Parallel Implementation of Gotoh Algorithm

Priyanka Ravi[1], Sudharshan Arutselvan[2]

[1] Donald Bren School of Information and Computer Sciences, [2] Department of Electrical Engineering and Computer science

ABSTRACT

Needleman-Wunsch is a well-known global sequence alignment algorithm used for finding similarity between biological sequences and is known to be high intensive task. Using ideal gap penalty for this increases the computation more. The Gotoh algorithm which is an improvement of NW uses affine gap penalty to reduce the computation. Though it has less computation than NW algorithm, it uses dynamic programming model to calculate the similarity scores. Hence we present a model to parallelize the scoring of the similarity and reduce the time consumed. Since, the biological sequences can be very long, it needs to be searched quickly, hence parallelization is implemented in this project. The project implements wavefront method of iterating through the values in the matrices and parallelize the dynamic programming model. The implementation is done is both OpenMP and CUDA and the experimental results of the speedup compared to the serial implementation have been shown.

1. INTRODUCTION

Today, one of the most powerful methods for inferring the biological function of a gene (or the protein that it encodes) is by sequence similarity, using protein and DNA sequence databases. With the development of rapid methods for sequence comparison, discoveries based solely on sequence homology have become routine. Many algorithms were designed to find the optimal alignment between sequences.

The Needleman-Wunsch [6] and Smith-Waterman [7] algorithm for determining the similarity between two sequences provides optimal solutions for global alignment and local alignments respectively. Though they are optimal, they are computationally expensive. Other methods like FASTA and BLAST that are based on heuristics are much faster but do not produce optimal results. An improvement over the Needleman-Wunsch algorithm is provided by the Gotoh algorithm, which produces a more optimal alignment by using the affine gap penalty instead of the ideal gap penalty. This method reduces the number of computations needed by the Needleman-Wunsch algorithm. Though, the algorithm reduces the computational complexity, the number of computations it has to perform for large sequences is still large. Hence, to reap more performance using this algorithm, one can parallelize the same and reduce the time taken by the computations.

[2] discusses a parallel strategy based on divergence, z-align, to locally align sequences using an affine gap function. One parallelization strategy that can be extended and used to address our problem is the wavefront access pattern. If this is used, initially only one computation can occur. Then, the next anti-diagonal or two computations can occur and so on until all the computations are done. This results in an uneven number of computations in each iteration. [4] talks about issues in a fine-grained multithreaded programming model - such as unbalanced work amongst processors. So they partition the similarity matrix into blocks. Then, blocks along the diagonals are implemented in order. This reduces the communication to computation ratio in a distributed system.

The rest of the paper is arranged as follows. Section 2 describes the concepts for better understanding of this paper. Sections 3 and 4 discuss the methodologies followed and proposed. Section 5 shows the results. Section 6 and 7 provide the conclusion and future scope of the project.

2. BACKGROUND

In this section, we introduce the Needleman Wunsch algorithm and the Smith-Waterman algorithm that sets the base for the Gotoh algorithm.

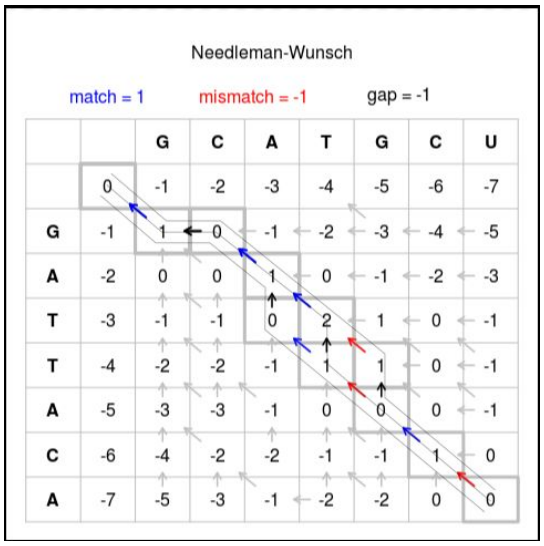
2.1 NEEDLEMAN- WUNSCH ALGORITHM

The Needleman-Wunsch algorithm was one of the first applications of dynamic programming to compare two sequences. This algorithm has two phases, calculating the distance matrices and obtain optimal alignment. In the first phase, it receives two input sequences, S0 and S1 with sizes m and n, respectively. A scoring matrix, H of size m+1 by n+1 is constructed so that, H_{ij} provides similarity score between the sub-sequences of the two strings that end in positions i and j respectively (S0[1...i] & S1[1...j]). Any partial sub-path that tends at a point along the true optimal path must itself be the optimal path leading up to that point. Therefore the optimal path can be determined by incremental extension of the optimal subpaths.

