panjo: a parallel neighbor joining algorithm

James Bullard
Department of Biostatistics
University of California Berkeley
bullard@stat.berkeley.edu
www.stat.berkeley.edu/~bullard/panjo

May 11, 2007

Contents

Abstract

We describe a parallel implementation of the neighbor joining algorithm frequently employed in phylogenetics to reconstruct evolutionary trees from a pairwise distance matrix. We present an implementation utilizing distributed memory and capable of scaling to massive distance matrices. We assess the performance of our algorithm and describe problems related to load balancing.

1 Introduction

In this report we present a parallel implementation of the popular neighbor joining algorithm used frequently to reconstruct phylogenetic trees. The neighbor joining algorithm was initially proposed by Saitou, N. and Nei, M. [6] and then revised by J.A. Studier, K.J. Keppler [3]. In this report we will investigate the revised algorithm which has been shown to be order $O(N^3)$. The structure of the report is as follows: First, we present some background on phylogenetic trees, second we describe how they are constructed in practice, next we review the neighbor-joining algorithm. We then go on to describe our parallel implementation, we discuss its theoretical "best-case" performance. We investigate the use of possible performance models in an attempt to determine where our algorithm suffers. We present results of running our algorithm. Finally, we conclude with a discussion about possible improvements as well as limitations of our algorithm and future work.

1.1 Phylogenetic Trees

Briefly, phylogentic trees are used to represent the hypothesized evolutionary relationships between organisms. Classically, we think of the phylogenetic tree of the higher primates or the phylogentic tree depicted in figure (1). The tree in figure (1) was constructed by comparing the similarity of a particular gene for every leaf against every other leaf, if two leafs (in this case bacteria) have very similar genes then they will most likely end up very close to one another on the tree.[1] The internal nodes on the tree are then hypothesized ancestors for their children leafs. It should be emphasized that constructing a tree based on genetic sequences is not the only way. We could just as easily construct a tree based on visibly observable features, however for things like bacteria using the genetic sequence data we have available seems like the most intelligent choice.

Bioinformatics has experienced an explosion of data in the last 15 years and this explosion of data has allowed us to examine phylogenies of larger and larger sizes. The software developed for this report was designed with this in mind. Often, one of the main challenges in this field is simply dealing with the size of the data set. The `panjo` software package has been designed to facilitate computations on data sets that might never fit into the memory of a single machine.

Our collaborators currently maintain a database of over 140,000 16S rDNA bacterial genes. This publicly available database can be downloaded at `http://greengenes.lbl.org`. The 16S rDNA gene can be thought of as a bacterial ID tag. The biologists wish to construct a phylogenetic tree for all of the species in the database - in this way they might better understand the process of evolution in the bacterial world[1].

1.2 Building Phylogenetic Trees

Many methods have been proposed to build phylogentic trees. Broadly these methods can be viewed as either "Maximum Likelihood" methods or "Distance Matrix" based methods[2]. The maximum likelihood based methods have a much more rigorous probabilistic underpinning, but unfortunately they often suffer from exponential scaling, thus their usage on large trees is generally not feasible. A current best guess as to the number of sequences which one could possibly hope to assemble using the fastest available software employing maximum likelihood methods is on the order of tens of thousands of sequences[8].

[1] Exactly what "similar" genes means is a highly researched field and dealing with this is not the focus of the report, rather we assume we can calculate some measure of closeness for two leaf nodes.

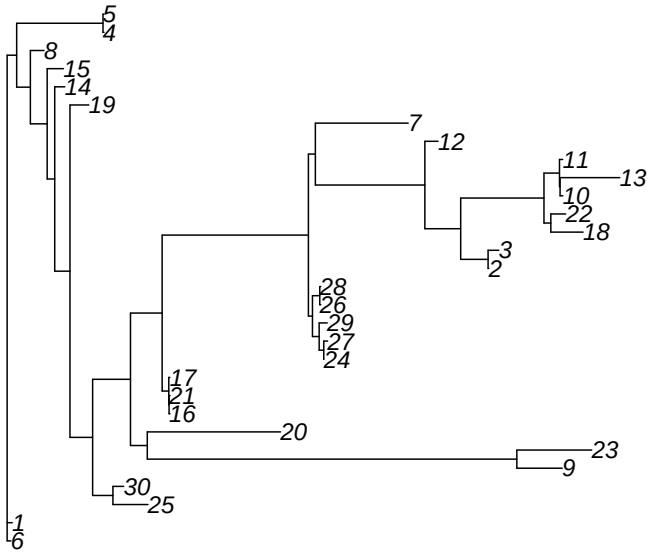


Figure 1: A phylogenetic tree. We have constructed this relatively tiny tree using the `ape` package in R from a subsample of the bacterial database. The numbers at the leafs represent genetic sequences from the database of over 140,000 bacterial 16S rDNA sequences.

