CS466 Project Final Report
Implementation for Parallel Hirschberg Algorithm

Wenda Qiu

December 2020

1 Introduction

Sequence alignment is an important and fundamental problem in bio-information research. By aligning sequences, people can find identical parts in the DNA or RNA, which will most likely to indicate a similar function or structure. In the field, edit distance is used as the criterion for aligning, with some pre-defined weighting for different types of editing (insert, delete, substitution).

In the class, Hirschberg Algorithm [1] is introduced for linear space global alignment. In this course project, I implement this algorithm and try to further accelerate it with multithreading. Thanks to the hardware development, modern computers often has multiple cores so it will be a huge benefit if we can do it in parallel.

The codes can be found on Github `github.com/Akigeor/CS466-Project---Parallel-Hirschberg` .

2 Implementation Details

2.1 Hirschberg Algorithm

Hirschberg Algorithm use a divide and conquer strategy to smartly calculate the backtracing path, instead of directly storing them in the memory, and can reduce the memory usage from quadratic to linear. Specifically, it will find the point at the middle column in the DP backtracing table, that the optimal path will pass through. This step is done using prefix and suffix global alignment without backtracing. The optimal point will be reported and divide the optimal path into two parts. The algorithm will recursively calculate the left path and right path. I put an illustration in the figure below.

For the implementation, I make the two original strings global variables, and the recursive Hirschberg function takes two positions in the DP array: the upper left corner $a = (x1, y1)$ and the lower right corner $b = (x2, y2)$. If the two corners share the same row or the same column, the case become trivial: the tracing path is simply the path from a to b. Other wise, I find the middle column

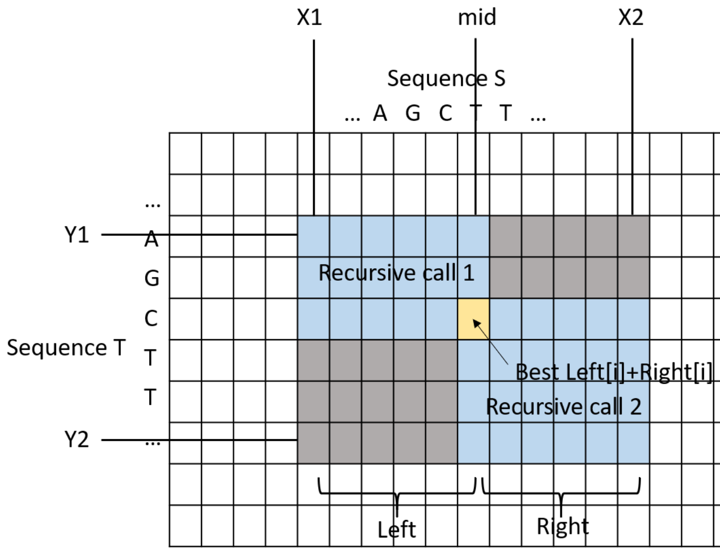

Figure 1: Hirschberg Algorithm

mid and compute the left part and right part, using Needleman–Wunsch algorithm without the back tracing by only memorizing the last column (assuming a column by column DP) in the DP array to achieve linear space.

One finding in my implementation is that the two parts for recursion must **not** share the same endings, otherwise it will result in an endless loop. For example, if the state $(1, 1, 2, 2)$ finds optimal passing at $(1, 1)$, the two recursive call can not be $(1, 1, 1, 1)$ and $(1, 1, 2, 2)$ (a replication of the original state). The solution is to make another step (i.e. going right, going down or going down-right) after the optimal passing point as the upper left corner for the second recursive call. In the example above, it will make the second call into $(1, 2, 2, 2)$ or $(2, 1, 2, 2)$ or $(2, 2, 2, 2)$, depending on the direction of optimal step.

2.2 Parallel Approach 1: DP Acceleration

Since the row-by-row dynamic programming introduced in class is not able to be computed in parallel ways, since the cell currently being computed requires the cell value we just completed. I implement a diagonal-by-diagonal DP to make the parallel acceleration possible. I put an illustration in Figure 2 below. diagonals and cells in a single diagonal does not depends on each other. Thus, we can partition the diagonal into several parts and use multiple threads to complete them separately. Although theoretically this can make the time usage smaller but in practice, the overhead of multi-threading, for example, copy time for data sharing between threads, can be a huge drawback on acceleration. I demonstrate this in the experiment section.

