Parallel K-Sequence Alignment Algorithm

Lai Wei Hong, 1226091

***Abstract*—The Needleman-Wunsch algorithm was introduced by Saul B.Needleman and Christian D.Wunsch to compare the similarities found in amino acid sequence of two proteins. This algorithm makes use of dynamic programming to compute the optimal pairwise alignment for any two sequences. However, the inherently sequential nature of the algorithm has led to long run times when comparing a large number of sequences. In this project, we have implemented various changes to the original algorithm to introduce parallelism and reduce the computational time for the algorithm. This was achieved by splitting the state space and processing them diagonally. We have also introduced dynamic thread allocation to reduce communication overhead between threads and dynamic task allocation to reduce the idle time for CPU cores. In the end, we compare and contrasted our approach with the original algorithm to highlight the speedup achieved.**

I. Introduction

SEQUENCE alignment is a problem in bioinformatics where protein sequences are matched in a bid to identify regions of similarity. Doing so allows researchers to uncover relationships between two sequences. Over the years, many techniques have been introduced to tackle this problem. Some examples of such techniques are maximal unique matching, dot-matrix methods and dynamic programming. The Needleman-Wunsch algorithm is an example of a pairwise sequence alignment method that utilises dynamic programming when computing the best alignment for two sequences[1].

Traditionally, dynamic programming methods are difficult to parallelise due to the reliance of its state space from a prior time step to perform computations in the current time step. This relationship inherently limits dynamic programming algorithms to be processed sequentially. However, several papers have been published that introduces ideas to parallelise these algorithms. In 1995, D.Tang and G.Gupta proposed a parallel dynamic algorithm that reduces the computational complexity to O(n), while employing O(n^2) number of processors[2]. The were able to achieve this by diagonalising the 2-dimensional state space and parallelise the computations in the outer most loop at every iteration. On the other hand, Bateni et al. proposed an algorithm that solves dynamic programming problems in trees in 2018[3].

II. Skeleton Code Overview

As part of the project, we were given skeleton code to act as a basis for further development. The underlying algorithm used to find the most similar sequence for k number of different sequences is based on the Needleman-Wunsch algorithm that we talked about in the Introduction section. Due to word limits, we will only be providing a high level description in the pseudocode section below.

Algorithm 1 Skeleton Code

```
1: $alignmentHash \leftarrow$ ""
2: for All possible sequence combination do
3:     Generate a table for storing optimal substructure answers
4:     Reconstruct the solution based on the state of the table
5:     Obtained the aligned sequences for the two sequence
6:     $alignmentHash += hash(alignedSequence1 + alignedSequence2)$
7: end for
       return $alignmentHash$
```

A. Problems encountered when experimenting with skeleton code

One of the key shortcomings of the skeleton code is the sequential nature of the dynamic programming approach.

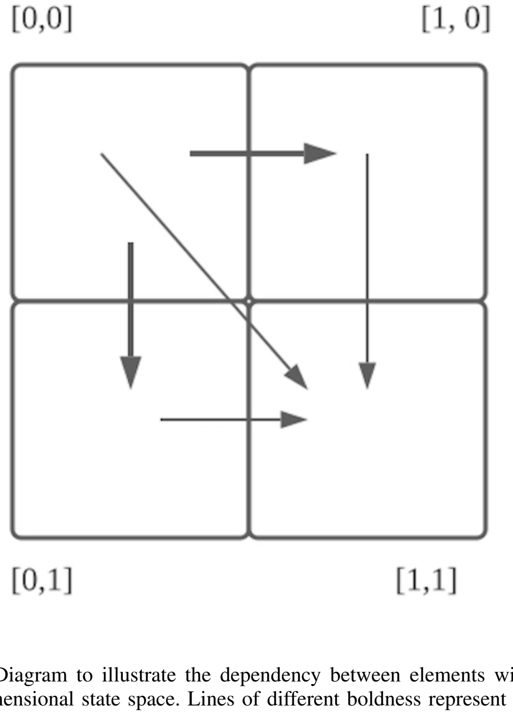


Fig. 1. Diagram to illustrate the dependency between elements within a 2 x 2 2-dimensional state space. Lines of different boldness represent different iterations

Consider the following state space with a size of 2 by 2, we aim to compute the value for element[1][1]. To do so, we are reliant on the values set in elements [0][0], [0][1] and [1][0]. Similarly, elements at [0][1] and [1][0] are dependent on the value at element [0][0]. As a result, a processing sequence of [0][0] $->$ [0][1] $->$ [1][0] $->$ [1][1] emerges. The dependency among elements within a state space, in turn, enforces sequential computations during table generation. This could lead to long run times, particularly for long sequences where sequence.size() $>$ 40000.

