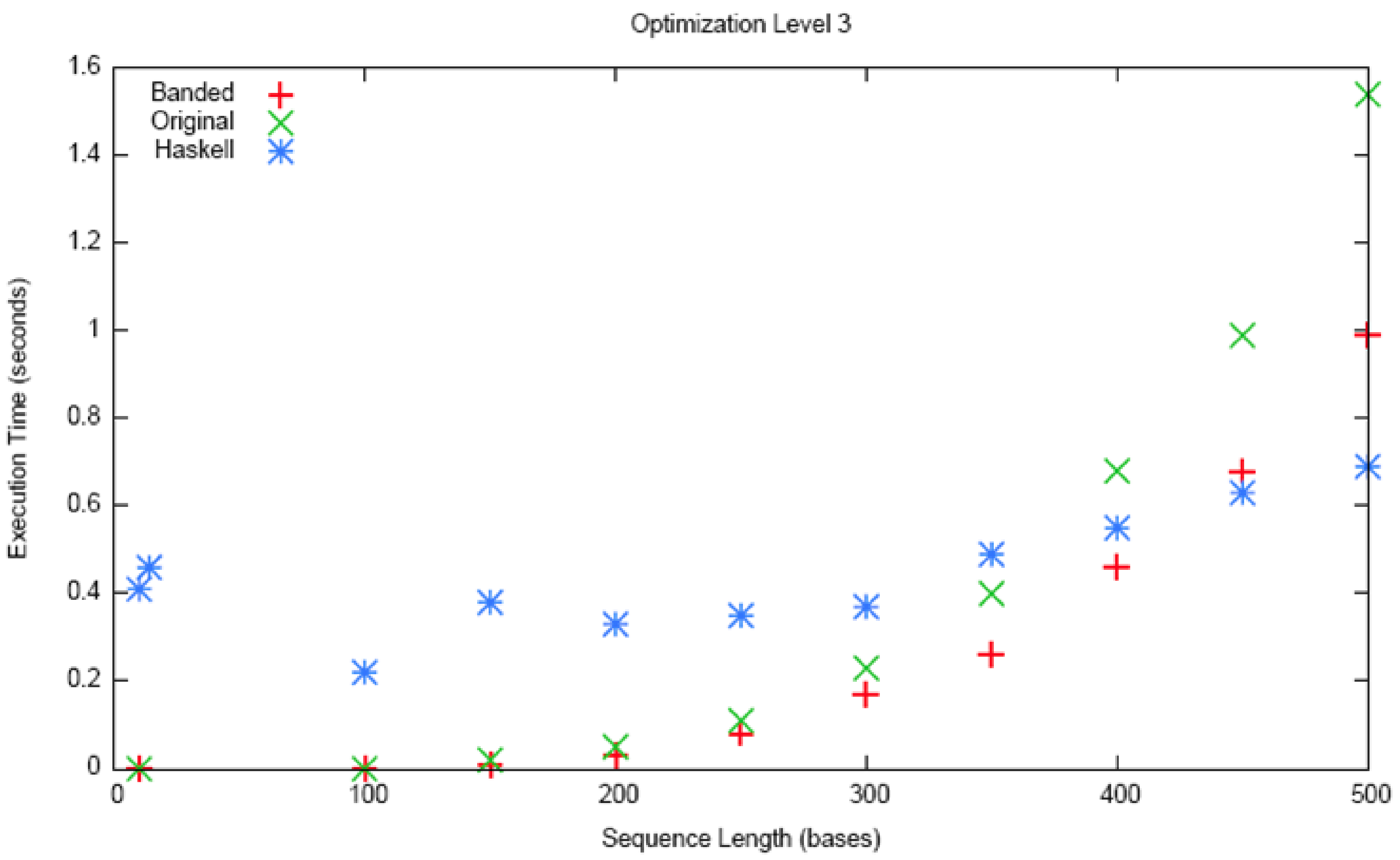


Figure 6. Execution time for smaller sequences. Crossover point occurs around sequence length of 420.