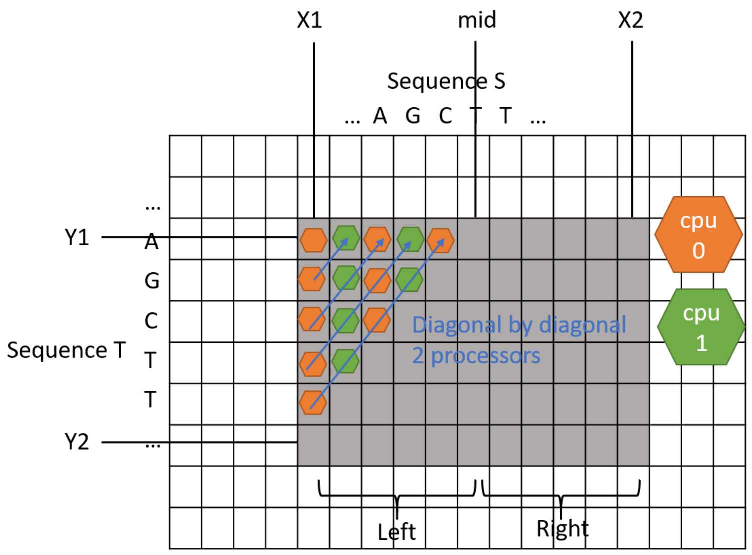

Figure 2: Parallel Approach 1: DP Acceleration with 2 processors

2.3 Parallel Approach 2: Recursion Acceleration

The divide and conquer procedure can be accelerated by multiprocessing, since the states will form a tree and states in the same tree level can be done in parallel. I implement the Hirschberg Algorithm in a BFS manner in stead of the original DFS design. This is because we need to compute the states level by level. In practice, I use the `imap` in `mutiprocessing.Pool` in Python stdlib to compute the states in parallel. I show the illustration in Figure 3 below.

3 Experiment

3.1 Provement of Correctness

In the experiment part, I first prove the correctness of the implementation by checking the generated path in the DP array against the results using a Needleman–Wunsch implementation (I reuse part of the code in homework). I tested the implementations on randomly generated AGCT gene sequences with random lengths under 200. The matching score is the same as the one used in the homework: 1 for a match, -1 for a dismatch and gap. I run this testing with enough iterations ($10,000$ random sequence pairs).

3.2 Experiment in Time & Space Usage

I tested different methods with 10 random sequences with length $5,000$ and fixed random seed. I plot Figure 4 using the time usage as x-axis and memory

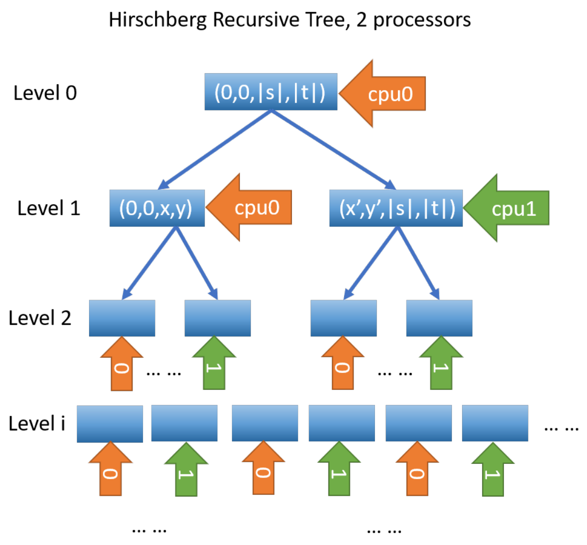

Figure 3: Parallel Approach 2: Recursion Acceleration with 2 processors

usage as y-axis, both in log scale. I put the raw values of the figure in appendix. From the figure, we can see that Needleman–Wunsch consumes huge amount of memory comparing to Hirschberg Algorithm. And the first approach of multi-threading causes much overhead, looking at the significantly worse time usage. The second approach on recursion acceleration can effectively leverage multiple processes and even surpasses Needleman–Wunsch in terms of time uasge where Needleman–Wunsch is supposed to be faster. One hypothesis is that the memory addressing in large list of list dp array cost more time. The 32 workers version is the fastest among them, which is quite intuitive. However the speed up seems to be negligible for methods more than 4 processors. I guess the reason is still the overhead caused by multiprocessing (e.g. initializing and data sharing).