Essentially, maximum likelihood methods assume a stochastic model for the evolution of organisms, under this stochastic model one can compute the likelihood of a given tree. All one needs to do then is generate a set of trees, compute the likelihood for each tree, and then see which tree gives you the highest likelihood. The problem then comes down to sampling the tree space. We don't believe it is necessary to convince the reader that the number of possible trees for 140,000 leaf nodes is a _really_ big number.[2] Although we would love to be able to compute a maximum likelihood tree we have to accept that it might not be final project material.

The second class of methods do not suffer from such poor scaling. These methods utilize a distance matrix to define pairwise distances between any two sequences. This distance matrix becomes the focal point of the computation. These algorithms have their own problems as we shall see. Firstly, they generally have least $O(N^2)$ space requirements. The most popular algorithm neighbor joining is $O(N^3)$ in time complexity and $O(N^2)$ in space complexity. There have naturally been improvements to these constraints, but generally at the cost of relaxations of the algorithms [7]. The trees constructed via this method are generally believed to be less accurate at reconstructing the "true" tree and therefore most recent research has focused on maximum likelihood methods. In our case, however

[2]I am being a bit imprecise here; one clearly needn't sample from the entire tree space, but nevertheless if one wishes to consider a reasonable region of the tree space then a non-trivial amount of trees need to be sampled. Furthermore, the problem of computing the likelihood while certainly tractable is still somewhat time consuming.

we would never truly "believe" a tree with 140,000 leafs constructed using either class of methods as both impose assumptions that are often practically violated.[3]

Another nice feature about the neighbor-joining algorithm is that it can produce a starting point for maximum likelihood methods. We can produce different trees by perturbing the input and then a "best" tree can be chosen among the set by choosing the tree which maximizes the likelihood under some evolutionary model.

1.3 The Neighbor-Joining Algorithm

We first review the neighbor-joining algorithm. The algorithm begins with an $N \times N$ pairwise distance matrix called M. This distance matrix is symmetric with 0's along the diagonal. A tacit assumption with this approach is that a reasonable distance matrix can be computed. How to calculate the distances between two genetic sequences is not the focus of this report, however we would like to point out that this is a critical step at producing a sensible tree and much research has been focused on this problem. Another area of the problem which we will not focus on will be "why" the neighbor joining algorithm is sensible. The interested reader can find an excellent starting point in [2].

In (1.3) we present pseudocode for the neighbor joining algorithm. In words, we begin with N clusters (these are our leaf nodes) - we wish to join these successively. We first "normalize" our distance matrix producing a new distance matrix D. We compute the column means r_i as in equation (1).

$$r_i \triangleq \frac{1}{1 - |\text{Active}|} \sum_{k \in Active} m_{i,k} \tag{1}$$

Using these "column means" we next construct a new matrix D which contains the "normalized" distances.

$$D_{i,j} = m_{i,j} - (r_i + r_j) \tag{2}$$

In equation (1) Active is simply the current columns of the matrix upon which we are operating (the clusters which we still need to join), Active is initialized to N and at each iteration we remove 2 from the Active list and add a new node which becomes the parent of these two removed nodes.

Once we have the new matrix $D_{i,j}$ in hand we search for a minimum $D_{i,j}$ among the entire matrix (actually due to symmetry we need only search along the lower diagonal). These i, j are the two nodes which become siblings in the tree. We construct a parent, remove i, j, and then add a new node U using the distance calculations specified explicitly in the algorithm. The new node U will get placed in the distance matrix and we will then recalculate the distance from U to every other node in Active. We arbitrarily place U in the spot of j. This means that we mark row, column i as invalid and all successive computations ignore values at i.

We can see that the algorithm is quite simple. We only need to keep track of which entries in the matrix are active and then iteratively remove them until we are left with only two clusters (branches), we join these branches and we have a tree.

We pause here for a moment to discuss some aspects of this algorithm which may make an efficient parallel solution difficult. The first observation is that the computation begins on an $N \times N$ distance matrix and then ends with essentially a 2×2 distance matrix (the final two clusters which we need to join). This means at each successive iteration each processor does less and less "real" work. We

[3]Many of these assumptions are probably more frequently violated when considering bacterial sequences, therefore we would be much more comfortable saying that panjo produces a hierarchical clustering and under certain assumptions this hierarchical clustering can be viewed as an estimate of the phylogenetic tree.

begin at the leaves and proceed to the root, but since we do not know which leaves will be siblings we cannot partition the computation in an elegant way.[4] For now, we leave it at that and proceed to discuss our parallel solution - there will be many "practical" improvements which one can make, but we will always be bound by the above observation in this report.[5]

NeighborJoin(M)

InitTree(M): ▷ Here simply initialize the tree with all the leaf nodes.
  for $\nu_i \in$ nrow(M)
    do
      ADD(ν_i, *Tree*)

addNode(U, (i, j)): ▷ Here we add node U to the tree calculating
  $m_{k,i} = \frac{1}{2}(m_{i,j} + r_i - r_j)$ the edge lengths to its children.
  $m_{k,j} = m_{i,j} - m_{k,i}$
  add(U, *Tree*)

minD(): ▷ Here we compute normalized distances and
  $r_i = \frac{1}{1-|Active|} \sum_{k \in Active} m_{i,k}$ then find the minimum.
  $D_{i,j} = m_{i,j} - (r_i + r_j)$
  min($D_{i,j}$)