Another issue with the skeleton code is the sequential processing of k number of sequences. This issue has contributed to the long running times when computing problems where the value of k is high, i.e. there is a large quantity of sequences to compare. Besides that, as a consequence of its sequential nature, a lot of the CPU cores were idling throughout the execution of the program.

To consider the problem more concretely, we can consider the time complexity of the skeleton code. The complexity of the underlying Needleman-Wunsch algorithm is O(mn)[1]. When we start comparing k number of sequences, the time complexity of the algorithm increases to O((k-1)(k)mn). If we start considering the case where k $->$ n, then the time complexity of the current approach can be said to be $O(mn^3)$. This is not ideal and needs to be addressed.

B. Proposed changes

Under close examination of the skeleton code, we determined that the sequential nature of the code has been hindering its performance. To combat this, we propose three major changes to the skeleton code.

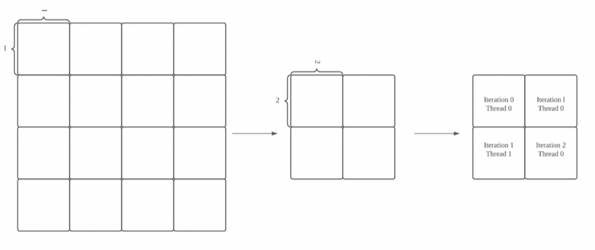


Fig. 2. Diagram to illustrate the partitioning and parallel diagonal processing of the partitioned tiles

1) Grid partitioning and diagonal processing: In this section, we will be considering the operations carried out when comparing two sequences.

Using ideas introduced by D.Tang and G.Gupta[2], we split the state space of the sequence alignment problem into (m + n)diagonals, where m and n are the length of sequences that are compared. Then, we process each diagonal with the maximum number of threads available to the program. Theoretically, this approach could reduce the time complexity of the Needleman-Wunsch algorithm to O(m + n) from O(mn) while employing O(p) processors where p is the MIN(m, n). In practice, such a speedup was not observed. The computational time required to obtain the value of a cell in the state space is much lower when compared to the communication overhead needed during context switching. As a result of this, we observe performance degradation.

Upon identifying the root cause of the issues from the diagonalisation process, we partition the state space into tiles of size n by n. Each tile will be executed sequentially. The diagonalisation process outlined above will be applied onto the tiles that were partitioned. Doing so yields multiple benefits. By partitioning the state space into tiles, we increase the computation time used in each thread to the point where it is magnitudes greater than the communication overhead required. On top of that, we reduce the need of a high number of threads used during parallel computation due to the reduced resolution. This further reduces the penalty caused by thread communication overhead.

2) Dynamic Tile Size Generation: When developing the logic for parallelising the table generation function as outlined in the previous subsection, we ran into the need to constantly adjust tile sizes to obtain optimal performance. A suboptimal tile size has a dramatic effect on the performance of the code as, intuitively, the tile size affects the number of threads assigned to the parallel table generation step. If the tile size is too small, it suffers from the thread communication penalties. In the opposite end, if the tile size is too big, we run into the issue where the parallel code converges to sequential performance.