Haskell provides up to a 14x speedup over the banded C implementation for larger sequences. Both regressions lines are O(-n^3 + n^2). The speedup and regression lines can be seen in Figure 7.

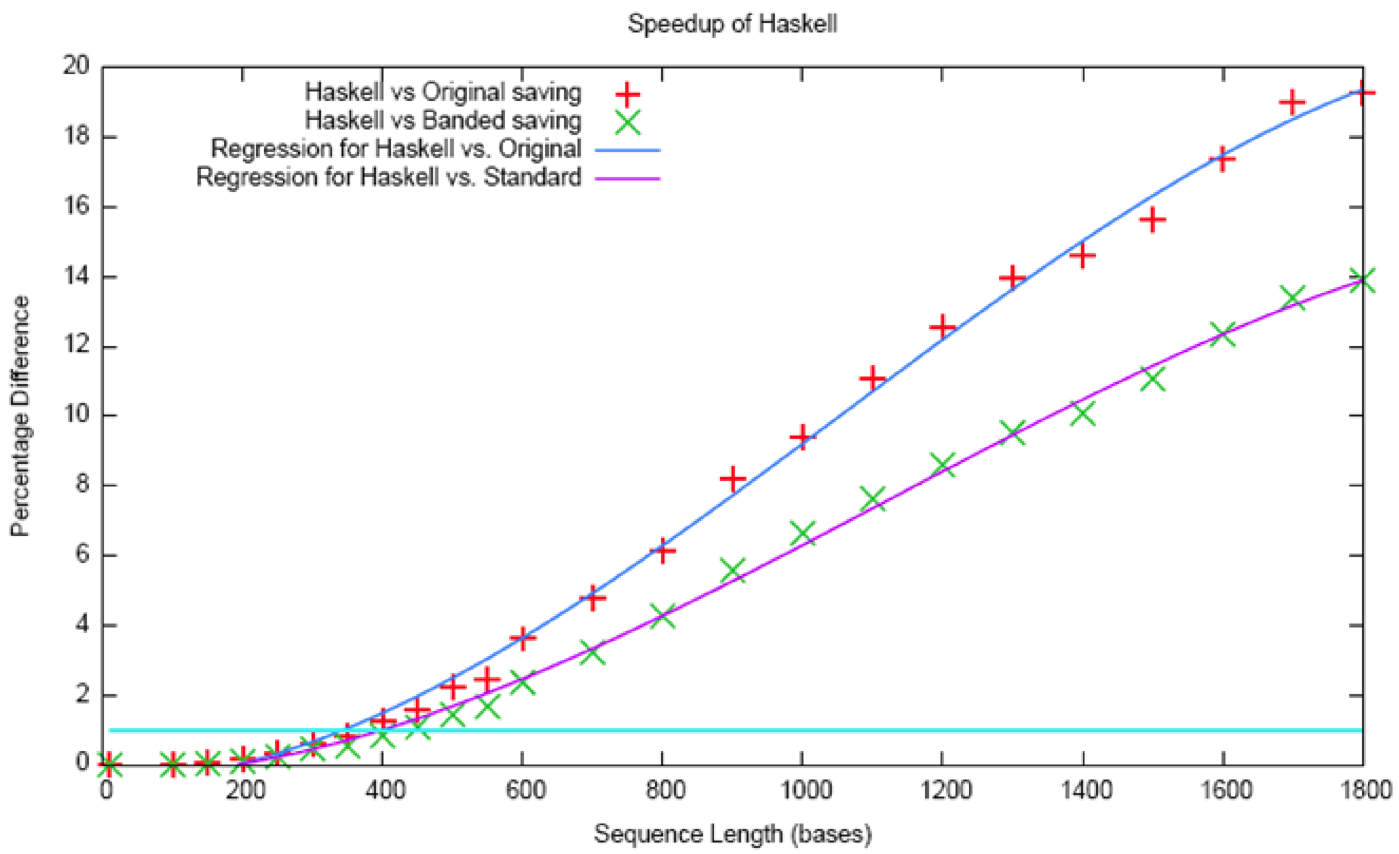


Figure 7. Speedup of Haskell. The regression line for the speedup of banded was y = -4.72*10^{9}x^{3} + 1.418*10^{-5}x^{2} - 0.495 and the regression line for speedup over the original algorithm was y = -2.802*10^{-9}x^{3} + 9.435*10^{-6}x^{2} - 0.328.