In a simple scoring system, each match, mismatch and gap insertions is given a score. Each score is calculated based on the top, top left and the left cells of the particular block. When a score is calculated from the top, or from the left this represents a gap in our alignment.. When we calculate scores from the diagonal this represents the alignment of the two letters the resulting cell matches to. Every cell takes the maximum of these three blocks. The score in the last cell (bottom right) represents the alignment score for the best alignment. In the second phase, the optimal alignment is obtained by backtracking from the last element in the matrix to the first cell of the matrix. In each step, we go to the highest neighbor.

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

Figure 1: Initial score matrix in Needleman-Wunsch Algorithm



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

Figure 2: Optimal path tracing.

2.2 SMITH-WATERMAN ALGORITHM

The Smith-Waterman Algorithm is similar to the Needleman-Wunsch Algorithm except that, this algorithm finds the optimal similarity for local alignment of the two given sequences. Instead of looking at the entire sequence, the Smith–Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. Score each element from left to right, top to bottom in the matrix, considering the outcomes of substitutions (diagonal scores) or adding gaps (horizontal and vertical scores). If none of the scores are positive, this element gets a 0. Otherwise the highest score is used and the source of that score is recorded. Hence we get many partial similarity between the two sequences.

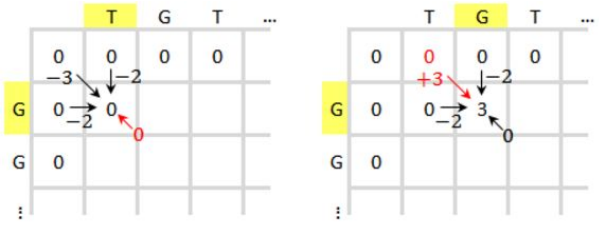


Figure 3: An example of scoring in Smith-Waterman Algorithm.

The traceback will start a large value and go until 0 is reached. This particular part of both sequences will have the similarity score. This can be done for

many other parts to determine different local alignments.

2.3 AFFINE GAP PENALTY

The Needleman-Wunsch algorithm assumes a linear gap penalty and hence as the gap increases, the penalty increases too. However, in biology, it is observed that a larger gap is more likely than a smaller one. Hence, penalizing each gap in the same way is not fair. We should score two small gaps to be worse than one large one Hence, the algorithm uses a ideal gap penalty which reduces the penalty as the gap increases. This is known as affine gap penalty.

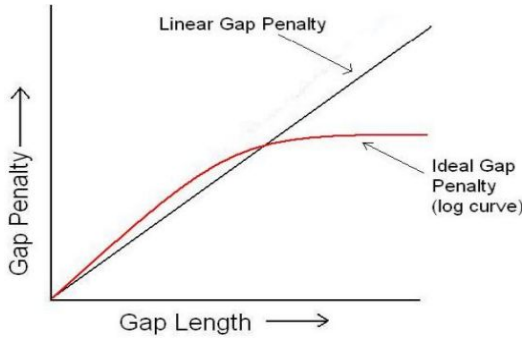


Figure 4: Ideal and linear gap penalty

The affine gap costs are defined by a linear function, g(l)=gap_start+(l-1)*gap_extend. Here, gap_start is the cost of launching a new gap, whereas gap_extend is the cost of extending an existing gap by one place. l is the length of the gap sequence. Using this algorithm better mimics the ideal gap penalty.