Tree = InitTree(M)
Active = fringe(*Tree*)

while (| *Active* | > 2)
    do
      (i, j) = minD() ▷ The algorithm successively removes two nodes
      addNode(U, (i, j)) and then adds their parent.
      remove((i, j), *Active*)
      add(U, *Active*)

2 Methods

2.1 Parallel Implementation

We now turn to a discussion of our parallel implementation. Our implementation is in the C language using MPI. We will refer to the algorithm listed in (1.3). First, we again assume an $N \times N$ distance matrix. This distance matrix is stored on disk because it is too big to fit in memory. The root processor reads the size of the distance matrix and then broadcasts that size to all other processors. Each processor then reads the appropriate piece of the distance matrix from disk. The matrix has been divided into block-column layout (see figure 2). and the matrix is stored in column major format on each processor. We make no demands on the numbers of processors which our implementation

[4] This is an admission of my lack of creativity/intelligence not an admission of an utter impossibility.
[5] Our pseudocode was based on both the book by Durbin et. al. [2] and the software package Grappa [4].

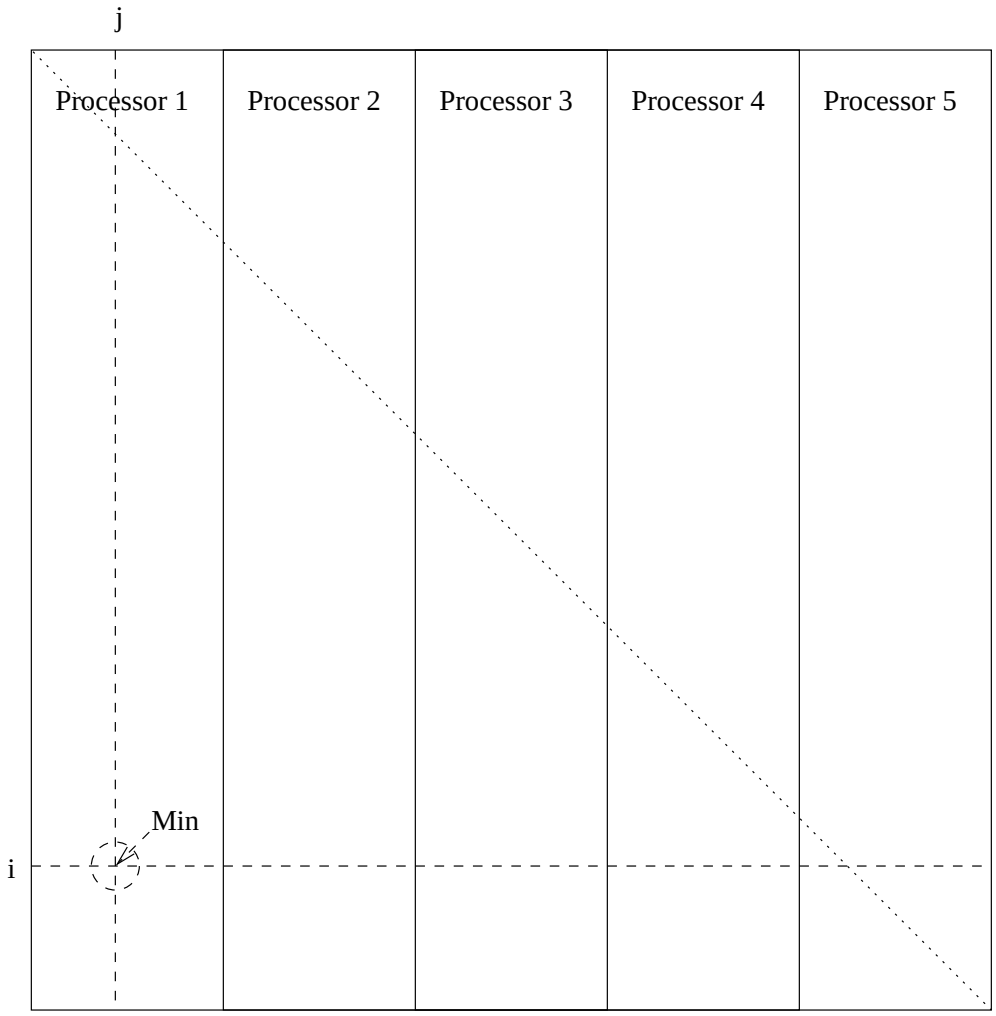


Figure 2: Step 1: The matrix at the beginning of the computation. Each process computes its minimum "normalized" value. The processors exchange and a global minimum value (Min) is found.

supports, however we do demand that the number of columns is divisible by the number of processors. We simply pad the distance matrix with columns and mark them as invalid at the start of computation.[6]

The shape of the algorithm is identical to that of (1.3). We describe the algorithm below dividing each step up into its logical components.