We initially believed that the significant speedup of the Haskell version over the C versions was due to the ephemeral nature of the cell objects. That is, since each cell must be dynamically created on an as needed basis, we believed that the overhead of calling malloc for each cell just prior to computation was the source of the performance gap. However, the execution profile for the banded C implementation for sequences of length 1800 bases, as seen in Figure 8, tells a different story: malloc constitutes 0% of the running time.

The real source of the performance penalty is the accounting of all of the objects once they have been created. Accessing the created objects (e.g., "getCell" and "getDiag") constitutes over 99% of the execution time. In our implementation, these objects are stored in a data structure that must then be referenced in order find the value that is needed. In Haskell, there is no need to have a data structure to store the values since all values are passed in as parameters to the function that uses them. It is not possible to use this approach of passing the values as function parameters in C since the depth of recursion needed to allow this is unsupported in C. This approach is possible in Haskell because lazy evaluation reduces the recursion depth.

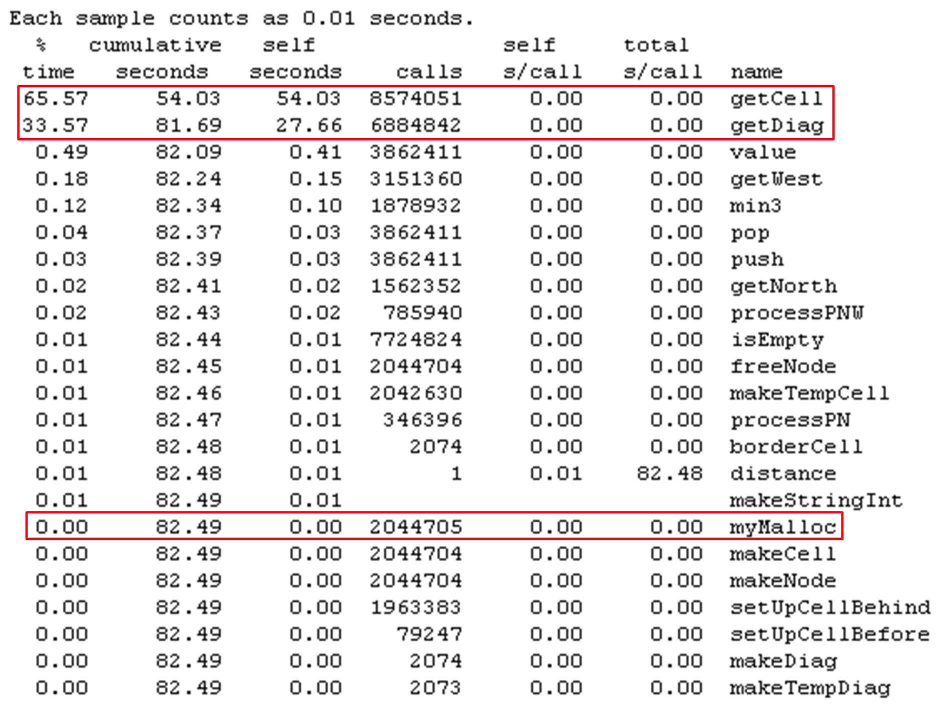

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 65.57     54.03    54.03  8574051    0.00     0.00  getCell
 33.57     81.69    27.66  6884842    0.00     0.00  getDiag
  0.49     82.09     0.41  3862411    0.00     0.00  value
  0.18     82.24     0.15  3151360    0.00     0.00  getWest
  0.12     82.34     0.10  1878932    0.00     0.00  min3
  0.04     82.37     0.03  3862411    0.00     0.00  pop
  0.03     82.39     0.03  3862411    0.00     0.00  push
  0.02     82.41     0.02  1562352    0.00     0.00  getNorth
  0.02     82.43     0.02   785940    0.00     0.00  processPNW
  0.01     82.44     0.01  7724824    0.00     0.00  isEmpty
  0.01     82.45     0.01  2044704    0.00     0.00  freeNode
  0.01     82.46     0.01  2042630    0.00     0.00  makeTempCell
  0.01     82.47     0.01   346396    0.00     0.00  processPN
  0.01     82.48     0.01     2074    0.00     0.00  borderCell
  0.01     82.48     0.01        1    0.01    82.48  distance
  0.01     82.49     0.01                             makeStringInt
  0.00     82.49     0.00  2044705    0.00     0.00  myMalloc
  0.00     82.49     0.00  2044704    0.00     0.00  makeCell
  0.00     82.49     0.00  2044704    0.00     0.00  makeNode
  0.00     82.49     0.00  1963383    0.00     0.00  setUpCellBehind
  0.00     82.49     0.00    79247    0.00     0.00  setUpCellBefore
  0.00     82.49     0.00     2074    0.00     0.00  makeDiag
  0.00     82.49     0.00     2073    0.00     0.00  makeTempDiag
```

Figure 8. Profiling results for banded C implementation of sequences of length 1800 bases.