2.4 GOTOH ALGORITHM

The Gotoh algorithm uses the affine gap penalty described in the previous section, while trying to optimize the Needleman-Wunsch Algorithm. Apart from the score matrix, this algorithm uses two other matrices, which are used to store the gap penalties in sequence 1 and sequence 2. These three matrices can be described as:

A (i, j) - the optimal similarity score of the optimal alignment of the prefix u [1..i] of u and the prefix v [1..j] of v under affine gap costs.
B (i, j) - the optimal similarity score of the optimal alignment of the prefix u [1..i] of u and the prefix v [1..j] of v which ends with a gap in u.
C (i, j) - the optimal similarity score of the optimal alignment of the prefix u [1..i] of u and the prefix v [1..j] of v which ends with a gap in v.

Each cell score is calculated based on the top, left and top-left cells, similar to Needleman-Wunsch algorithm, using a recursion that is given in the following figure.

recursion:

$$A(i,j) = \max \begin{Bmatrix} A(i-1,j-1) + w(u[i-1], v[j-1]) \\ B(i-1,j-1) + w(u[i-1], v[j-1]) \\ C(i-1,j-1) + w(u[i-1], v[j-1] \end{Bmatrix}, 1 \le i \le m,\ 1 \le j \le n$$

$$B(i,j) = \max \begin{Bmatrix} A(i-1,j) + \text{gap\_start} \\ B(i-1,j) + \text{gap\_extend} \\ C(i-1,j) + \text{gap\_start} \end{Bmatrix}, 1 \le i \le m,\ 1 \le j \le n$$

$$C(i,j) = \max \begin{Bmatrix} A(i,j-1) + \text{gap\_start} \\ B(i,j-1) + \text{gap\_start} \\ C(i,j-1) + \text{gap\_extend} \end{Bmatrix}, 1 \le i \le m,\ 1 \le j \le n$$

Figure 5: Representation of scoring in each matrix

2.5 EFFICIENCY OF GOTOH ALGORITHM

NW algorithm implemented with Affine Gap penalty would involve a computational complexity in the order, $O(n^2m+nm^2)$ as evaluating the gap penalty needs a loop through all the previous nucleotides to find the one that gives the maximum score. Gotoh algorithm reduces this complexity to O(nm) by maintaining the two additional gap penalty matrices. However, this raises the space complexity by 3 times. This is negligible as of the problem increases.

3. SERIAL IMPLEMENTATION AND OBSERVATION

The serial implementation of the Gotoh Algorithm we used can be found in [8]. This was run on Class64-AMD machine of the HPC cluster. By measuring the time taken for initialization, optimal scoring and back tracing using checkpoints, we observed that the time taken by the optimal scoring is around 65% of the total time. Hence, parallelizing this part of the program could lead to potential speedup and this was the intuition on which we based our proposed parallel implementation.

4. PARALLEL IMPLEMENTATION

In this section, we propose an idea to parallelize the calculation of the similarity matrix. As we saw in the serial implementation, each step involves calculation of a single cell as the best possible value

using the values of three other cells on the left, top and top-left. For correctness, this implies that these three values are calculated before the value of the current cell is being evaluated. The serial implementation iterates through each element in each row. This cannot be directly parallelized as there are dependencies between the iterations. Iterating along the columns will not be possible due to similar dependency reasons.

One observation we made while computing the scores is that the score computation for elements along the same anti-diagonal can be done independently. However, each anti-diagonal should be evaluated in order. If this order in computation is followed, computations can be performed in parallel without losing the correctness in computation and results.

Hence, instead of iterating along the rows or columns, the proposed approach suggests an iteration in which the outer loop goes over the anti-diagonals and the inner loop iterates through all the elements in each anti-diagonal. This method is called wavefront method. **The number of elements in each anti-diagonal keeps increasing to the minimum of the lengths of the two sequences and reduces after iterations of maximum of lengths of the two sequences**.

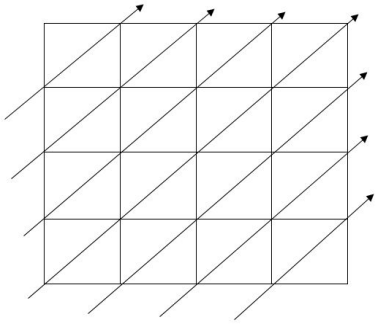


Figure 6: Wavefront method depiction

This pattern in the wavefront method can be exploited for parallelization. For every outer loop all the elements in the inner loop can be parallelized.

