Parallel genomic alignment and clustering tools in Julia
Omar Abudayyeh
18.337 Parallel Computing
December 15th, 2015

1. Abstract

In this project, I implement in Julia two important bioinformatics algorithms: 1) Smith-Waterman alignment and 2) Markov clustering. For both algorithms, serial and parallel versions of the algorithms are implemented and compared in performance on distributed and shared memory systems up to 40 cores. Because of unique data dependencies in Smith-Waterman, a parallel algorithm can compute different sections of the alignment in parallel efficiently using Julia's dynamic task scheduler. For Markov clustering, I implement a form of parallel multiplication using spawned tasks on slices of the input matrices. For both algorithms, remarkable performance speedups are achieved using parallel implementations in Julia.

2. Introduction

2.1 Background

As the amount of data generated from sequencing increases, it is necessary to have accurate algorithms that are scalable and fast[1,2]. Protein and sequencing databases have swelled in size in recent years. For instance, there are now more than 23,000 bacterial genomes which are sequenced and publically available on NCBI (http://www.ncbi.nlm.nih.gov/). With so much data readily available, algorithms are needed to infer new relationships between genomic and protein sequences. The standard method for identifying new proteins depends on previously classified and analyzed proteins, as proteins can be grouped by similarities in their primary sequences and structure. It is common for proteins that are homologous, or evolutionary similar, to have strong conservation of certain residues across key protein domains. Therefore, it has become common practice for scientists to align new protein sequences against large databases of sequences in order to determine the structure and function of new proteins. Alignments can thus be used to cluster sequences into groups of related proteins, allowing scientists to then generate hypotheses about the potential function of novel proteins. These tools have been fundamental in the analysis of genomic and protein networks in many diseases, such as cancer and Alzheimer's. However, with sequencing technologies rapidly advancing, vast amounts of data sit unprocessed due to long analysis times. It is critical to have algorithms that are rapidly able to analyze large datasets. Julia offers a platform for implementing bioinformatics algorithms due to its combination of high performance computing with high-level usability, allowing for rapid implementation of parallel algorithms.

The goal of this project is to build a scalable set of alignment and clustering tools for genomic sequences. Typically, biologists who are confronted with a new dataset of sequences will compute all pairwise alignment scores and then use a clustering algorithm to understand the important relationships and identify similar sequences. Therefore, I aimed in this project to design and implement two key algorithms for genomic sequence analysis: 1) Smith-Waterman alignment for determining sequence similarity and 2) Markov clustering of nodes in biological networks. These

two algorithms are important for inferring relationships in large datasets and are part of the same workflow for clustering proteins. Because performance of these approaches is critical for large datasets, I am also presenting in this report the implementations of both serial and parallel versions and show the performance comparison of these different approaches. The implementations will use key aspects of Julia, including its shared memory methods and scheduling libraries.

2.2 Sequence alignment algorithms

A central problem in alignment algorithms is the tradeoff between accuracy and efficiency. Needleman-Wunsch[3] and Smith-Waterman[4] are two alignment algorithms that were originally designed to have high accuracy and to find the most optimal alignment between two sequences, but were computationally expensive. New algorithms, such as FASTA and BLAST, tried to ameliorate the high computational cost of these older algorithms but at the sacrifice of accuracy[1]. However, Smith-Waterman is still an important algorithm because many newer algorithms depend on it to help refine the alignment and improve accuracy. Therefore, developing a fast and scalable version is broadly useful for many existing alignment tools.

The Smith-Waterman algorithm was developed in 1981 as a tool for computing optimal local alignment[4]. Instead of determining the entire alignment of two input sequences, Smith-Waterman focuses on finding the most similar region between two sequences, which is biologically meaningful since the ends of proteins tend to be non-conserved due to high mutation rates at their ends.

As a pairwise sequence algorithm, Smith-Waterman uses a scoring function, F, which assigns different scores to various alignments of any two input sequences. The function is designed such that the optimal alignment emerges with the highest score. The algorithm takes advantage of the fact that the sum of the scores of aligning two subsequences equals the total alignment score. This allows for an iterative calculation of the optimal score by growing the alignment from the beginning of each input. This implies that the score at residue *i* in sequence *x* and residue *j* in sequence *y* is the sum of the score of residue *i* and *j* and the score of subsequences preceding these residues. The score of any two residues *i* and *j* is based on one of three possibilities: 1) residues *i* and *j* are a match or mismatch, 2) residue *i* aligns to a gap, or 3) residue *j* aligns to a gap and this may mathematically be described as follows:

1) $F(i,j) = s(i,j) + F(i-1,j-1)$where $s(i,j) = m$ if $x_i = y_i$; $s(i,j) = -s$ otherwise

2) x_i aligns to a gap and "$F(i,j) = -d + F(i-1,j)$"

