Nofar Alfasi

Parallel implementation of Sequence Alignment

MPI+OpenMP+CUDA Integration:

At first we create two processes. The input file initially is known for one machine only, this machine is related to the first process (with rank 0). The same machine at the end of the run writes the final results to the output file.

The first process reads the data from the input file and then sends the weights, the first sequence (Seq1), and half of the query sequences (Seq2) to the second process (with rank 1). The communication between the two processes is done using MPI.

Both processes perform the exact same computations on their half of the sequences - partially with OpenMP, partially with CUDA.

The results are sent from the second process to the first process, which after writes the final results to the output file.

Sequence Alignment Algorithm Implementation:

My implementation for finding the offset and the hyphen location with the best alignment score for a given sequence is composed of the following steps:

1. Calculating the scoring matrix based on the given weights and score function. For this step I have used OpenMP and CUDA (separately), both performing the same calculations.
2. Computing the sum matrix based on the results of the scoring matrix.
3. Finding the best offsets based on the highest value of the sum matrix.
4. Finding the offset and the hyphen location of the Mutant Sequence MS(k) with the best alignment score. This step is done in parallel by using OpenMP.

For a given Seq1 and Seq2 we create two matrices. Both matrices have the same dimensions of NxM, where N is the number of rows and it is equal to the length of Seq2+1, and M is the number of columns and it is equal to the length of Seq1+1.

For example, if our sequences Seq1, Seq2 are PSHLQY, SHQ respectively, our matrices will have 4 rows and 7 columns.

In each matrix all the elements of the first row and the first column are set to 0.

The Scoring Matrix

The first matrix is the scoringMatrix, and it contains the scores of the sequences in each cell of two chars from Seq1 and Seq2.

For example, for the sequences PSHLQY, SHQ we get the next scoringMatrix:

		P	S	H	L	Q	Y
	0	0	0	0	0	0	0
S	0	-1.1	2	-1.3	-4	0	0
H	0	-4	-1.3	2	-1.3	-1.5	0
Q	0	-4	-4	-1.5	-1.3	2	-1.3

The blue numbers are the calculated scores for a pair of chars, and the red numbers are the penalty value for chars outside of bounds.

The computation of the scoringMatrix is done in parallel since the computation of the value of each cell is not dependent on other cells of the matrix.

In my project for the MPI+OpenMP+CUDA integration, the calculation of the scoringMatrix is done partially with OpenMP and partially with CUDA (half of the sequences are computed by OpenMP, and the half by CUDA).

For OpenMP we use number of threads as the number of the sequences we have, so that all of our sequences are being computed concurrently. For the part of the scoringMatrix computation we use nested parallelism to achieve even better results, since we have no dependencies this matrix calculations.

OpenMP Runtime Complexity: O((N * max{N, M-N})/OMP)

For CUDA we create a kernel for each sequence. In each block we have 32x32 threads (total of 1024 per block), and for each grid of a kernel we have (M/32+1)x(N/32+1) blocks. Each thread is responsible for one cell of the scoringMatrix. That way by using a GPU we can run all blocks concurrently, executing the threads in each block simultaneously.

CUDA Runtime Complexity: O(N)

The Sum Matrix

The second matrix is the sumMatrix, and it contains in each cell a special calculation value (explained next) of the total sum of the scores from the beginning of the sequence (Seq2).

The idea of this special calculation for the total sum is taken from the Smith–Waterman [1] algorithm. The Smith–Waterman algorithm is a general sequence alignment method based on dynamic programming. Smith algorithm does not take in consideration different weights values (of the Conservative and Semi-Conservative groups). This complicates our problem since we have to take in consideration the contribution of the corresponding weight to the total score.

To solve this problem, I have created a different equation for the computation of the sumMatrix:

$$\mathrm{sumMatrix[i][j]} = \max \begin{cases} \mathrm{sumMatrix[i-1][j-1] + scoringMatrix[i][j]} \\ \mathrm{sumMatrix[i-1][j] + scoringMatrix[i][j] + minimumWeight} \\ \mathrm{sumMatrix[i][j-1] + scoringMatrix[i][j] + minimumWeight} \\ \mathrm{penalty} \end{cases}$$

For example, for PSHLQY, SHQ we get the next sumMatrix:

		P	S	H	L	Q	Y
	0	0	0	0	0	0	0
S	0	-1.1	2	-0.8	-4	0	0
H	0	-4	-0.8	4	1.2	-1.8	0
Q	0	-4	-4	1	2.7	3.2	0.4

where the $minimumWeight$ is the value of the minimum weight contribution (-1.5).

The computation of the sumMatrix can not be done in parallel, since the computation done in each loop is dependent on the previous iterations.

Runtime Complexity: O(N * (M - N))

Finding Best Score

For finding the values of n and k with the highest score we use the traceback process (explained next). First have to find the element on the last row of the sumMatrix with has the highest value.

After finding the index with the highest score, we also want to check the cells next to it. We start on the last row of the sumMatrix on the index we found and move up the sumMatrix to find the best results.

Starting at the element with the highest score, traceback based on the source of each score recursively, until the first row or first column is encountered. The location of the hyphen that have the highest similarity score based on the given scoring system is generated in this process.

On each iteration we check if the value of the left cell is higher then the top left cell. If so we compare the current highest score and the score for this location of hyphen, and when we find a better score we update the current highest score and the hyphen sign location.

Estimated Runtime Complexity: O(N)

Future Improvements

1. For large amount of sequences consider creating a kernel for more than one sequence.
2. Managing Mutant Sequences with more than one hyphen sign using the sumMatrix.
3. If possible, consider using only one matrix instead of both scoringMatrix and sumMatrix.

Conclusion

- By using MPI we manage to reduce time complexity by nearly half of the original time.
- CUDA execution time is faster than the OpenMP execution time, especially for large amount of data. However, CUDA approach to this problem was problematic, because CUDA cannot handle these kinds of data dependencies batter than other parallel libraries.
- There are many common problems affecting the final speedup in parallel computing using OpenMP, like data dependencies between threads and lack of memory space.
- By implementing the idea of using the sumMatrix, taken from Smith algorithm, we are saving a lot of time by summing the scores for each offset one time, which allows us to find the offset and hyphen location with the best score easily.
  The cost of this implementation is that it is using more space, having to keep two matrices for each sequence, and we still might need to calculate the total score for a given offset and hyphen location a few times, depending on the sequences.

References

1. Wikipedia contributors. (2020, August 29). Smith–Waterman algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved 00:17, September 1, 2020, from https://en.wikipedia.org/w/index.php?title=Smith%E2%80%93Waterman_algorithm&oldid=975605920