This paper proposes implementation of this type of parallelization in both OpenMP and CUDA and compare the results of speedup of both the implementations.

4.1 OpenMP Implementation

To implement the proposed method in OpenMP, the computations in each anti-diagonal are scheduled to be calculated in parallel using a loop with omp parallel for construct. The outer loop cannot be parallelized as the order of computation must still be maintained - diagonal wise.

4.2 CUDA Implementation

Cells along an anti-diagonal can be computed in parallel. So each cell can be thought of as the equivalent to a block in the CUDA language. Now, each cell requires the computation of three values - A[i,j], B[i,j], C[i,j]. So, three threads can be active in each block to perform this computation. However, this computation is super inefficient and degrades performance as most of the GPU is inactive if this is the case. We are not reaping maximum parallelism by using the GPU.

So, a blocked approach is described and implemented. In this approach, a block size - height and width is used; blocks are then defined along each diagonal. The computations of the cells in each of blocks is then carried out diagonal by diagonal. Now, more number of threads are active per block. However, the maximum number of threads that is active per block is always restricted by the order imposed in the diagonal operations. The number of blocks vary by the iteration and hence the deployment of the required number of blocks is handled by the CPU. Only the computations take place in the GPU. Finally, additional computation is performed for the last few cells of the matrix that are not covered in the blocked approach.

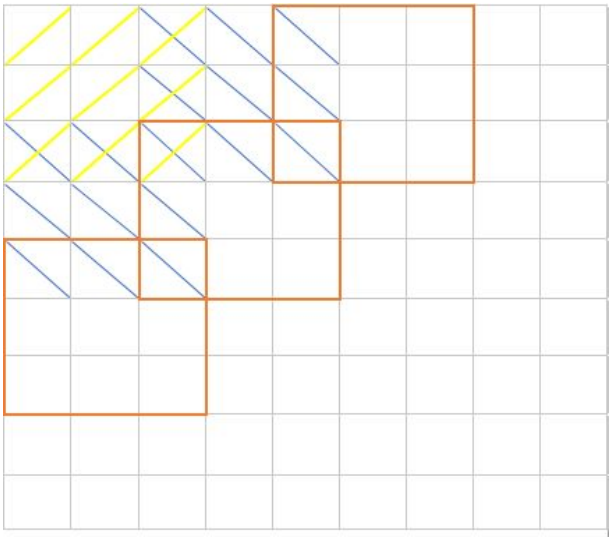


Figure 7: Depiction of execution in CUDA

For each block,

- Copy values from the global memory to the shared memory (in size of blocks).

- Perform computation of the cells in the block (anti-diagonal wise).
- Shared memory helps reduce overhead of communication.
- Transfer values back to the global memory.

5. EXPERIMENTAL RESULTS

Serial implementation [8] results for running 5 trials with input sequences of size 5520 provides and averages 2.25 seconds for calculating the optimal similarity score and 1.35 seconds for the remaining program. These were results obtained from execution on class64-amd machine in HPC cluster.

5.1 OpenMP Results

The same input sizes for the sequences were used in the execution for the OpenMP implementation and run on the same class64-amd machine for proper comparison in compiler C99 and flags O3, and fopenmp. The results obtained for the OpenMP implementation parallelizing the dynamic programming part provide the following results. We are using a BLOSUM62 for the match and mismatch scores between the RNA elements.

Number of Cores	Time Taken	Speed Up
4	0.64	3.52
8	0.55	4.09
16	0.34	6.62
32	1.67	1.67

Table 1: Results for OpenMP parallel implementation with static scheduling.

Figure 8: Speedup of OpenMP implementation compared to different cores.

These results show that the speedup reduces for cores above 16. This is due to the effect of improper scheduling. Due to this improper balance, many threads may need to wait for other threads to complete the job assigned. This is hugely affected in 32 core system is because, when there are 32 threads and just 40 elements to execute, the schedule will assign the first few threads more than other and hence, they have to wait longer.