`col-sums`: Each processor computes the column sums r_j for each column it owns. These column sums are then exchanged in an all-to-all MPI call.

`local-min`: Next each processor searches for a minimum $D_{i,j}$ among its block - each $D_{i,j}$ is computed from the exchanged r_j as detailed in (1.3). This search happens only in the lower diagonal. All processors exchange their minimum $D_{i,j}^{\min_p}$ in another MPI all-to-all exchange.

`g-min`: Next each processor decides which $D_{i,j}^{\min_p}$ is the global minimum hence labeled as: $D_{i,j}^{\min^*}$. This

[6]Why? because we investigated using all-to-all MPI communication where one can exchange different size chunks, however because $N >> P$ the amount of padding necessary is small, therefore we decided that it was not worth the extra effort.

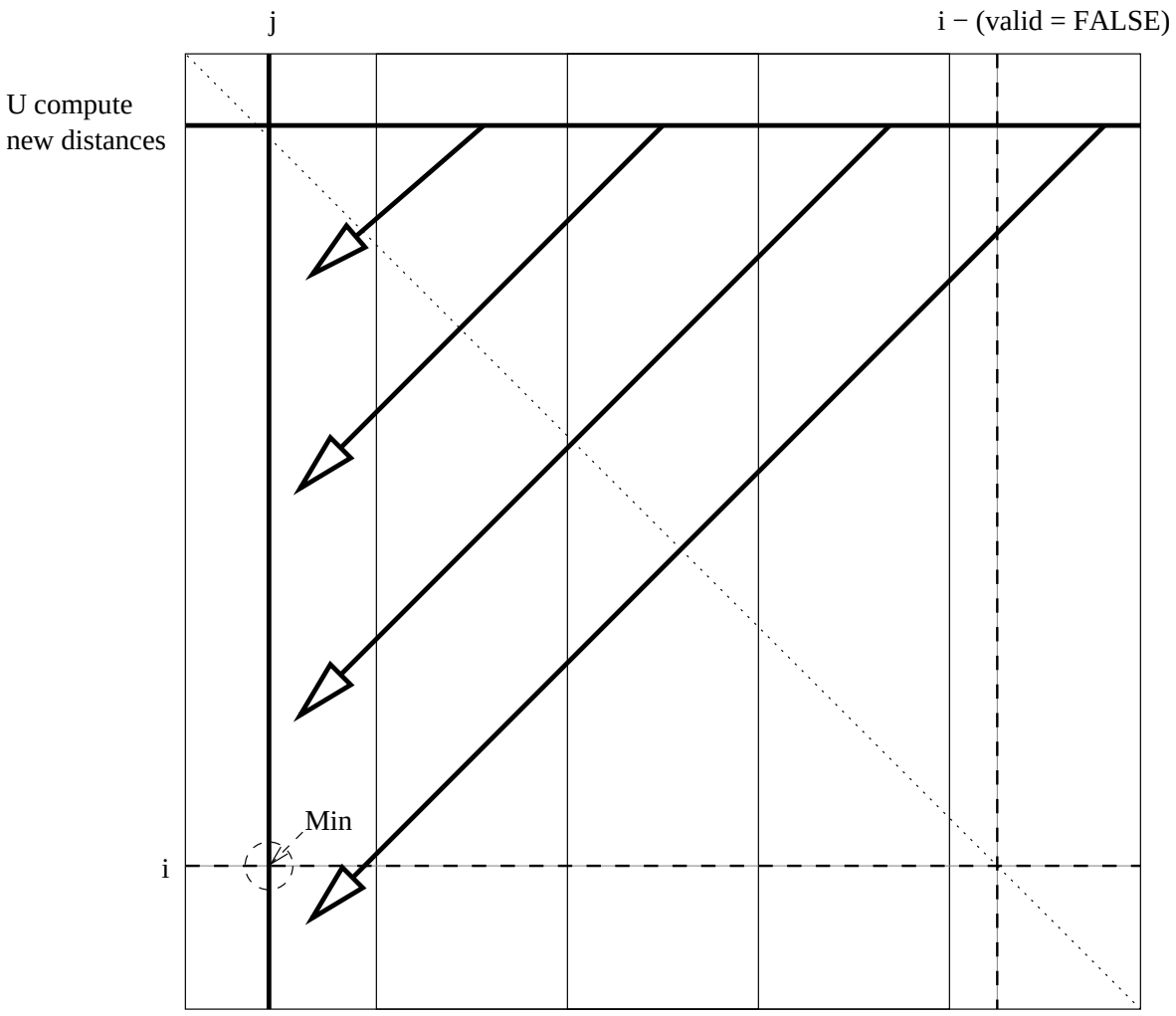


Figure 3: Step 2: The row and column corresponding to i are eliminated from future work. The new node U is added and distances to it and every other node are computed. Each processor computes its corresponding piece and then the pieces are sent to the processor which owns column j; This is effectively a row-column transpose operation.

occurs in a way which guarantees that if the $D_{i,j}^{\min_p}$ are not unique all processors will pick the same one. The processor who owns $D_{i,j}^{\min^*}$ will then retrieve: $m_{i,j}$ from its elements and then broadcast this value to all other processors. This needs to occur so that each processor can compute distances between its nodes and the soon-to-be added node U.