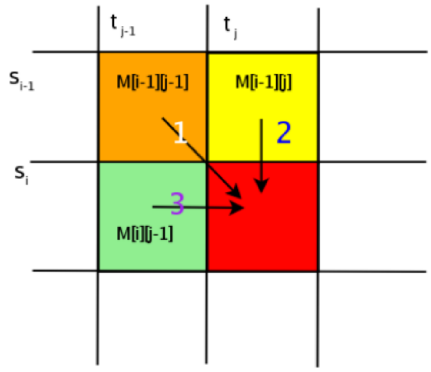
3) y_i aligns to a gap and "$F(i,j) = -d + F(i,j-1)$"

where *m* is the matching score, *s* is the mismatch penalty, and *d* is the penalty for a gap. By taking the maximum of these three possibilities at any given position (Fig. 1a), we can determine the alignment score for all possible subsequences as:

$$F(i,j) = score(i,j) + max\begin{cases} F(i-1,j-1) \\ F(i-1,j) \\ F(i,j-1) \\ 0 \end{cases}$$

In order to filter out regions of dissimilarity and to focus on optical local alignments, it is important to never let an alignment score become negative. Upon determining the optimal alignment, if a zero is encountered, the local alignment is therefore terminated. This allows the algorithm to only focus on regions with high similarity and zeros therefore partition areas of high similarity. The algorithm begins by creating a matrix that is *mxn* in size where *m* is the length of sequence 1 and *n* is the length of sequence 2 (Fig. 1b). Thus, horizontal and vertical axes correspond to each of the input sequences. Each cell (i,j) will be filled with the score F(i,j). The matrix is initialized with a top most row (0,j) and left most column (i,0) of zeros . The algorithm starts at cell (1,1) and begins filling the matrix from left to right and then top to bottom. Once the matrix is complete, a traceback is performed by identifying the maximum cell in the entire matrix and then following the maximum path from right to left and bottom to top until a zero is encountered. At this point, the alignment terminates. As shown in Fig. 1b, the alignment matrix is generated for two input sequences (ATGCATGCATGC and ATGGGCATG) with a match score of 2 and a mismatch/gap score of -1. The optimal alignment path is shown in red by tracing back from 13 to 0 and the optimal local alignment is output as shown in Fig. 1c. As can be clearly seen in the traceback path, when the path travels horizontally there is a gap in sequence y, and when it travels vertically there is a gap in sequence x. When the path travels diagonally, it is accepting the alignment as a match or mismatch.

a



b

	^	A	T	G	C	A	T	G	C	A	T	G	C
^	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	**2**	1	0	0	2	1	0	0	2	1	0	0
T	0	1	**4**	3	2	1	4	3	2	1	4	3	2
G	0	0	3	**6**	5	4	3	6	5	4	3	6	5
G	0	0	2	5	**5**	4	3	5	5	4	3	5	5
G	0	0	1	4	4	**4**	**3**	**5**	4	4	3	5	4
C	0	0	0	3	6	5	4	4	**7**	6	5	4	7
A	0	2	1	2	5	8	7	6	6	**9**	8	7	6
T	0	1	4	3	4	7	10	9	8	8	**11**	10	9
G	0	0	3	6	5	6	9	12	11	10	10	**13**	12

c

Figure 1: a) Each cell's score is determined by finding the maximum of the three cells adjacent to it (top, left, and diagonally). b) An example score matrix for the alignment of two sequences as shown. c) The alignment discovered from tracing back (red) the score matrix in (b).

This dynamic programming algorithm has an overall runtime cost of O(MN) due to the cost of traversing every element of the score matrix.

2.3 Genomic clustering algorithms

Although many clustering algorithms exist, recently the Markov clustering algorithm (MCL), which was originally developed as a graph-clustering tool, has been adopted for a wide range of biological network applications[5]. It has been particularly successful for identifying families of interacting protein in large protein networks. Through many applications, the MCL algorithm has been shown to be effective, fast, and quite robust, making it an attractive choice for biological interaction networks. It is based on two simple operations, expansion and inflation, applied to the Markov matrix, M, of the associated network graph, G. The algorithm requires a normalized Markov matrix, which is generated by normalizing all the columns of the adjacency matrix of G. The clustering then begins by iteratively performing expansion and inflation operations on the normalized M matrix. Expansion is used to stimulate flow throughout the graphs. Every expansion operation performs a random walk. After enough iterations, the flow equilibrates and the structure of the graph emerges. Inflation operations help promote this process by strengthening flow where it is strong and weakening it where it is weak. Eventually, convergence is achieved where regions of clusters are marked by strong flow and are separated from other clusters by regions of no flow.

MCL expansion is performed by taking the *p*th power of the matrix M as follows:

$$Exp(M) = M^p$$

By default, p=2. The MCL inflation operation takes a matrix $M \in \mathbb{R}, M \geq 0$ and a power $r \in \mathbb{R}, r > 0$. The inflation operator Γ_r: $\mathbb{R}^{mxn} \rightarrow \mathbb{R}^{mxn}$ to M with power coefficient r as described below:

$$(\Gamma_r M)_{ij} = {\left(M_{ij}\right)^r} \Big/ {\sum_{k=1}^{m}\left(M_{kj}\right)^r} ; i = 1 \dots m, j = 1 \dots n.$$

This operation essentially performs the Hadamard product of the matrix M to the power r and automatically normalizes for the next round of expansion. As the algorithm iterates between expansion and inflation, an idempotent matrix will eventually be resolved with clusters within it. The algorithm terminates when a given chaos threshold is reached and no significant changes within the matrix occur during subsequent expansions. The chaos is defined as:

$$chaos = \max(\Gamma_r M) - \min(\Gamma_r M).$$

When chaos is ~0, the algorithm terminates and analyzes the underlying structure of the graph to output clusters.