5.2 CUDA Results

Following the CUDA implementation described in the previous section, the blocked approach used blocks of size 3*3 - calculating 2*2 new values per block. Another assumption is made that the two input sequences are equal in length. The GPU used here, is NVIDIA Tesla K80 in the HPC cluster. For the same input sequences of size 5520, the average time taken for the calculation of the optimal similarity score using a GPU is 0.057 seconds. This provides a speedup of 39.47x ~ 40x.

5.3 Serial vs OpenMP vs CUDA Results

As expected, both the parallel implementations of the Gotoh algorithm are significantly better than the serial version. While OpenMP provided a speed up of 7x, the CUDA counterpart raised this number close to 40x. This demonstrates the power of a GPU over a CPU.

Intuitively, one would think that this problem cannot be adapted to fit the architecture of a GPU due to the nature of the dependencies. However, tweaking the algorithm to expose parallelism has given us enough proof in the form of the results obtained, that the speed of the GPU can still allow it to improve performance of such an algorithm.

6. CONCLUSION

In this report, we have discussed the wavefront approach in parallelizing the Gotoh-Algorithm using OpenMP and CUDA. We can see from the results that OpenMP implementation provided good results in terms of speedup which came upto 3-6 times more than the serial implementation.

Though this method is faster in OpenMP, it has high amount of cache misses and data fetch operations as we try to access elements in different rows. This will be a problem as the sequence size keeps increasing. If the total size is less than the cache size then the cache

misses will be low as in the results we had zero page faults. The data fetches will not be reduced though here.

CUDA provides a significant speed up of approximately 40x which demonstrates that the GPU can can suit such a problem due to its computational sources and speed despite the dependencies posed in the problem. However, the GPU is still not efficiently used due to the limited resources spent in solving such problems, that is, the number of threads active at different times varies and is not always fully utilized.

7. FUTURE SCOPE

In this paper, we have seen how OpenMP and CUDA improve the speed of the Gotoh-algorithm by parallelizing the calculation of the similarity score matrix.

The parallelization mentioned in this project involves static scheduling due to which load balancing is not the proper. Few threads finishes earlier and waits for the other threads to complete. As shown in the results, as the number of cores increases, the scheduling becomes worse. Hence, to avoid this we can experiment changing the chunk sizes for the static schedule, use dynamic and guided schedules to gain higher speedups. Also, experiment can be done to choose best scheduling for different number of cores.

CUDA implementation can be modified to perform the algorithm on unequal length sequences. Varying the block sizes and visualizing its effect on performance would be an interesting task. For large sequences, this performance would scale till the maximum number of computations per block is equal to the warp size itself.

One could also think of how the algorithm would scale in online processing of the inputs.

8. ACKNOWLEDGEMENT

The authors would like to thank Prof. Aparna Chandramowlishwaran for giving the opportunity to explore an interesting domain and document the results.

9. REFERENCES

[1] Ananth Prabhu G, Dr. Ganesh Aithal. Parallelization of Biological Gene Sequencing Technique with Optimized Smith Waterman Algorithm. International Journal of Innovative Research in Computer and Communication Engineering, Vol. 2, Issue 6, June 2014.

[2] Batistab, Boukerchea, Magalhaes & Melob. A parallel strategy for biological sequence alignment in restricted memory space. Journal of Parallel and Distributed Computing, Volume 68, Issue 4, April 2008, Pages 548-561 http://www.sciencedirect.com/science/article/pii/S0743731507001530

[3] Harsh Shukla, Monika Shah. Optimizing Parallel Scan Smith Waterman Algorithm on GPU. Proceedings of 6th SARC-IRF International Conference, 06th July-2014.

[4] Martins, Del Cuvillo, Useche, Theobald & Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. Pacific Symposium on Biocomputing(2001). https://psb.stanford.edu/psb-online/proceedings/psb01/martins.pdf

[5] Rose, J and Eisenmenger, F. J. A Fast Unbiased Comparison of Protein Structures by means of Needleman-Wunsch Algorithm. Journal of Molecular Evolution, April 1991, Volume 32, Issue 4.

[6] Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology Volume 48, Issue 3, 28 March 1970, Pages 443-453

[7] T. F. Smith and M. S. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology, vol. 147, no. 1, pp.195-197, 1981.

[8] https://github.com/oboes/gotoh