Algorithm 2 Dynamic Thread Allocation

```
Input: The size of sequences that are being compared, m and n
Output: The optimal number of threads for the problem, p
 1: if $m > 10000$ then
 2:     $r_1 \leftarrow 2$
 3: else
 4:     $r_1 \leftarrow \frac{1}{2}$
 5: end if
 6: if $m > 10000$ then
 7:     $r_2 \leftarrow 1.5$
 8: else
 9:     $r_2 \leftarrow \frac{2}{3}$
10: end if
11: $TileSizeM \leftarrow 2000 * r_2 * \frac{\log(\frac{m}{10000})}{\log r_1}$
12: $threadNumM \leftarrow \frac{m}{TileSizeM}$
13: repeat the steps above by replacing m with n and obtain the value of threadNumN
        return MIN(threadNumM, threadNumN)
```

We designed an algorithm to dynamically determine the optimal tile size given the size of the state space. The design of this algorithm hinges on our observation of the performance of different tile sizes across different sequence lengths, as seen in below. From the diagram, we can conclude that to maintain optimal performance, the tile size needs to grow by 50% whenever the sequence length doubles. This relationship can be generalised as a geometric sequence at Equation 1.

$$A_n = A_1 r^{n-1} \tag{1}$$

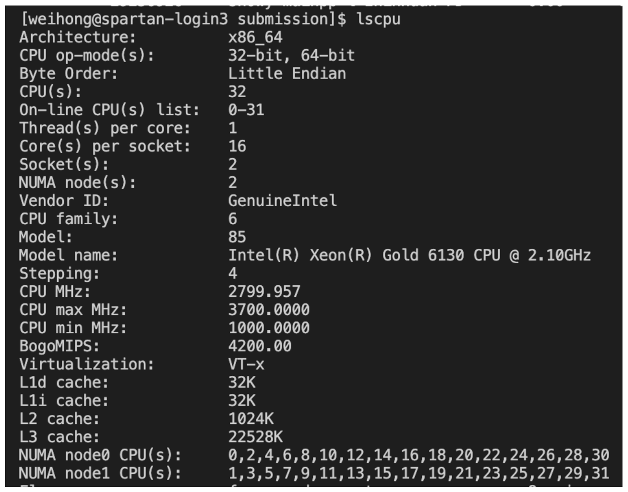
Where A_1 = 10000 and r = 1.5 or $\frac{2}{3}$ depending on the value of A_n

3) Dynamic Task Allocation: This change is introduced to deal with the sequential execution of the skeleton code for different sequences, where alignments are calculated one at a time as outlined in skeleton issue 2. To reiterate the findings from the previous section, all (k-1)k/2 number of sequence combinations are processed sequentially, leading to long run times for the skeleton code in cases where k is high. Initial experimentation revealed that naively applying #pragma omp for on the existing for loop in the skeleton code is unfeasible due to the need to put the state space for all (k-1)k/2 sequences into memory. Therefore, we propose a queue based system where sequence matching tasks are carried out by in order based on the available resources.

Algorithm 3 Dynamic Task Allocation

```
Require: The number of sequences, k , System Memory Capacity, Mem, Maximum number of threads, T
1:  Insert $\frac{(k)(k-1)}{2}$ sequence combinations into a map, M
2:  Compute the memory requirement for all the elements in M
3:  if $k > 5 \mid\mid \sum_{Mem_i \in M} Mem_i.memoryRequirement >$ 0.6 * Mem then
4:      while All elements within the map has not been processed do
5:          $availableMemory \leftarrow Mem$
6:          $availableThread \leftarrow T$
7:          $execKeyQueue \leftarrow \{\}$
8:          for All elements in M do
9:              if $M[i].threadNum + 1 < availableThread$ and $M[i].memoryRequirement < availableMemory$ then
10:                 push M[i].key to execKeyQueue
11:             end if
12:         end for
13:         for every element, e in execKeyQueue do
14:             $problemKey \leftarrow e$
15:             create a new thread to solve sequence alignment problem for sequences at M[$problemKey$]
16:         end for
17:     end while
18: end if
```

The program first creates a map that stores all the sequences that needs to be processed, along with its memory and thread requirement. The task allocation mechanism sits within a main loop that checks if all the sequences have been processed. In every iteration of the loop, the program starts by scheduling tasks based on the memory available to the program. Scheduling priority is given to sequences with high memory requirements. Scheduling is paused when either all the memory or all the threads are allocated. A thread is spawned for each of the scheduled sequences to compute the optimal alignments for the sequences. Once all the threads has finished execution and joins the main thread, a new round of scheduling takes place.



```
[weihong@spartan-login3 submission]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    1
Core(s) per socket:    16
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz
Stepping:              4
CPU MHz:               2799.957
CPU max MHz:           3700.0000
CPU min MHz:           1000.0000
BogoMIPS:              4200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              22528K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
```

Fig. 3. Spartan Node Info

4) Thread affinity: Considering that the program will be running on Spartan where each node has the specifications shown in Figure 3, work has to be done to ensure that threads are distributed to the correct CPU cores. If not handled appropriately, the massive difference in the data transfer speed seen in the interconnection network and core to core communication will introduce severe performance penalty to the code.