`set-valid`: Each processor now has $m_{i,j}^{\min^*}$ as well as $i^{\min^*}, j^{\min^*}$. Every processor marks valid$[i^{\min^*}]$ = FALSE and this row/column is excluded for the remainder of the computation.

`add-u`: Now each processor has everything it needs to compute distances from its nodes to the new node U. We can see in figure (3) how this occurs on each processor. The new distance is computed across the columns on each processor. In the diagram we see that the global minimum was owned by processor 1. For each processor we compute a new distance between all nodes we own and the new node U (see the 1.3 for how the new distances are computed). We store this distance in row: $j^{\min^*}$ of M - this row/column becomes the new home for node U.

`transpose`: Now to maintain the symmetry of the matrix. we broadcast the results of step `add-u` to the processor which owned node $j^{\min *}$ and is now the new owner of node U. This is illustrated by the arrows in figure (3).

`add-node` Finally, on processor 1 we add a new node to the tree with the appropriate distances.

As we can clearly see the parallel implementation has a number of implicit barriers corresponding to the MPI all-to-all communications. This seems to be necessary based on the algorithm, that is we need to iteratively compute global minimums - at each step we add node U and new normalized distances to this node need to be factored in to the next global minimum search.

2.2 Assessing Performance

Now that we have a better understanding of the parallel algorithm we can better understand just how well we might be able to perform. As mentioned before the neighbor-joining algorithm in its serial form is an $O(N^3)$ algorithm and this is apparent because we have an outer loop over the N initial clusters where at each iteration we need to search over an N^2 object. By parallelizing the algorithm we have reduced the inner matrix search to an order $O(\frac{N^2}{P})$ search because each processor only deals with its particular block which is of size $N \times N/P$. We need to be careful however, because we enforce that $P \leq N$ due to the way we partition the matrix. Practically speaking this does not matter too much as N is generally quite large. However, it means that the best we can hope for is an N^2 parallel algorithm by choosing $P = N$.[7]

A careful reader might notice the lunacy of choosing $P = N$; apart from being impossible except for small matrices or huge clusters each iteration will effectively remove a processor from doing meaningful work. If we could "return" processors to the cluster after we have used them this might not be a bad solution, but we are not aware of how to do this, nor does it appear to be a common idiom.

2.3 Implementation Choices/Regrets

As always implementation choices are made and then often regretted. Our main concern while implementing the algorithm was load-balancing. If we have a sorted matrix then for a sequence of time steps we can imagine that the same processor will remove nodes, thus after a certain amount of time this processor will stop doing meaningful work. The algorithm proceeds from N clusters to 1 so we know that processors will need to stop doing meaningful work at some point, but we could leverage this fact by sending some work to any processor which is not currently performing meaningful computations. We know that at each time step we have removed one node from the computation so the effective size of the matrix is at time step t is given by $N - t$. We can imagine proceeding until $t = N/2$ and then at this step redistributing the load. This could be a very interesting way of ensuring that we are doing more real work at each step.

Another way in which the parallel algorithm is load-imbalanced is via the block-column layout. This is visible in figure (2). In this figure we see that the last processor will have a much smaller piece of the matrix to examine when looking for its local minimum. The suggestion to use a column-cyclic layout seems to be a clear win. The column-block layout makes it easy to arrange the code, but in fact there are no technical challenges to use a column-cyclic layout and we expect to get a big win in terms of load balancing. Unfortunately, due to time constraints we were unable to measure how

[7]This is "limit" of our current implementation. There are many ways to improve this and we will discuss some in the next section.

much load imbalance we have due to this unequal partition. It is clear however that processors will have to wait until all other processor have found their minimums and thus by using a column cyclic layout we will decrease this waiting time.