The cost of MCL is $O(N^3)$ where N is the number of nodes or vertices in the network graph. This cost is due to the multiplication of two matrices of dimension N. The inflation step only takes $O(N^2)$. So, most of the cost and speed up benefits can be achieved by improving the expansion step. While convergence is not proven, experimentally it is shown to be between ~10-100 steps and this result has been very robust.

2.4 Parallelization of Smith-Waterman

Improved performance of the Smith-Waterman algorithm can be achieved through parallelization, which can be accomplished by taking advantage of the unique data dependencies in the DP score matrix (Fig. 2a)[6,7]. On a single processor, the cells of the matrix are evaluated sequentially, but a parallel implementation can efficiently compute independent cells from the score table in parallel. The score function of the algorithm is structured such that each cell only depends on the cell above it, below it, and to the upper-left. This means that cells from each anti-diagonal can be computed in parallel (up to min(m,n)) since they are all independent of each other, allowing for the entire score matrix to be computed in m+n-1 passes by processing each anti-diagonal sequentially[6]. Because the smaller anti-diagonals will not efficiently use all the available workers, there can be stall equal to p(p-1), where p is the number of processes available (Fig. 2b). Because Julia has significant overhead in submitting work to processes, the gain in parallelization may be offset by the communication and overhead time. This usually means that the work each process is given must be computationally complex enough to make the overhead negligible. Since computing each cell is a simple calculation, it may be necessary to split the DP matrix into a grid of many cells (e.g. grid units of 50x50 cells). Then, these grid units can be processed by parallel computing the anti-diagonals of grid units, where each process will then receive a 50x50 chunk of cells to compute. This approach was also taken as a precaution for the original approach not resulting in a significant speed enhancement due to overhead.

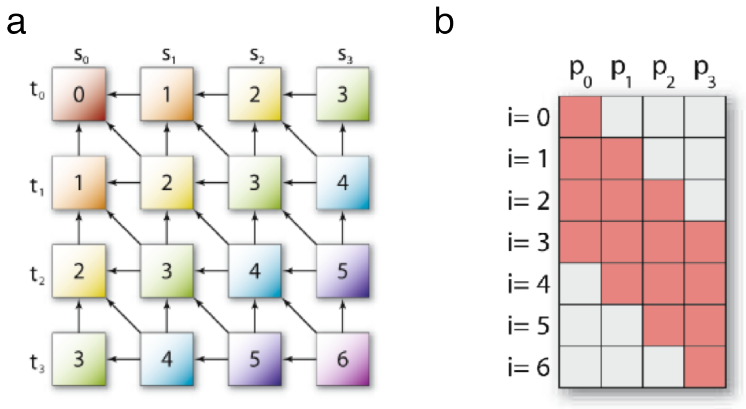


Figure 2: a) Depiction of the data dependencies in the Smith-Waterman score matrix. Arrows indicate dependencies. Cells colored with the same color are independent of each other (termed anti-diagonal) and can be computed in parallel. b) Geometrical representation of loading processors *p* with anti-diagonal *i.* Red blocks indicate cells from anti-diagonals that are sent to a given processor *p.* The grey colored cells indicate inefficient use of available processors due to the smaller anti-diagonals. Images adapted from Liu et al.[6]

2.5 Parallelization of Markov clustering

Parallelization of Markov clustering can be accomplished by parallelizing the expansion operator, which is the most costly step of MCL. This can be accomplished by splitting the work of matrix multiplication across multiple workers[8]. The parallelization of matrix multiplication $M^2 = M_1 M_2$ is performed by assigning worker *i* to perform multiplication of M_1 against a slice of *j* columns from M_2. The resulting *m x j* matrices are concatenated together to form the final matrix product. The reduction can be performed with distributed memory by communicating the position of the slices from the original matrix and organizing the results accordingly. Using shared memory, this

can be performed with less overhead by directly updating the matrix cells in a pre-allocated empty matrix.

3. Genomic alignment and clustering in Julia

3.1 Features of Julia

The Julia programming language offers convenient features for scheduling tasks in parallel. The spawn macro allows for dynamic scheduling of parallel activity by automatically sending work to available workers. Additionally, Julia allows for the synchronous parallelization of tasks, which is important for the Smith-Waterman algorithm, as the anti-diagonals must be sequentially processed. Another useful aspect of Julia is its integration of methods to deal with shared memory systems. Because these bioinformatics algorithms tend to deal with large matrices, shared memory data structures allow for faster computation since there is no communication overhead. Due to the high-level nature of the language, it is quite easy to implement a variety of parallel designs for these two algorithms with distributed and shared memory data.

3.2 Smith-Waterman implementation

Serial version

The standard implementation for Smith-Waterman is to iteratively fill in each row of the DP score matrix from left to right (Fig. 3). The pseudocode shown in Fig. 3 searches for the maximum score of the three possible alignments prior to the current cell and then saves the best score. It also keeps track of which of these options is the maximum in order to perform the traceback at the end.

```
for i = 2:row
        for j = 2:col
            scores = []

            push!(scores,matrix[i,j-1]+indel)
            push!(scores,matrix[i-1,j]+indel)
            if seq1[j] == seq2[i]
                push!(scores,matrix[i-1,j-1]+match)
            else
                push!(scores,matrix[i-1,j-1]+indel)
            end

            val,ind = findmax(scores)
            if val < 0
                matrix[i,j] = 0
            else
                matrix[i,j] = val
            end

            path[i,j] = pathvals[ind]

        end
    end
```

Figure 3: Pseudocode depicting how to perform serial Smith-Waterman. **"matrix"** is the DP score matrix and **"path"** is the matrix keeping track of the paths followed from subalignment to subalignment.