We have identified to key regions where thread placement influences the resultant performance, which is the parallel region where task are allocated and when the parallel generation of the substructure table. In this project, we define the OMP _PLACES environment variable to be OMP_PLACES="{0:16:2}, {1:16,2}". This corresponds to each place binding to one of the two NUMA nodes in the CPU.

In the task allocation mechanism, we set proc_bind to be "spread" during program run time. Doing so ensures that the main threads for the underlying alignment computation work is not confined within a single NUMA node. This has helped reduce the communication penalty since child threads do not need to return data back to the parent node via the interconnection network.

As for the parallel table generation region, we need to address to different aspects. First, we need to consider the memory placement where the state space of the problem is stored. We need to ensure that the state space data is initialised and stored in the same thread to reduce the number of cache miss rate. Besides that, contrary to the task allocation mechanism, a lot of communication will occur between the threads during the table generation process, so attention was given to ensure that threads in this parallel region are all placed within the same NUMA node. To achieve these two objectives, we have to use proc_bind = "close". On top of that, we are utilising the First Touch memory policy available in Linux system to ensure that the data pages that store state information is placed close to the corresponding thread.

III. Experiments

In this subsection, we will compare the run times of different configurations of the code in different scenarios. By comparing the run times of the programs would give us some good insight on the performance improvement brought by parallelism. Information on the implementation tested and the test scenarios are listed in Appendix A.

A. Results

Sequence Length	Average runtime across 3 runs (10^{-}1s)		
	Seq	Fixed Tile	Dynamic Task + Dynamic Tile
100	0.003	0.003	0.003
1000	0.1	0.2	0.2
5000	3.2	2.8	3.5
10000	12.6	6.9	7.7
30000	127.9	51.9	60.5
60000	537.2	193.0	219.9
80000	952.3	265.7	366.5

TABLE I
AVERAGE RUN TIME WHEN K IS FIXED AND SEQUENCE LENGTH IS VARIED

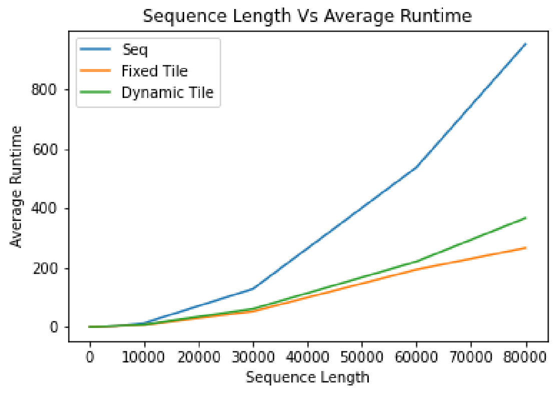


Fig. 4. Graph of Sequence Length against Average Runtime

k	Average runtime across 3 runs (s)			
	Seq	Fixed Tile	Dynamic Tile	Dynamic Task + Dynamic Tile
5	12.6	10.2	10.6	10.3
7	26.5	21.2	22.6	13.1
10	56.8	45.8	47.5	27.1
12	83.3	66.3	69.7	42.8
15	132.6	108.0	110.6	66.4
17	171.3	136.4	143.6	87.7
20	239.2	190.8	199.6	120.1