4 Github Deployment

I commit the codes of this project to Github `https://github.com/Akigeor/CS466-Project---Parallel-Hirschberg` by requirement. Here is a brief description of all the files in the repo:

- `Hirschberg.py` The implementation for plain Hirschberg.
- `main.py` The code for experiments. Also served as an example usage.
- `NeedlemanWunsch.py` The implementation for Needleman Wunsch algorithm, which is used as ground truth.

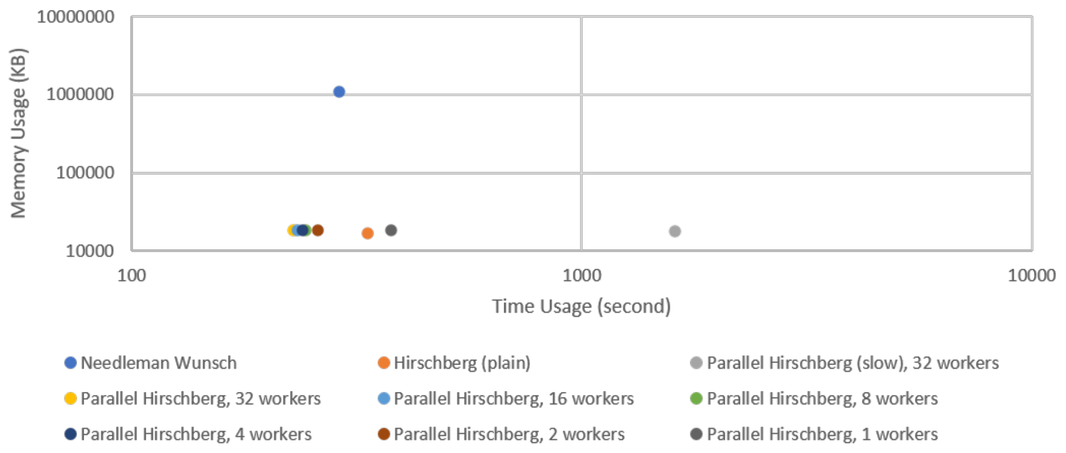

Figure 4: Time - Memory plot for compared methods

- `ParallelHirschberg.py` The implementation for parallel Hirschberg using approach 2.
- `ParallelHirschberg_failed.py` The implementation for parallel Hirschberg using approach 1, which turns out to be not effective.
- `input.txt` The input file for time benchmarks of 10 sequence pairs of length $5,000$.
- `logs.txt` The logs for experiment commands.

5 Conclusion

In this project, I implemented a parallel Hirschberg algorithm, that can find the global sequence alignment in quadratic time and linear memory, without much loss of time efficiency. I make 2 approaches to leverage multiple processors, the DP acceleration failed due to too much overhead on data sharing and another approach on recursion acceleration seems to be effective. I committed my codes to Github `https://github.com/Akigeor/CS466-Project---Parallel-Hirschberg` .

6 Future Work

In this project one approach to accelerate DP failed. Currently I'm using Python for the implementation. In the future work I may use more efficient programming languages such as C++, to see if this approach can work.

A Raw Values in Figure 4

Parallel Hirschberg (slow) in the table refers to the failed approach 1.

Method	Time (s)	Memory (KB)
Needleman Wunsch	288.82	1082548
Hirschberg (plain)	334.2	16984
Parallel Hirschberg (slow), 32 workers	1,613.66	17932
Parallel Hirschberg, 32 workers	229.41	18484
Parallel Hirschberg, 16 workers	234.32	18240
Parallel Hirschberg, 8 workers	244.03	18356
Parallel Hirschberg, 4 workers	239.93	18252
Parallel Hirschberg, 2 workers	259.01	18496
Parallel Hirschberg, 1 workers	376.63	18460

Table 1: Raw Values in Figure 4

References

[1] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.