At the end of the algorithm, the *matrix* and *path* tables are output such that the traceback can be performed to output the optimal alignment.

Parallel version

Dynamic and synchronous scheduling can be achieved in Julia with the sync and async macros. Async allows for a local process to send out all work to the worker processes while sync forces the local process to wait until all tasks are completed within a given "sync" block. This feature is incredibly important for the parallel implementation of Smith-Waterman as each anti-diagonal can only be computed when the diagonal before it is finished. This offers a powerful mechanism to send each anti-diagonal out to workers iteratively for processing. Additionally, because Julia has shared memory matrices readily available, it becomes easy to update cells of the matrix with little communication overhead between workers and the local process.

A pseudocode parallel version modifying the serial version is shown in Fig. 4. The main flow of this approach is to iterate through each anti-diagonal and send each cell through a helper function *get_score()* to each worker. The results of each worker are then updated within the worker using shared memory matrices.

```
for j = 2:col
    jcol = j
    irow = 2
    @sync begin
        count = 1
        w = workers()
        while jcol > 1 && irow < row + 1
                if seq1[jcol] == seq2[irow]
                    equal = true
                else
                    equal = false
                end

                if count > length(w)
                    count = 1
                end

                @async remotecall_wait(w[count],shared_get_score!)
                jcol -= 1
                irow += 1
                count += 1
        end
    end
end
```

Figure 4: Parallel implementation of Smith-Waterman. Shown is the first of two for loops that sends the first half of anti-diagonals to workers.

An additional modification to this parallel algorithm to enhance performance is to split the matrix into a grid. An issue with Julia is that there is significant overhead involved in sending tasks to the workers and if the computations made on each process are relatively simple compared to the overhead cost, then it is not useful to parallelize as the overhead cost dominates. Therefore, in case

overhead becomes an issue, the Smith-Waterman algorithm can be enhanced by splitting the DP matrix into a grid with *pxp* units, where *p* is the number of workers available. This will involve sending grid units of size *r x r* cells, where $r = n/p$ and the DP matrix is of size *nxn.* In this approach, anti-diagonals of grid units are sent to each worker and thus each worker performs more work for each anti-diagonal iteration. Pseudocode of this implementation is shown in Fig. 5.

```
lwl = length(wl)
  for j = 1:lwl
      jcol = j
      irow = 1
      @sync begin
          count = 1
          w = workers()
          while jcol > 0 && irow < lwl+1
                  if count > length(w)
                      count = 1
                  end

                  startx = 1+(irow-1)*div(n,p)+1
                  starty = 1+(jcol-1)*div(n,p)+1

                  endx = irow*div(n,p)+1
                  endy = jcol*div(n,p)+1

                  @async remotecall_wait(w[count],grid_get_score!,startx,starty,endx,endy)
                  jcol -= 1
                  irow += 1
                  count += 1
          end
      end
  end
```

Figure 5: Parallel grid implementation pseudocode of Smith-Waterman. Startx/starty and endx/endy denote the two corners of each grid unit that is sent to a single worker.

3.3 Markov clustering implementation

Serial version

A serial version of Markov clustering essentially performs expansion and inflation operations iteratively until some convergence threshold is met. The pseudocode outlined in Fig. 6 shows an example of how MCL can be implemented. Once convergence of the Markov matrix is met, the underlying structure can be analyzed for clusters, which are essentially denoted through non-zero edges.

```
function mcl(M,p=2,r=2,maxl=25, mult=1)
    count = 0
    for i = 1:maxl
        println(i)
        M = inflate(M,r)
        M = expand(M,p)
        if stop(M,i)
            break
        end
        count = count + 1
    end
    return(M)
end
```

Figure 6: Pseudocode for serial version of MCL. *M* is the Markov matrix; *p* is the expansion power; *r* is the inflation parameter; *maxl* is the max number of iterations allowed; *mult* is a constant added to the diagonal to add self-loops to each node and promote seeding of clusters.

Parallel version

Although Julia matrix multiplication is already parallel through the BLAS library, further speed up can be achieved by using some number of workers for further subdividing the work of matrix multiplication. Julia's @spawn macro offers the ability to compute jobs in parallel and to wait for these jobs to complete when performing work that depends on the results of each remote call. In this manner, the expansion operation can be parallelized and inflation can be delayed until all chunks of the matrix multiplication are computed in parallel and reduced to the final product (for the distributed case). Both a distributed and shared memory version of this algorithm will be implemented and compared. The shared version updates matrix cells within each worker and each matrix multiply is kept in sync using the sync and async macros. A pseudocode example of the shared memory parallelized matrix multiplication is shown in Fig. 7

```
@everywhere function mymatmul!(n,w,sa,sb,sc,p)
    range = 1+(w-1) * div(n,p) : (w) * div(n,p)
    sc[:,range] = sa[:,:] * sb[:,range]
end


function sharedmult(n,p,sa,sb,sc)
  @sync begin
      for (i,w) in enumerate(workers())
          @async remotecall_wait(w, mymatmul!, n, i, sa, sb, sc,p)
      end
  end
  return sc
end
```

Figure 7: Pseudocode for parallelized matrix multiplication using shared matrices.

