Concurrent Programming Project Report

Emily Owens and Josh Weaver

We created sequential and concurrent implementations of DNA Sequence Alignment, the goal of which is to identify how similar two genes are to each other. We start with a sequence, and return a list of similar genes found in a set of genes. This set is not provided by the user, but rather is a pre-existing database of gene sequences. For the list of similar genes, we indicate where the similarity occurs in the respective DNA sequences, and provide a score which indicates their level of similarity.

Sequential Implementation

Our sequential implementation works with our test Derby database containing 100 rows. Each row in this database has three columns - id(integer), geneName(string), and sequence(string). The strings vary in size from 5000 to 20,000 chars.

Our implementation starts by first defining a 'target' sequence. This is the sequence we start with, and we compare it to every sequence in the database in order to find the most similar string. After defining the target, we access the database mentioned above. It then pulls out the very first row in the database. From this row, we store both the geneName and sequence in local variables. Next, we begin to split the sequence into 'k-mers'. These k-mers are all the possible substrings the length of our target sequence. We will then calculate the alignments for each of these k-mers.

The alignment process starts by creating a 2-dimensional array which will compare individual chars of the two strings. The array is then filled, indicating the longest common subsequence, and where gaps need to be placed to keep each string the same length. Then, using this constructed array, we create the new aligned sequences, and assign them a score based on similarity. In order to store the results of the previous action, we have created class named AlignResult. AlignResult stores the original geneName, the two aligned sequences, and the score of the alignment.

Once we have the AlignResult for a given k-mer, we may place it into our initialResults list. This is a LinkedList which will contain the best AlignResults from a single row in the database. The AlignResult is added to the head of the initialResults list if one of two conditions is satisfied: The list is either empty, or the score variable for the AlignResult is greater than or equal to the score of the current head of the list. The previous steps are repeated for each k-mer of a single row of the database, and then we clean the list up. After all k-mers have had a chance to be entered into the list, we iterate through the list and remove all AlignResults with a score lower than the maximum.

Then, we add the entirety of the initialResults list to another LinkedList named finalResults. This list contains only the best alignment scores found in the entire database, and may contain a score from every row in the database. We add to, and clean up, the finalResults list using the same requirements as the initialResults list. Finally, the list is returned and the program terminates.

Concurrent Implementation

There are a few different ways we parallelized this algorithm. We have producer threads which pull a row from the database, and insert them into a shared queue. We also have Splitter threads which take an entry from the queue and calculate the best alignment for that gene.

These Splitter threads will perform the k-mer splitting action, and generate a few Aligner threads each. The k-mers are also placed in a queue, and the new Aligner threads pull k-mers from the queue and calculate the k-mer's alignResult. Each Splitter Thread will have its own initialResults CleanLazyList, which each of the Aligner threads access. initialResults has elements both added to and removed from it concurrently.

Similarly, the finalResults list is accessed by multiple Splitter threads at a time. We keep the top AlignResults which are within 10 points of the maximum score, as all these alignments may be of interest. Access to the finalResults list continues until all genes in the database have had their alignment calculated. Then, like the sequential alignment, we return the finalResults list to the user.

Difficulties

We ran into many issues in writing our concurrent implementation. One of the first difficulties we had was with our threads overwriting each others variables, which gave us incorrect results and many null pointer exceptions. We solved this by making every variable the threads use ThreadLocal variables. This was a brute force method that eventually caused us to have a reduction in performance, so we had to go back through and figure out which variables needed to be ThreadLocals and which ones did not.

We also ran into issues with our initialResults and finalResults arrays. Initially we were using LinkedLists like in our sequential implementation, however we quickly realized that they would not work because they were not threadsafe. We decided to use the CleanLazyList that we used in Homework 3 with some modifications so that it could take in AlignResult objects and fetch a specific index of the list.

Performance Testing

We compared our Sequential implementation's runtime to that of our Concurrent implementation under a few different circumstances. We compared against two different databases: testtable3, which had a total of 100 entries of 1000 nucleotides(nts) length each, and realtable2 which had a total of 100 entries of between 1000nts and 5000nts length each.