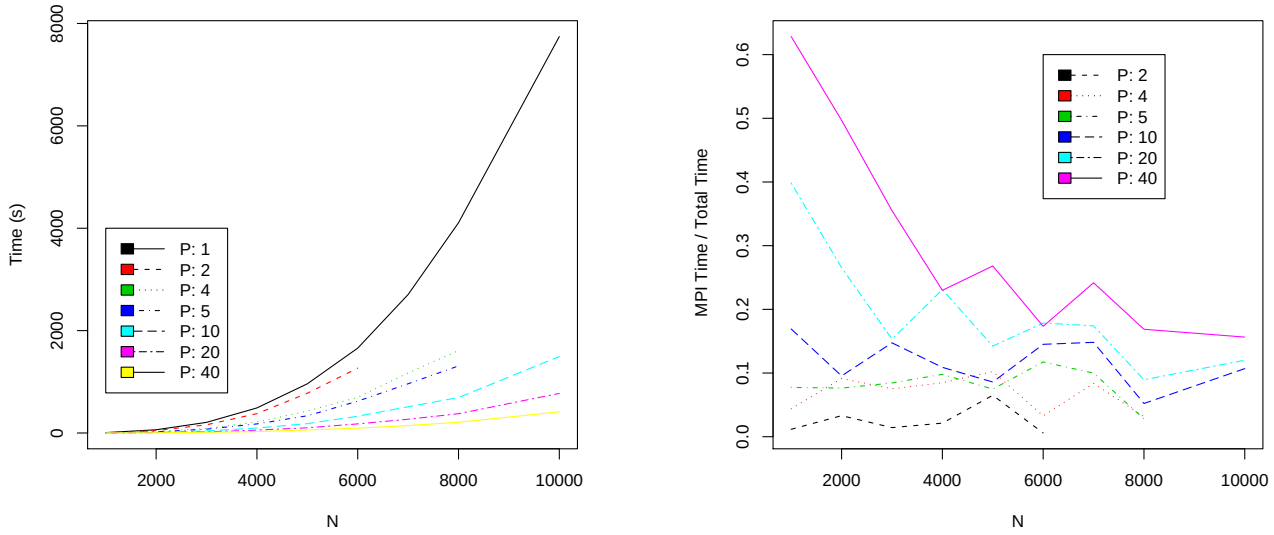
Finally, the choice of MPI as opposed to UPC or Titanium was long-debated. The nature of the problem made a solution using Titanium or UPC very attractive, however since we wanted to produce a solution that would be available to our collaborators we had to fall back to MPI (sadly).[8]

3 Results

3.1 Overall Performance

We have successfully run the code on a number of platforms/clusters, but performance numbers on different clusters are not available. All performance numbers were generated on Jacquard. Jacquard has the following characteristics for its MPI stack: bandwidth: 620 MB/s, latency: 4.5 usec. We expect that the performance will be worse for high latency networks due to the communication pattern, but without data we leave it at that.

We present performance plots for matrices of increasing size from 1000 to 10000. In addition we present performance numbers for matrices of size 20,000 and 40,000.



(a) Total run time for each matrix size and processor combination

(b) Time spent in MPI routines divided by Total run time.

Figure 4: Performance of both serial implementation both with respect to total time as well as the ratio of time spent in MPI to the total time.

In figure (4 (a)) we show the wall clock time for each matrix with a separate line representing the number of processors used. We note that for some larger matrix sizes and smaller processor numbers all matrix sizes were not computed. We did however compute all matrix sizes for 1 processor

[8]It is clear that thinking about the redistribution of work throughout the computations would be much easier to write in a shared memory model as opposed to a message passing model.

to accurately report speedup and parallel efficiency. We can see that in overall performance for $10,000 \times 10,000$ distance matrices we get significant improvements with additional processors.

In figure (4 (b)) we plot the ratio of time spent in MPI versus the total run time. This plot is a little unexpected in its variation. We would have expected a smooth decline approaching some small constant around 0 as we know that as N large we are dominated by the N^2 scaling. In this figure we can begin to see however that as we increase the problem size we begin to hover around .15. We tried to construct a performance model where we could accurately predict this ratio, but this was not too easy because we exclusively use MPI collectives. Knowledge of the implementation of these collectives is critical for properly employing something like an α, β model. In fact we spent a good amount of time trying to fit this model to our data, but we never were able to get sensible results.

3.2 Parallel Efficiency/Speedup

In figure (5) we now present parallel efficiency and speedup plots. We recall that under the observations that there is a smaller amount of parallelism which we might hope to eek out of the algorithm while ensuring correctness to the serial version we are quite pleased with the speedup plots. We also need to declare that our single processor implementation was not "optimized." However, it is comparable to other packages in terms of absolute performance. One nice implementation aspect is that our serial code uses the same codebase as our parallel version with all MPI calls conditionally compiled away.