4. Performance results

4.1 Testing Conditions

For testing performance of my code, I used the 80-core julia.mit.edu shared memory machine with both randomized matrices and real biological data. For both algorithms, I tested a variety of dataset sizes and a range of cores. To ensure optimal performance, run times were recorded after initially compiling the Julia code since subsequent executions usually compute faster. Additionally, distributed and shared memory versions of the algorithms are tested. All raw data can be found in the Appendix. The most interesting results are presented in the following sections.

4.2 Parallelized Smith-Waterman optimization

Given the communication overhead of sending work to processors, it might be expected that the parallel version of Smith-Waterman sending a single cell to each worker would be slower than the serial version. Given this consideration, only shared memory arrays were used to minimize communication workloads. In order to test the algorithm's performance, random DNA sequences of lengths 40, 200, 1000, 3000, 5000, and 7500 were generated. While Smith-Waterman can be used for protein sequence alignment, for testing purposes it is easier to use DNA sequences since the protein version of the algorithm has a more complex scoring system for matches versus mismatches. As expected, the parallel versions using 16 or 32 cores were three to four orders of magnitude slower (Fig. 8). While the parallel versions failed to improve performance, the Julia code itself was 2.5x faster than the Python code. This experimentation with Smith-Waterman implementation reveals the power of Julia to generate faster runtimes simply by porting code over from other languages, such as Python.

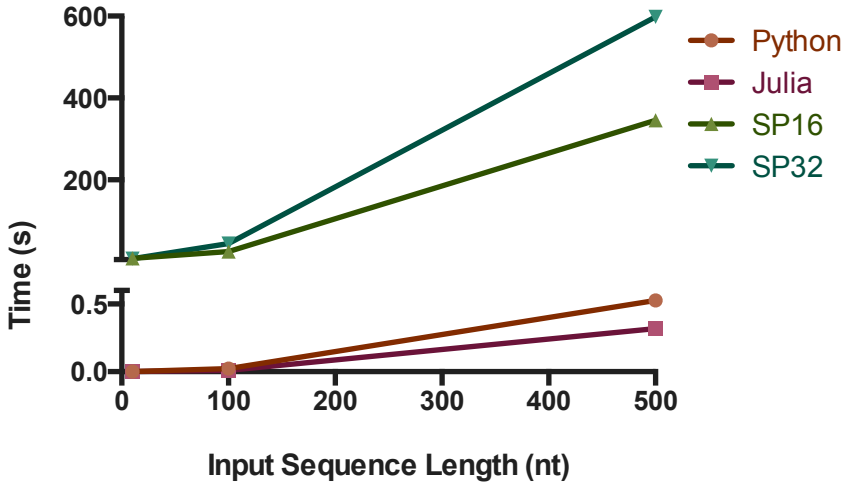


Figure 8: Performance of parallel Smith-Waterman implementation using random DNA sequences of varied lengths. SP16 and SP32 refer to parallelization on 16 and 32 cores, respectively. The Python and Julia executions are performed with the serial version of the algorithm.

It was clear that communication overhead was the issue with the original parallel implementation of Smith-Waterman. Every remote call was significantly slowing the code since the computation of each task was so little and it was thus faster to compute in serial. I, therefore, decided to implement the grid version of Smith-Waterman in order to send larger chunks of the DP matrix to each worker. For testing, I used sequences of length 40, 200, 1000, 3000, 5000, 7500, 10000, and 15000 and the following numbers of cores: 2, 4, 5, 10, 20, and 40. After implementing this new algorithm, it was clear that performance was much improved for larger sequence inputs (Fig. 9). For sequences less than 1000nt, the serial version was superior regardless of the cores used for parallel computation. Because parallel computation only becomes efficient when the work being sent to each processor is sufficiently large that overhead becomes negligible, it is understandable that performance of the parallel grid version correlates with sequence input length. Above sequence lengths of 1000nt, a maximum speedup of 6.3x is achieved. A couple interesting trends emerge from the performance curves in Fig. 9. Two workers seem to be optimal for most

conditions, suggesting that communication overhead is still an issue for most lengths tested. Eventually, as very long sequence inputs are used, the number of processes does not matter and the speed up converges to ~4x. Thus, to perform Smith-Waterman alignment optimally, it would make sense to operate the serial version for sequence inputs of length less than 1000nt and the parallel grid algorithm for longer sequences.