Realtable2 is more representative of a real-life database scenario, and so the results found from it are more significant in our minds than those found from using testtable3. Additionally, our baseline for comparison are the results found when testing 25 entries in realtable2 against a target value of length 500nts using 4 Splitter Threads(4 Aligner Threads per Splitter). Unless otherwise mentioned, assume all reported tests were performed using the above parameters.

Testtable3:

Sequence Length - 50 DB entries

We evaluated the difference in performance due to length of the target sequence. The Concurrent implementation performed about 30% better on average for lengths of 100 and 500 nts, but performed approximately the same with the sequential for length of 750nts.

This implies that the major performance effecting factor is not size of the final alignment, but rather the number of different kMer alignments.

Results are found in Table1.

Number DB entries - Target 500nt

We also evaluated the performance as the number of DB entries compared increased. All were compared against a target sequence with a length of 500nts. Here, the Concurrent Implementation performed about 16% better on average in all cases.

This implies that while increasing the number of DB entries does increase the overall completion time, the number of DB entries alone does not significantly affect performance when comparing the Sequential and Concurrent implementations.

Results are found in Table2.

Realtable2:

Number of Splitter Threads

We evaluated the performance as we changed the number of Splitter threads (spawning 4 Aligner Threads each). Here, both Concurrent implementations are drastically improved, about 91% faster, compared to the Sequential implementation. Additionally, using four Splitters resulted in a 3% faster runtime than using only two.

This implies that increasing the number of Splitters past a certain point is not a very significant factor.

Results are found in Table3.

Number Aligner Threads

We also evaluated the performance as we altered the number of Aligner Threads, while keeping constant four Splitter threads total. Both Concurrent implementations were drastic improvements, about 91% faster, over the Sequential Implementation. Eight Aligners resulted in about 6% better performance than four Aligners.

This implies that increasing the number of Aligners after a point is not a very significant factor, although it seems more significant than increasing the number of Splitters.

Results are found in Table4.

Conclusion

When all the DB sequences are the same length, we see a relatively small increase in performance from the Concurrent implementation. However, the performance increase when using a DB of varying sequence length is significant. There are two main reasons that this may have occurred: The increased size of some of the real DB sequences, or the fact that the Concurrent implementation can process a longer gene sequence while also processing smaller sequences and there is not a huge sequential bottleneck effect on it.

They are both likely factors, although due to time and hardware constraints, we were unable to perform additional tests to determine the relative effects.

Ultimately, we conclude that our Concurrent implementation is a significant improvement over the Sequential implementation in likely real-life DB simulation. And, although it may not perform significantly better in cases of small sequences of constant length, an improvement overall.

TABLES

Target Sequence Length	Sequential Time(ms)	Concurrent Time (ms)
100	8,494.5	5,385
500	50,708.5	38,783
750	49,404	49,416

Table1. Using testtable3, we evaluated the difference in performance due to length of the target sequence. The Concurrent implementation performed better for lengths of 100 and 500 nts, but was approximately even with the sequential for 750 nts.

DBLength	Sequential Time(ms)	Concurrent Time (ms)
25	24,538	22,140.5
50	50,708.5	38,783
75	77,424	66,888

Table2. Using testtable3, we evaluated the performance as the number of DB entries compared increased. All were compared against a target sequence of 500nts. Here, the Concurrent Implementation performed better in all cases, but it's performance increase was less marked when comparing only 25 entries.

Number of Splitter Threads

	Time (ms)
Sequential	1,231,827
Concurrent (2s's)	125,334.5
Concurrent (4s's)	122,307

Table3. Using realtable2, we evaluated the performance as we changed the number of Splitter threads (spawning 4 Aligner Threads each). Here, both Concurrent implementations are drastically improved compared to the Sequential implementation. Additionally, using four Splitters resulted in a faster runtime than using only two.

Number of Aligner Threads

	Time (ms)
Sequential	1,231,827
Concurrent (4a's)	122,307
Concurrent (8a's)	114,835.5

Table4. Using realtable2, we evaluated the performance as we altered the number of Aligner Threads, while keeping constant four Splitter threads total. Both Concurrent implementations were drastic improvements over the Sequential Implementation. Eight Aligners resulted in better performance than four Aligners.