From a maintenance point of view, the Haskell code is more expressive and concise. The Haskell code is shown in its entirety in Figure 4. The C code is over 1000 lines in length and therefore cannot be shown. Simply by being 37x larger, the C code contains a greater potential for bugs and requires more code to optimize and maintain.

5. Future Work

The next step in this work is to consider how optimizations and parallelization affects the large performance gap between the Haskell and C code. It would also be beneficial to consider larger sequences as it is likely that biologist will want to consider sequences of length greater than 1800 and to observe if there is a point where the speedup of Haskell over C remains constant or decreases. Another possible consideration is to compare sequences of different lengths since each individual alignment in this study had input sequences with equal length. Also, another computationally intensive biological problem, multiple sequence alignment, uses pairwise sequence alignment in many of the existing heuristics used to solve this problem. It would be interesting to determine if there was any time decrease if the standard pairwise alignment algorithm was replaced by this modified version

6. Conclusions

As suggested by the significant performance gap between the Haskell and C implementations, for highly recursive problems, Haskell can be result in an improvement over C without any tuning or optimizations. Granted, it is possible to write a C implementation that is faster than the Haskell implementation for this particular problem using a different algorithm; however, the purpose of this work is to consider the comparison of the two languages for this particular *algorithm type*, as opposed to the respective best algorithms for this particular problem. It is clear that Haskell performs better than C for highly recursive algorithms. We feel that this is a fair justification for our work because while we believe that one should always use the best tool for the job, comparing As can be seen by the length of the respective source codes, the Haskell code is both more efficient to write and easier to maintain. Thus, Haskell can provide both maintainability and performance for highly recursive problems and therefore be considered as another tool in the high-performance suite of sequence alignment tools.

References

1. "ProbCons: Probabilistic consistency-based multiple sequence alignment," C.B. Do, M. S. P. Mahabhashyam, M. Brudno, et al. Genome Research, 15 330-340, 2005.

2. mpiBLAST, http://www.mpiblast.org/

3. mpiHMMER, http://www.mpihmmer.org/

4. "A general method applicable to the search for similarities in the amino acid sequence of two proteins," S. B. Needleman, C. D. Wunsch , Journal of Molecular Biology 48 (3), March 1970.

5. "Lazy Dynamic-Programming can be Eager," L. Allison, Information Processing Letters 43(4) 407-212, Sept. 1992.