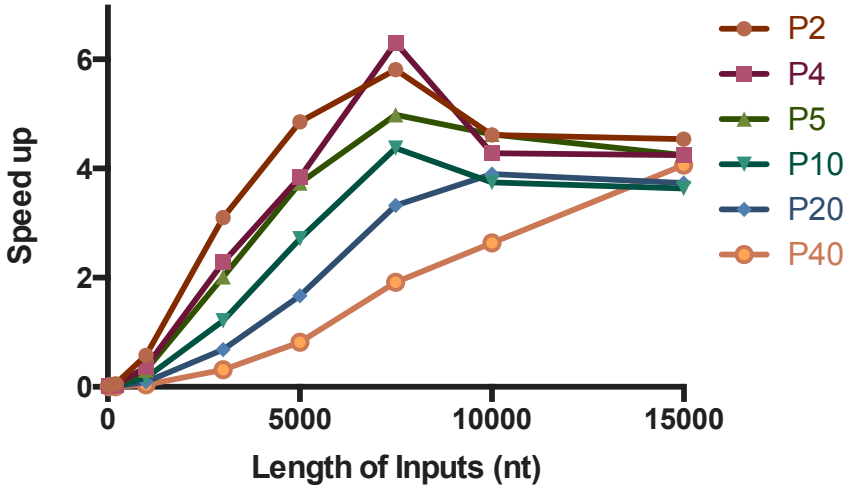


Figure 9: Performance of the parallel grid implementation of the Smith-Waterman algorithm. Runtimes are compared to the serial Julia execution.

4.3 Parallelized Markov clustering optimization

Improving the expansion operations in MCL involves parallelizing matrix multiplication. The implemented algorithm was tested on random *nxn* matrices (meaning *n* nodes in the graph) of sizes n equal to 1600, 3200, 4800, and 6400 and the following numbers of cores: 2, 4, 8, 16, 32, and 40. The distributed and shared versions of the algorithm were tested using regular or shared arrays, respectively. While the distributed version achieved a maximum speed up of 5.6x, the shared algorithm achieved an optimal 26x speedup (Fig. 10). This difference is likely due to the work required in shuttling chunks of data back and forth to the workers. It is also clear that the algorithm scales well with larger datasets as the speed up is quite improved for larger matrices.

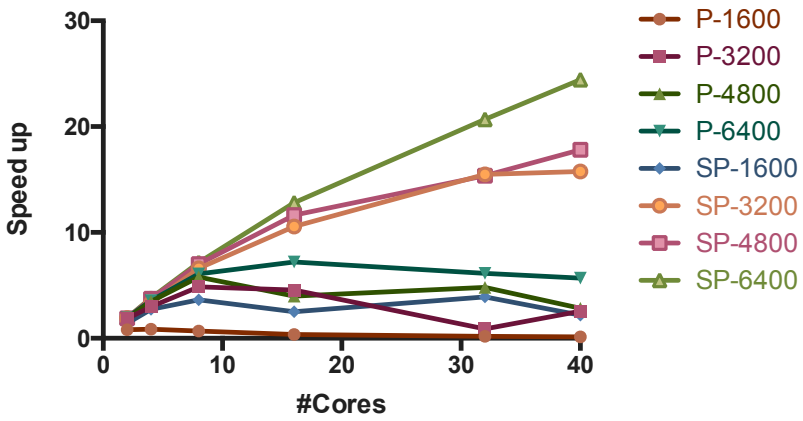


Figure 10: Performance of parallelized matrix multiplication on a varying number of cores.

With parallel matrix multiplication optimized, I implemented two versions of MCL for distributed and shared arrays. The algorithm was tested on matrices of dimensional sizes of n=1600, 2400, and 3200 (Fig. 11). The MCP performance for shared arrays scaled linearly with the number of cores and the same speed ups were observed regardless of dataset size. The equivalence in speed up for the MCL despite differences in the matrix multiplication runtimes is due to longer convergence times for the larger matrices. The distributed version of MCL performed better for larger datasets and displayed no improvement in performance beyond 30 cores. This is likely due to the need for more communication work as the number of workers increases.

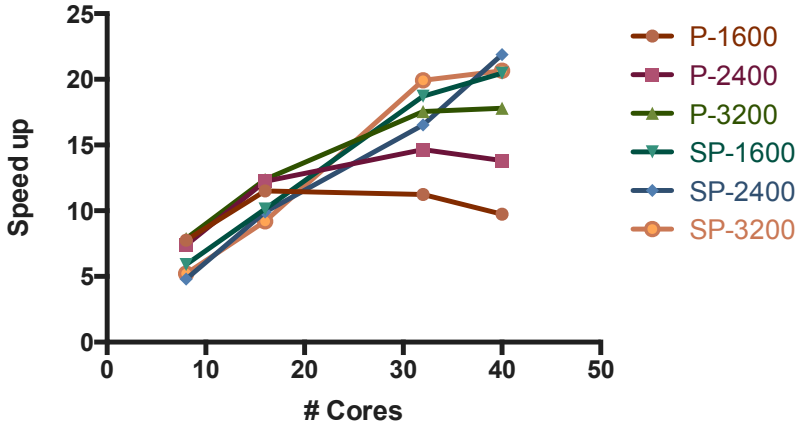


Figure 11: The performance of MCL on a varying number of cores.