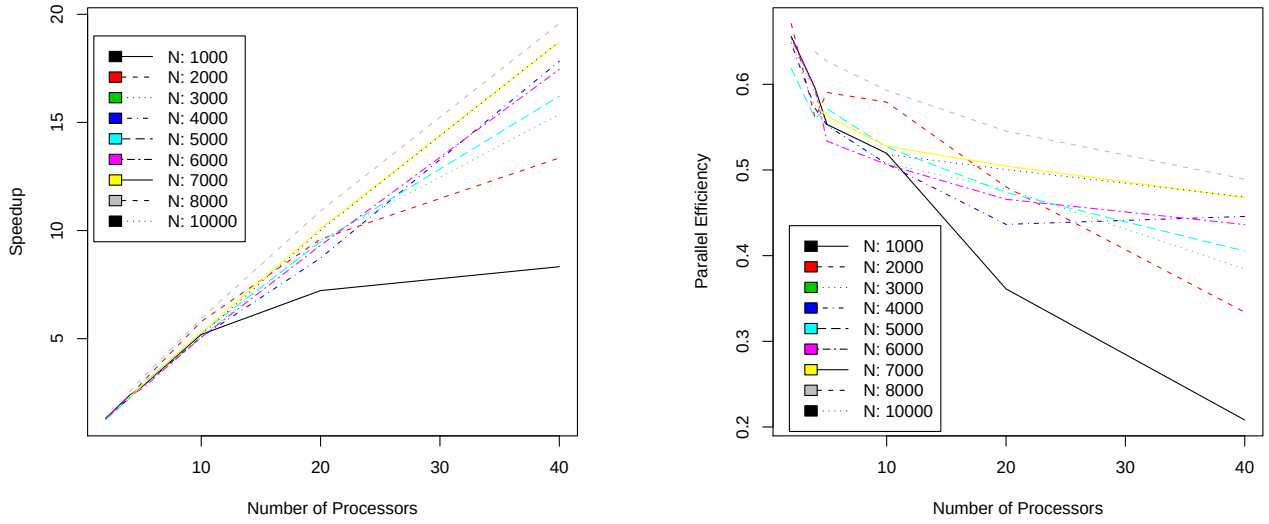
Overall, we see a speedup of about .5 which again pleases us, but the question remains - is this the best we can do? We see that the MPI routines are taking some portion of the computation time, but not that much. So where does the rest of our speedup go? We know that as we remove nodes the computation effectively stops using processors so as N goes to 1 we linearly decrease our "effective" processor usage. If we have $P = N$ at each step we remove a processor from doing meaningful work so really we are not using P processors but rather $P/2$ processors. How did we get $P/2$? Imagine the case where $P = N$. At each step we remove a processor so the following number of processors are active at each step: $N, N-1, N-2, \ldots, 2, 1$. We can see if we add these up and divide by N then we will be left with $(N+1)/2$ or in this case $\sim P/2$ so we believe that our speedup loss is due almost entirely to the fact that we are just not really using P processors but on average $P/2$.[9][10] So with the current implementation of the algorithm we believe we are getting performance numbers well within the range of the expectation.

3.3 Performance on Massive Data Sets

Current implementations of the neighbor joining algorithm are both limited by the $O(N^3)$ scaling or in relaxed neighbor joining the $O(N^2 \log N)$ scaling. Furthermore, from a practical standpoint a parallel implementation has not existed which allowed the entire operation to be performed in memory. We present results for distance matrices of size $N = 20,000$ and $N = 40,000$ respectively which represent some of the larger phylogenetic tree constructions which we are aware of. We ran on 50 and 100

[9] To be clear we understand that our speedup numbers are correct - that is we used a certain number of processors and that gives us the standard form $T(1)/T(p)$. What we wish to express by this exercise is simply that we can "identify" the problem and magnitude.

[10] In addition we wish to be clear about our calculation. First, we divide by N because that is the number of steps we take and we wish to compute the "average" processor usage for the entire computation. Second we get $\sum_{i=1}^{n} i = n(n+1)/2$ from Gauss. Therefore when $P = N$ we get $P(P+1)/2P = (P+1)/2$.

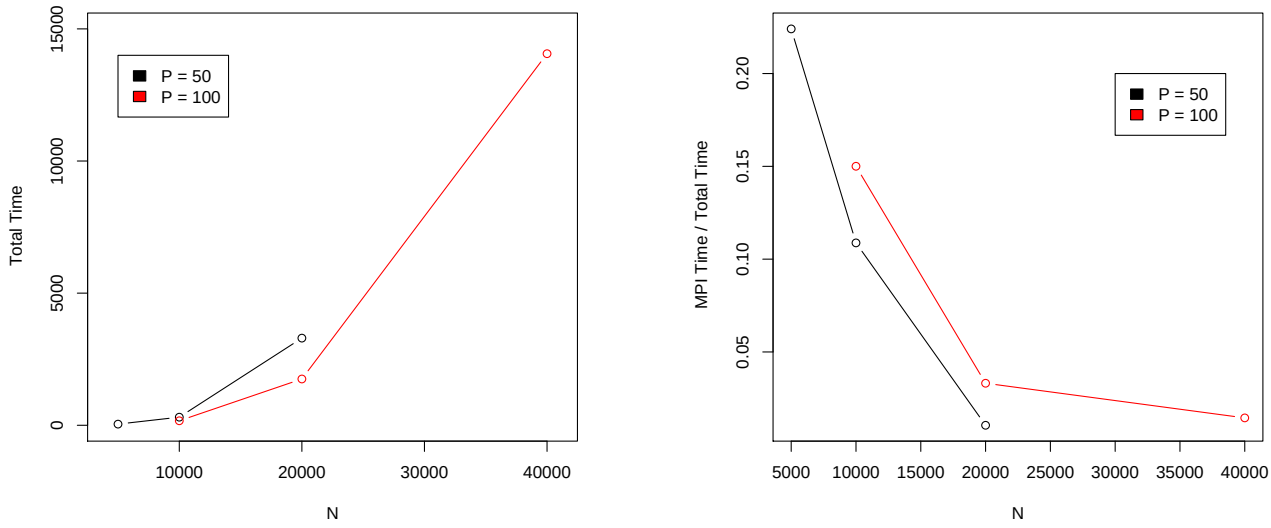


(a) Speedup

(b) Parallel efficiency

Figure 5: Speedup and Efficiency plots

processors respectively. In figure (6) we present performance results as well as relative time spent in MPI routines.



