Load Balancing Issues with Constructing Phylogenetic Trees using Neighbour-Joining Algorithm

To cite this article: S M Al Mamun 2012 J. Phys.: Conf. Ser. 341 012017

View the article online for updates and enhancements.
Load Balancing Issues with Constructing Phylogenetic Trees using Neighbour-Joining Algorithm

S.M. Al Mamun
Department of Computer Science
University of Manitoba
Winnipeg, MB, Canada
E-mail: mamun@cs.umanitoba.ca

Abstract. Phylogenetic tree construction is one of the most important and interesting problems in bioinformatics. Constructing an efficient phylogenetic tree has always been a research issue. It needs to consider both the correctness and the speed of the tree construction. In this paper, we implemented the neighbour-joining algorithm, using Message Passing Interface (MPI) for constructing the phylogenetic tree. Performance is efficacious, comparing to the best sequential algorithm. From this paper, it would be clear to the researchers that how load balance can make a great effect for constructing phylogenetic trees using neighbour-joining algorithm.

1. Introduction
A phylogenetic tree shows the evolutionary relationship among the biological species. There are several well-established strategies to build a phylogenetic tree. Neighbour-Joining algorithm is one of them, which is frequently used to reconstruct a phylogenetic tree based on the distance between each pair of taxa, where taxa is group of organisms. Parallel implementation of this algorithm can make computationally efficient tree for large amount of data. Constructing the phylogenetic tree with high performance has become one of the major problems in bioinformatics. As the sequence becomes larger, huge amount of time is needed to calculate. For example, an input with 50,000x50,000 may not be even computed within several days. If we use the same input for parallel implementation, it would be done by a single day. The tree construction might be finished by several hours even if the code is highly optimized.

2. Background and Related Work
Saitou et al. [1] first designed neighbour-joining approach, which is an iterative algorithm. Several multiple sequence alignment package use neighbour-joining algorithm. Thompson et al. [2] used neighbour-joining algorithm in ClustalW, which is a widely used tool for aligning multiple sequences. Li [3] developed a parallel version of ClustalW, using message passing interface. RapidNJ [4] and NINJA [5] are another two implementations of neighbour-joining algorithm. Stamatakis et al. [6] developed another package named RaxML for phylogenetic tree construction using neighbour-joining algorithm. Sheneman et al. [7] designed a faster version, named as Clearcut which is based on relaxed neighbour-joining algorithm. Liu et al.[8]
implemented a new approach with neighbour-joining using CUDA. A lot of works are already done with phylogenetic trees using parallel neighbour-joining approaches, but to the best of my knowledge this is the first approach where load balancing is specifically focused.

3. The Neighbour-Joining Algorithm

Neighbouring algorithm is a hierarchical clustering algorithm. It takes a distance matrix, D as input where \( d_{ij} \) is the distance between two clusters named i, j. Neighbour-joining algorithm repeatedly joins pairs of closest clusters. This repeating process continues until the tree is completely resolved.

In some of the previous works, researchers used biological sequences as input. From the sequences, they calculated the distance matrix. On the other hand, designers directly provided distance matrix in some other implementations. In this paper, we emphasized on the neighbour-joining algorithm only and considered distance matrix as direct input.

4. Solution Strategy and Implementation

We implemented the parallel neighbour-joining algorithm in C language using Message passing interface (MPI). At the A NxN distance matrix is used as input where each value is a distance between two species. The root processor reads the size of matrix and broadcasts it to other processors. The matrix was distributed by column-wise to the processors. Here, we faced a peculiar complexity when the datasets were not equally divisible by the number of processors. As a result, we used MPI_Alltoall instead of using MPI_Alltoallv. There might be curiosity why we did not use other libraries like, OpenMP. Because OpenMP performs well only in case of for loops. On the other hand, MPI is an excellent choice as it can handle distributing and gathering data, utilize the perfect flavor of a parallel programming. However, each processor then computes the local sum (which is denoted by \( r_j \)) and exchanges the sums with MPI_all-to-all. Again, all the processors search a minimum distance, \( D_{i,j} \) and exchange their value with another all-to-all MPI call. A global minimum distance is calculated to construct the tree. Each processor computes the node, comparing with all of it’s distances and the received minimum distance. At the end, root processor draws the tree as described in the previous section.