4.4 Markov clustering on sparse protein network

I decided to perform a final test of the MCL algorithm using a real biological dataset in order to test performance in a "real world" scenario and to evaluate accuracy. Costanzo et. al. generated an interaction map of 5.4 million gene-gene pairs from the budding yeast, *Saccharomyces cerevisiae*, represented by a network of 3,886 nodes (proteins) and 15,100,996 edges (correlations between proteins as denoted in the interaction dataset)[9]. Actual biological networks are much sparser than the dense random networks tested in the previous sections. This particular dataset has a sparsity of

26%. When the algorithm was applied to this dataset, the performance observed scales linearly for the shared version and saturated as before at 30 cores for the distributed version. For the shared memory algorithm, a maximum speedup of 27x was achieved, similar to what I saw for the random test cases. Additionally, the accuracy of the algorithm was sufficiently high as 714 clusters were generated with an average size of 6.45 proteins, similar to what Costanzo et al found.[9]

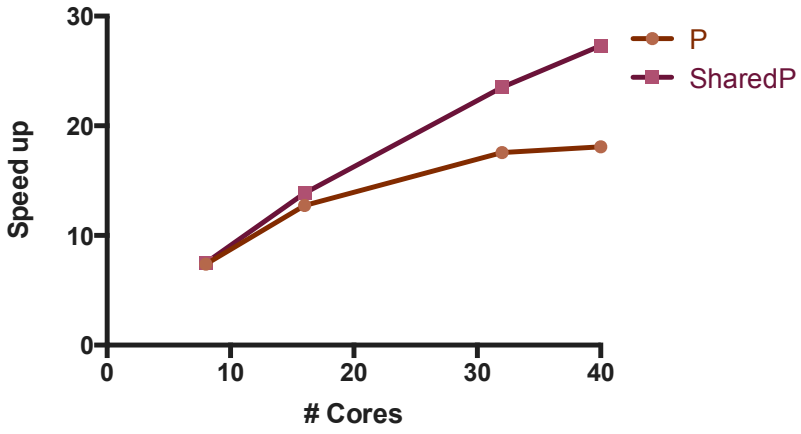


Figure 12: The performance of MCL on a biological network from Costanzo et al.

5. Future Directions

5.1 Stacking of Smith-Waterman

In this project, I implemented a faster parallel version of Smith-Waterman optimized for the alignment of a single sequence against another sequence. The representation of worker load can be geometrically represented as a parallelogram (Fig. 13a)[6], since smaller anti-diagonals will not adequately use all available processes. This waste in stalled processors can be ameliorated by filling the available processors with work from the alignment of other queries against the target sequence. Usually, biologists will attempt to align many inputs against a target sequence or database. Therefore, given the many comparisons that must be made, an even faster Smith-Waterman algorithm can be developed for large database queries by interweaving the anti-diagonals from different queries (Fig. 13b). It would be worth investigating how well a stacked Smith-Waterman would be perform using the Julia language.

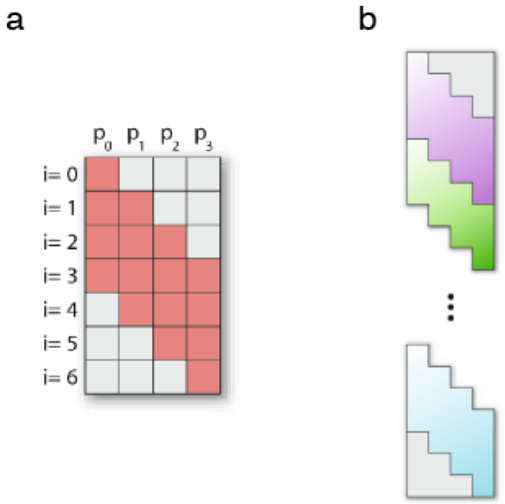


Figure 13: a) The geometrical representation of tasks sent to each processor p_i from anti-diagonal *i.* b) Stacking of worker loads between different sequence alignments to more optimally use all available workers. Imaged adapted from Liu et al.[6]

Additional improvements in performance can likely be achieved by using GPU libraries available in Julia to perform the alignment computations on the GPU.

5.2 Other alignment algorithms

Further work should focus on extending the parallelism to other alignment algorithms. Smith-Waterman is usually used in conjunction with other less accurate algorithms such as FASTA. FASTA is a k-mer based alignment tool that uses hashing to improve speed at the expense of accuracy[1,2]. Modern day alignment tools will combine this k-mer hashing search approach to quickly identify local regions of similarity and then Smith-Waterman to perform more accurate local alignments. Because these pipelines are generally used on many input sequences and large target sequence databases, they can certainly be improved using the parallel approaches described in this work.

5.3 Markov clustering using GPU

A few more optimizations can be made to the Markov clustering approach described here. As biological datasets continue to grow in size, it will be advantageous to reduce the memory required of these algorithms. Usually, biological networks are quite sparse (20-30%) since most proteins only interact with a small subset of other proteins. Memory improvements can be achieved by using a sparse matrix format to represent the network matrices, such as the sparse column format. I briefly explored using Julia's Compressed Sparse Columns (CSC) format for representing these matrices and used a parallel sparse shared memory matrix package to perform the computations. Unfortunately, this version was quite slower perhaps due to the overhead of having to compute a single column and row multiplication at a time and thus the large number of worker remote calls that have to be made, which increases the overhead time. Recently, an algorithm for MCL was published that adopted a sparse column matrix version of MCL for the GPU using CUDA.[8] It might be

worth implementing this approach using Julia's CUDA package to further improve both performance and memory usage.