(a) Total Time

(b) MPI Time/Total Time

Figure 6: Performance of `panjo` on massive data sets.

An important observation in figure (6 (b)) is that the relative time spent in MPI is quite low when compared to the previous figures. This is good news as we don't have to worry about exchanging

less information, what we have to worry about is doing more communication so that we engage our processors more effectively.

In a work by OOta S. and Saitou N. [5] they present a parallel algorithm which utilizes a master-worker architecture. Such an implementation could do much better at constantly utilizing the processor resources, unfortunately we chose not to implement a worker client model due to the memory constraints. The careful reader however will notice that this does not preclude using a language like UPC or Titanium where implementing such an architecture is more straightforward and the memory concern is not relevant.

3.4 Measuring Load Imbalance

It is clear that there is a good deal of load imbalance in the algorithm, but just how much is there? In order to measure this we wanted to time how much time was spent in MPI barriers. Unfortunately, time constraints have forced us to abandon this effort for now.

It is not clear how much of the barrier costs can be eliminated with the current implementation of the algorithm. Clearly, we need to understand what portions of the MPI code are consuming the most time. It should be clear however that almost no time is spent in MPI time when looking at larger problems - the algorithm's run-time is dominated by the computations and therefore the only real path is to investigate solutions which do a better job at redistributing the load throughout the computation.

4 Conclusions

We implemented a distributed memory implementation of the popular neighbor-joining algorithm from phylogenetics. We implemented an algorithm that has the desirable property that for arbitrarily large distance matrices we can compute a solution given enough processors. We have released the software under the GPL and it is available at `http://stat.berkeley.edu/~bullard/panjo`. Finally, we investigated some of the load balancing issues related to our implementation.

Currently, there is a great deal of research in the area of phylogenetics. Computing the tree of life is a grand challenge problem and while the author does not expect that their program is going to be used for the endeavour anytime soon we feel that we have gained some insight on some of the challenges facing researchers in the field. Furthermore, we believe that with more work the software we develop could be useful to researchers in the area.

4.1 Future Work

There is much to be done. The first step is to convert the current codebase to use column-cyclic layout. As mentioned this does not present significant technical problems and therefore should be straightforward. The next step is to add a front end for actually computing the distance matrix. Biologists tend to know how to deal with gene sequences, but not distance matrices so we hope to include some publicly available software to make `panjo` more user-friendly. We also wish to add a input randomizer so that we "guarantee" that the distance matrix will be roughly unordered.

The real bit of interesting work is to implement the dynamic load balancing scheme. This entails determining at what point it pays to redistribute the available work to the processors. Each processor knows exactly how much work each other processor has at each step and therefore it is simply a matter of determining when and how to send the work. By when we mean "how often.", and by how

we mean "who do we send work to?" These questions present an interesting area of work and much can be learned from similar algorithms in other fields.

5 Acknowledgements

I would like to thank Kathy Yelick for all the helpful feedback throughout the project implementation. I would also like to thank Sandrine Dudoit, my advisor who encouraged me to take the class - without her support Berkeley would be a lot less fun. Finally, we'd like to thank Kasper Daniel Hansen who I work with. He pushed me to tackle the problem and helped suggested possible improvements.

References

[1] T. Z. DeSantis, P. Hugenholtz, N. Larsen, M. Rojas, E. L. Brodie, K. Keller, T. Huber, D. Dalevi, P. Hu, and G. L. Andersen. Greengenes, a chimera-checked 16s rrna gene database and workbench compatible with arb. *Appl Environ Microbiol 72:5069-72.*, 2006.

[2] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

[3] K.J. Keppler J.A. Studier. A note on the neighbor joining method of saitou and nei. *Mol. Biol. Evol.*, 5 (1988) 729-731.

[4] Bernard M. E. Moret, Jijun Tang, Li-San Wang, and Tandy Warnow. Steps toward accurate reconstructions of phylogenies from gene-order data. *J. Comput. Syst. Sci.*, 65(3):508–525, 2002.

[5] Satoshi OOta and Naruya Saitou. Njml+p a hybrid algorithm of the maximum likelihood and neighbor-joining methods using parallel computing. *Genome Informatics*, 13:434–435, 2002.

[6] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol Biol Evol*, 4(4):406–25, 1987.

[7] Luke Sheneman, Jason Evans, and James A. Foster. Clearcut: a fast implementation of relaxed neighbor joining. *Bioinformatics*, 22(22):2823–2824, 2006.

[8] Alexandros Stamatakis, Thomas Ludwig, and Harald Meier. Raxml: A parallel program for phylogenetic tree inference.