TABLE II
AVERAGE RUN TIME WHEN SEQUENCE LENGTH IS FIXED AND K IS VARIED

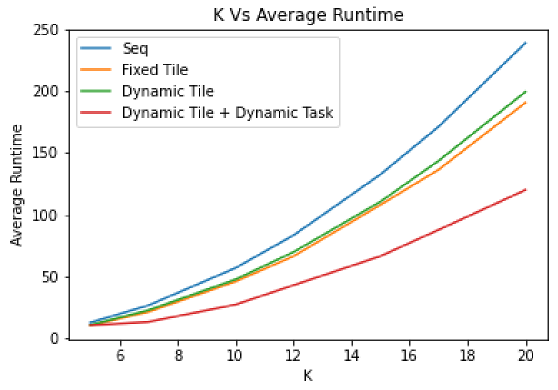


Fig. 5. Graph of K against Average Runtime

Range of values for sequence length	Average runtime across 3 runs (s)	
	Dynamic Tile	Dynamic Task + Dynamic Tile
30000 - 40000	348.4	239.3
20000 - 40000	271.8	186.7
10000 - 40000	170.2	123.3
0 - 40000	104.4	79.5

TABLE III
AVERAGE RUNTIME WHEN ALL K SEQUENCES ARE MADE UP OF SEQUENCES OF DIFFERENT LENGTH

B. Analysis

The experiments yielded a set of interesting results.

In the first experiment, we observe that by parallelising the table generation mechanism of the Needleman-Wunsch alogrithm we are able to achieve a lower running time as compared to the pure sequential implementation. Both the fixed tile implementation and dynamic task implementation has halved the running time. However, the significantly higher running time at high sequence lengths observed in the dynamic task implementation was unexpected, since both implementations share the same underlying logic for computing alignments for the sequences. A possibility for the reduced performance may stem from the overhead required in the task allocation mechanism. As mentioned above, information about the sequences (string data, memory requirements etc) are loaded into a map where task allocation decisions will be performed on. This, in turn, requires a large quantity of data to be initialised during program execution. Hence, we experience a drop in performance for the implementation with dynamic tiles as a resukt of the expensive startup cost.

The results obtained from the second results were what we expected when developing the program. Here, we observe all parallel implementations perform better than sequential implementation, with the dynamic task implementation perform better than the rest. This can be attributed to more work being processed concurrently in the dynamic task implementation.

In the final experiment, we compared two implementations on a set of sequences that are of different length. While the implementation with the dynamic task allocation mechanism performed better among the two, the margin between the performance of the two is smaller than expected. Currently, no work has been done to attempt to explain the observation, but we believe that it may be due to the expensive startup cost required by the dynamic task implementation that we alluded to in the previous paragraph.

IV. Conclusion

Although we have a achieved a decent speedup in our proposed implementation, a number of improvements can be considered and included into the current implementation. Improvements can be made to reduce the computational cost of setting up the map for task allocation in the dynamic task implementation. Additionally, the performance of the thread allocation mechanism for each individual sequencing problem can be increased if a better policy were to be adopted. All in all, this project has been a great stepping stone in exploring and experimenting with OpenMP, as well as to refine our udnerstanding of developing parallel code.

Appendix

Configuration	Description
Seq	Sequential implementation as given in skeleton code
Fixed Tile	Parallel implementation where parallel table generation utilises a fixed partition size for every type of sequences
Dynamic Tile	Parallel implementation where partition sizes are dynamically determined based on sequence length
Dynamic Tile + Dynamic Task	Similar to Dynamic Tile, but will parallel task allocation

TABLE IV
Implementation Description

Scenario	Description
Fix K, Vary Length	Fix the number of sequences to be compared, k to be 2 and vary the lengths for all sequences, such that elements of all sequences are of the same length
Fix Length, Vary K	Fix the length of all sequences to be 10000 and vary the value of k
Vary Range of Values in K Sequence	Fix the value of k to 10 and all sequences can have a length value as specified in the range. Different sequences can have different lengths.

TABLE V
Experiment Description

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970. DOI: 10.1016/0022-2836(70)90057-4.

[2] D. Tang and G. Gupta, "An efficient parallel dynamic programming algorithm," *Computers Mathematics with Applications*, vol. 30, no. 8, pp. 65–74, 1995. DOI: 10.1016/0898-1221(95)00138-o.

[3] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, and V. Mirrokni, *Massively parallel dynamic programming on trees*, 2021. [Online]. Available: https://www.arxiv-vanity.com/papers/1809.03685/.