5.4 Code optimization

Certain optimizations in the code are needed for further development of these packages. Particularly, Smith-Waterman was designed and implemented for DNA sequences specifically, but protein alignment is equally as important. Further iterations should add support for protein sequences using more complex scoring functions such as BLOSUM. Additionally, further experimentation with the MCL package is necessary to understand how and to what extent changing the parameters of expansion and inflation affect performance.

5.5 Availability of package

Code can be found on github: https://github.com/oabudayyeh/JuliaAlignmentToolbox/.

6. References

1. Li, H. & Homer, N. A survey of sequence alignment algorithms for next-generation sequencing. *Brief Bioinform* **11**, 473-483 (2010).
2. Flicek, P. & Birney, E. Sense from sequence reads: methods for alignment and assembly. *Nat Methods* **6**, S6-S12 (2009).
3. Needleman, S.B. & Wunsch, C.D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol* **48**, 443-453 (1970).
4. Smith, T.F. & Waterman, M.S. Comparison of Biosequences. *Advances in Applied Mathematics* **2**, 482-489 (1981).
5. Enright, A.J., Van Dongen, S. & Ouzounis, C.A. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Res* **30**, 1575-1584 (2002).
6. Liu, Y., Huang, W., Johnson, J. & Vaidya, S. GPU accelerated Smith-Waterman. *Lect Notes Comput Sc* **3994**, 188-195 (2006).
7. Liao, H.Y., Yin, M.L. & Cheng, Y. A parallel implementation of the Smith-Waterman algorithm for massive sequences searching. *Conf Proc IEEE Eng Med Biol Soc* **4**, 2817-2820 (2004).
8. Bustamam, A., Burrage, K. & Hamilton, N.A. Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. *IEEE/ACM Trans Comput Biol Bioinform* **9**, 679-692 (2012).
9. Costanzo, M.*, et al.* The genetic landscape of a cell. *Science* **327**, 425-431 (2010).

7. Appendix

Table 1: Run times for parallel matrix multiplication serially in Python and Julia (on one thread and multithreaded) and in parallel using distributed memory (P) and shared memory (SP) on 2, 4, 8, 16, 32, and 40 cores.

Size	Python (1 thrd)	Julia	Julia (1 thrd)	P2	P4	P8	P16	P32	P40	SP2	SP4	SP8	SP16	SP32	SP40
1600	1.09	0.77	1.64	1.99	1.92	2.38	4.38	8.96	11.23	1.17	0.60	0.45	0.65	0.42	0.75
3200	8.62	0.66	8.67	4.68	2.91	1.78	1.90	9.90	3.37	4.56	2.62	1.31	0.82	0.56	0.55
4800	28.91	1.88	28.87	15.28	8.30	4.98	7.25	5.99	10.15	15.22	7.71	4.11	2.48	1.88	1.62
6400	68.99	4.45	68.93	35.66	19.40	11.32	9.57	11.25	12.13	35.80	18.03	9.62	5.38	3.33	2.82

Table 2: Run times for MCL using random datasets serially in Python and Julia (on one thread and multithreaded) and in parallel using distributed memory (P) and shared memory (SP) on 8, 16, 32, and 40 cores.

Size	Python	Python (1 thrd)	Julia	Julia (1 thrd)	P8	P16	P32	P40	SP8	SP16	SP32	SP40
1600	9.88	331	28.2	397	51.2	34.5	35.3	40.8	67.1	39.2	21.2	19.4
2400	60.93	1042	83.3	1155	156.3	94.3	78.8	83.6	240.1	116.5	69.9	52.8
3200	203.3	2605	203.7	2771	351.1	223.6	157.8	155.6	531.1	301.2	139.0	134.2

Table 3: MCL run times using real data set from Costanzo et al. serially in Julia (on one thread and multithreaded) and in parallel using distributed memory (P) and shared memory (SP) on 8, 16, 32, and 40 cores.

	Julia (1 thrd)	Julia (multi)	P8	P16	P32	P40	SP8	SP16	SP32	SP40
Time (s)	5453	485.3	739.2	427.8	310.4	301.5	724.68	393.1	231.8	199.7

Further results from MCL on the Costanzo et al. dataset.
Average cluster/family size: 6.45
Clusters with greater than 5 members: 229
Singlets: 253
Total # of clusters: 714

Table 4: Smith Watermann run time results implemented serially in Python and Julia and in parallel on 16 (P16) and 32 (P32) cores.

Size (nt)	Python	Julia	SP16	SP32
10	0.0003	0.00012	8.597	7.535
100	0.0238	0.0712	24.783	44.283
500	0.5273	0.3177	345.3	598.5

Table 5: Smith Watermann parallel grid implementation run times for serial Julia and in parallel on 2, 4, 5, 10, 20, and 40 cores.

Size (nt)	Julia	P2	P4	P5	P10	P20	P40
40	0.067	3.34	3.16	5.15	6.93	13.48	30.27
200	0.10	1.88	3.18	3.76	6.95	13.34	30.36
1000	1.18	2.04	3.20	3.83	7.04	13.38	29.34
3000	9.56	3.08	4.19	4.75	7.89	14.00	30.49
5000	26.03	5.36	6.78	6.97	9.60	15.57	31.72
7500	64.80	11.15	10.28	13.00	14.82	19.50	33.80

10000	102.36	22.19	23.90	22.13	27.32	26.26	38.78
15000	234.59	51.72	55.23	55.26	64.54	62.86	57.77