5. Experimental Framework

Programs were run in the Helium machine, which is a cluster of Linux servers. It consists of one head node(Sun Fire X4200 machine) and five other computing nodes(Sun Fire X4600 machines), operating system is CentOS 5. Here, head node is a dual core 2.6GHz AMD Opteron processor.

![Figure 1. Processors Vs Time(in seconds)](image-url)
with 2GB memory and each computing node features 8-processor dual-core cache-coherent Non-Uniform Memory Access Symmetric Multiprocessing (ccNUMA SMP) computing model with a dual core 2.6GHz processor and 32GB memory.

6. Results
Several distance matrices were used as input. We used 3 matrices of size 500x500, 1130x1130 and 2700x2700 from http://www.daimi.au.dk/~cstorm/courses/AiBTaS_e05/project2/. For these three datasets, the runtime for different parameters was quite similar. That’s why we did not put any graph that would show the mean runtime for the three datasets.

When the load per processor is too high, the performance is poor. As the load decreases, performance gets better. Again, when the load per processor is too low, the performance becomes poor. That means, an average load gives the best performance. In figure 1, different timings are plotted for all these 3 datasets and also for the sequential program. For sequential program, we used quicktree [9] as it can build the tree amazingly faster than the other existing sequential implementations. For the same three datasets, quicktree gives the execution time of 0.55 second, 2.91 seconds and 27.11 seconds.

Figure 2 and 3 show the speedup and efficiency for various number of processors. In figure 2, the speedup is highest for 10 nodes. Then, speedup starts decreasing. In figure 3, efficiency also decreases with the number of increasing nodes. But it is reasonable up to 10 nodes. After that, it becomes worse. As, it is highly iterative algorithm, the speedup and efficiency may be a bit low. In figure 4, the cost for different datasets is shown. The same case happens here as like efficiency, up to 10 nodes costs are reasonable. For more than 10 nodes, costs have been
increased drastically. It means that for neighbour-joining algorithm whatever the matrix size is, we will get the best performance for a certain number of processors.

7. Conclusion and Future Work
We implemented the simplest version of neighbour-joining algorithm. From the experiment and comparison with the performance of quicktree, we can see in figure 2 that when each processor is responsible for a reasonable amount of workloads, it gives the best performance. Only then, the highest possible speedup can be gained. In future, we will be working on it so that it can handle when data size can not be divided by the number of processors. We will also apply the hybrid programming techniques to examine the load balancing issues with constructing phylogenetic trees using neighbour-joining algorithm.

8. Acknowledgments
I would like to thank Dr. Parimala Thulasiraman for all of her helpful guidelines and feedbacks starting from the beginning through the end of this implementation. I would also like to thank Dr. Michael Domaratzki who encouraged me to face the challenge and suggested various guidelines during the implementation.

References
[1] Saitou N and Nei M 1987 Mol Biol Evol. 4 406–425
[2] Thompson J D, Higgins D G and Gibson T J 1994 Nucleic Acids Research 22 4673–4680
[3] Li K B 2003 BIOINFORMATICS 19 1585–1586
[4] Simonsen M, Mailund T and Pedersen C 2002 Algorithms in Bioinformatics vol 5251 pp 113–122
[5] Wheeler T 2009 Proceedings of the 9th Workshop on Algorithms in Bioinformatics (WABI 2009) pp 375–389
[6] Stamatakis A 2006 BIOINFORMATICS 22 2688–2690
[7] Sheneman L, Evans J and Foster J A 2006 BIOINFORMATICS 22 2823–2824
[8] Liu Y, Schmidt B and Maskell D L 2009 IEEE International Workshop on High Performance Computational Biology (HiCOMB 2009)
[9] Howe K, Bateman A and Durbin R 2002 BIOINFORMATICS 18 1546–1547