CSC357 Final Report
Andy Baay

Abstract:

The goal of this project was to implement a parallel algorithm for comparing tree structures as part of larger thesis work. My thesis focuses on comparing glycan structures for similarity in order to rank them as search results for a database query engine that is operated by the Swiss Institute for Bioinformatics. This thesis addresses an access problem in the glycomics community by increasing the availability of previous research through enhanced structure searching. This project makes my thesis more robust and extends its ability to handle comparisons between the large structures that are being generated by enhanced research methods today. The result of this project is code that can successfully calculate the difference between two sufficiently large trees in less time than the sequential implementation.

Previous Work:

This project is based on Zhang and Shasha's "Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems," as well as code forked from https://github.com/ijkilchenko/ZhangShasha. The repository contains a sequential implementation of the algorithm that was ultimately used to benchmark the parallel implementation. The final test of the parallel implementation also relies on a GlycoCT formatted file parser that I built over the course of this project to help load more complex test structures.

Algorithm:

The algorithm itself relies on a post-order indexing of the two input trees and the identification of the so-called "keyroots. " These keyroots are either the root node of the tree or the greatest unique ancestor of a left leaf node. These keyroots are used to simplify the number of computation completed when comparing two trees. By calculating the distance between sub-trees rooted at two keyroots, the dynamic programming algorithm guarantees the availability of this score for the calculation of larger trees in which the sub-tress are members. It is easy to see how computation time is saved by the guaranteed availability of sub-tree comparison scores.

To see a complete sequential implementation of this algorithm, please see the repository referenced above. To help illustrate the space complexity, suppose we have two simple trees T1 and T2 that will be rooted at *a* and *e* respectively. *a* has two children: *b* and *c. e* has two children: *f* and *g,* such that the trees look like this:

To compare these two trees we need a large array that corresponds to the mapping of every sub-tree rooted at the specified node in T1 to the sub-tree rooted at the specified node in T2. This would look like this:

Large Array

(,)	**(a,)**	**(b,)**	**(c,)**
(,e)	**(a,e)**	**(b,e)**	**(c,e)**
(,f)	**(a,f)**	**(b,f)**	**(c,f)**
(,g)	**(a,g)**	**(b,g)**	**(c,g)**

Temporary Array

0	1	2	3
1			
2			
3			

Then, to calculate these values a temporary array is required at certain node pairs (where the *i* and *j* are both keyroot members) as indicated above on the right.

Parallel Implementation:

The sequential implementation was broken into phases as described in Zhang and Shasha. This breakdown is as follows:

Phase 1 – Initialize the large array to hold the calculated scores between each the disjointed forests created by sub-indexing tree T1 at *i* and tree T2 at *j*. Then for each index (*i,j*) that corresponds to keyroot indices in T1 and T2, initialize a temporary array for calculating the score at these positions.

Phase 2 – In each of the initialized temporary arrays for T1 and T2 keyroot index pairs, calculate the score that corresponds to each node of T1 mapping to a blank node of T2. In terms of the array, this represents the top bounds of the edit distance. This is performed in k waves where K = 1->size(T1). The left array below shows which indexes are calculated on wave K. The temporary array for scores between 2 trees of size 3 would then look like the array on the right. Because this has to be done for all the temporary arrays within the larger distance array, each respective temporary array can be worked on in parallel during a single step of K. We must block for the step K-1 to finish before K runs, since K depends on the value of K -1.

0	**K1**	**K2**	**K3**

0	1	2	3

Phase 3 – In each of the initialized temporary arrays for T1 and T2 keyroot index pairs, calculate the score that corresponds to each node of T2 mapping to a blank node of T1. In terms of the array, this represents the left bounds of the edit distance array. This is performed in k waves where K = 1->size(T2). The left array below shows which indexes are calculated on wave K. The temporary array for scores between 2 trees of size 3 would then look like the array on the right. Because

this has to be done for all the temporary arrays within the larger distance array, each respective temporary array can be worked on in parallel during a single step of K. We must block for the step K-1 to finish before K runs, since K depends on the value of K -1.

0	1	2	3
K1			
K2			
K3			

0	1	2	3
1			
2			
3			

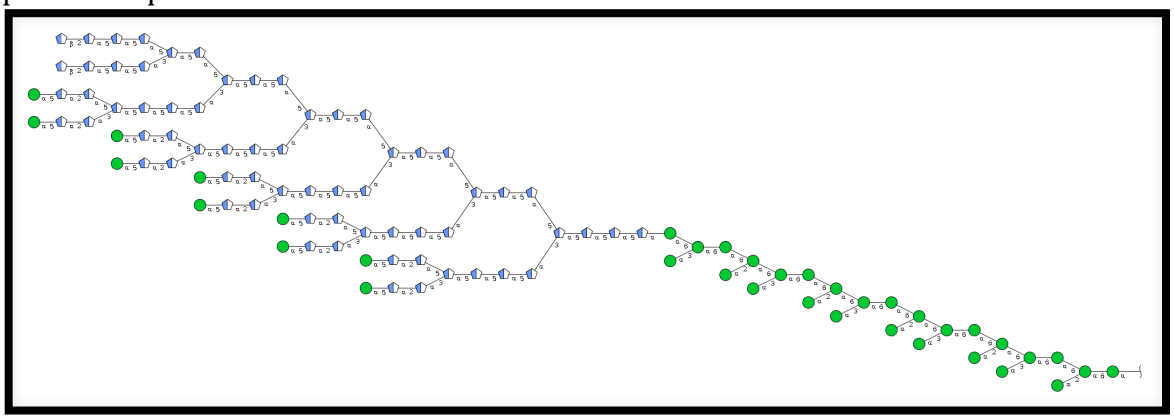
Phase 4 – To fill in the rest of the array, we run K = size(T1) + size(T2) -2 waves in which each internal distance is calculated. The array below shows on which iteration of K each distance would be calculated. Each distance requires the value of the indexes (i-1, j-1), (i, j – 1), and (i-1, j). This can be run in parallel on all temporary arrays so long as we block after each iteration of K.

0	1	2	3
1	**K1**	**K2**	**K3**
2	**K2**	**K3**	**K4**
3	**K3**	**K4**	**K5**

Results:

For small trees (<20 nodes) there is no speed advantage in the parallel algorithm compared to the sequential algorithm. This is because the overhead required to start a thread pool, submit tasks, and check for completion consumes any gains that come from parallel calculations.

However, when I test the parallel implementation with two large glycan structure containing about 100 nodes each, I recorded a 5X speed increase. Two features can explain this. Firstly, there more temporary arrays requiring computation for each step K. Second, each temporary array will have a greater diagonal length, which represents the number of task that can be complete in parallel in phase 4.



5X speedup was achieved comparing structures of this size.

Conclusion:

The implementation of a parallel tree comparison algorithm will contribute to my thesis work and add scalability to the final product. Though I was not expecting to see a speed increase over the sequential implementation when dealing with small trees, the fact that a speed increase can be seen with trees of only 100 nodes is exciting. As research techniques continue to improve, so too will the ability of researchers to observe larger and more elaborate structures. This final project has increased my understanding of the tree comparison algorithms as well as the implementation of thread pools. It was also interesting to see the difference between the sequential and parallel implementations.