Parallel Load Balancing Schedules for Protein Phylogeny

Jeff Cullis
Faculty of Computer Science
Dalhousie University
Halifax, Canada B3H 4R2
cullis@cs.dal.ca

Jeremy Moses
Faculty of Computer Science
Dalhousie University
Halifax, Canada B3H 4R2
jmoses@cs.dal.ca

December 19, 2005

Abstract

In this paper we study the creation of phylogenetic trees using maximum-likelihood methods, and in particular the protein phylogeny program covSEARCH. We specifically examine the existing load balancing used in covSEARCH, and present a number of widely-used load-balancing schemes that could potentially be implemented instead. Finally we implement two static schedules and three dynamic schedules and present the results that are yielded under the new implementations. It is found that a simplified version of the guided self scheduler runs more quickly than the original algorithm when the number of processors is large. The static scheduler that was successfully implemented gives reasonable, though slower results, and the final two dynamic schedules suffer from poor performance.

1 Introduction

A fundamentally important problem in biology is that of inferring the evolutionary relationships (referred to as *phylogeny*) of a set of organisms (or *taxa*) of interest. Usually this entails the construction of an evolutionary tree the shows how the organisms theoretically may have evolved from mutual ancestors. In the not-so-distant past, such analysis was accomplished though careful study of the physical characteristics of the organisms themselves, but today we have a much more powerful tool for such study: the DNA code of the organism and the proteins that the DNA codes for.

In order to study these biological molecules, however, they first must be parsed into human-readable form. The first step in this process is to *sequence* the DNA or protein, that is, to generate a sequence of letters that represent its molecular makeup. For DNA, these letters represent a *nucleotide* and can be either A, C, G, or T. For proteins, the letters of a sequence represent one of 20 different *amino acids*, labelled A,C,E,F. . . and so on. These letters are known as the *bases*, or *sites*, of the sequence.

Creating Phylogenetic Trees

Figure 1 shows what a tree looks like. The distances between nodes are proportional to the evolutionary distance proposed by the model.

Sequences can be many thousands of bases in length, and are therefore rarely examined directly by humans. More commonly, sequences are taken as input by various computational tools that perform much of the initial sequence analysis. In this paper we are interested in computational tools for phylogenetic tree creation. There are three main types of tool for this task: distance-based methods, parsimony methods and likelihood methods. We now look more closely at each of these.

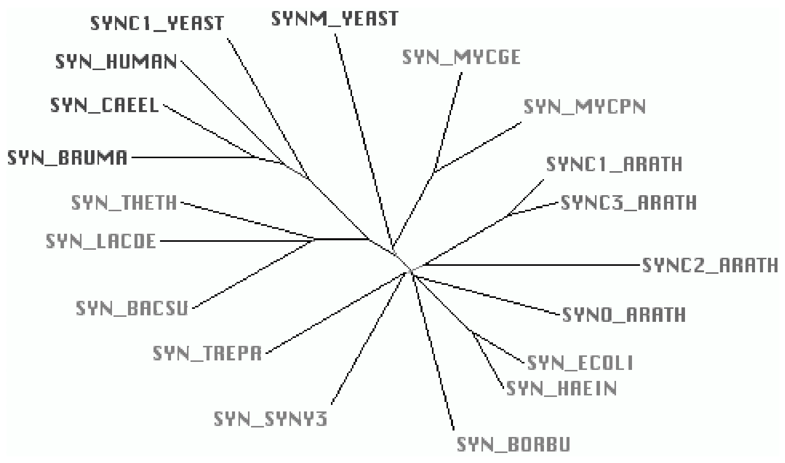

Figure 1: A sample phylogenetic tree.

Distance-based methods are among the most popular for phylogenetic tree creation (in fact, they are somewhat of a standard) due mostly to their fast runtime. The most widely used distance-based tool is the CLUSTAL family of programs. Under CLUSTAL, differences in bases along a pair-wise alignment are used as the values in a distance matrix. The pairs of sequences with the least distance between them are recursively joined to each other (with branch lengths used as an indication of this distance) and this joining in the end creates a tree. Although distance-based methods are fast, their results are not the most accurate [2].

Parsimony consists of both a scoring algorithm and a tree creation component. The parsimony score is a measure of the number of mutations that are suggested by a tree, with trees that imply an improbably high number of mutations being discarded. In parsimony tree creation, many different trees are generated and examined, and those with the best parsimony score are kept. Parsimony methods tend to give more accurate results than distance-based methods, however the problem space for parsimony is NP-complete, so runtimes can be slow.

As with parsimony, maximum likelihood (ML) methods consist of a scoring and a tree-creation component. With ML, however, a tree's score gives an indication of how likely it is that this tree gives the correct evolutionary relationships between the taxa that it models [1]. As with parsimony methods, many ML methods work by taking some input trees, permuting them in various ways, and then repeating the process on those permutations with highest ML scores.

Maximum likelihood methods tend to return the most accurate trees. Unfortunately, ML methods are thought to be NP-complete, and in any case generally run an order of magnitude slower than parsimony methods in practice [2].

Parallel Phylogeny

The tree-creation method of interest here is maximum likelihood. The running time for ML methods can be many hours on fairly typical problems. Also, much of the work that is done consists of performing the same small set of operations (scoring, tree generation, and optimization) on a large number of trees. This problem is therefore ideally suited for solution with parallel computation methods. That is, instead of running the entire algorithm on a single processor, many processors are assigned to run the same code on different trees.

One problem that arises in the implementation phase of such an algorithm is that the time to pass through one set of tree creation operations is variable. This means that simply dividing the operations evenly across the processors will not necessarily make the most efficient use of the available resources. This is the problem of *load balancing.*

Outline

In this paper we examine the parallel phylogeny program covSEARCH, and in particular its load balancing scheme. We first look at past studies of load balancing and static and dynamic scheduling in parallel algorithms, then we look at the covSEARCH algorithm in detail, and finally we implement some dynamic and static schedules to the existing covSEARCH algorithm and present the results.

2 Static and Dynamic Scheduling

Loops are an important source of parallelism in many applications, and indeed covSEARCH is no exception. The basic idea is to hand out different loop iterations evenly across the number of processors available. However, there are several characteristics of loops which can aid or impede the effectiveness of such approaches. These include the *granularity*, *uniformity*, and locality of reference of the iterations.

Granularity refers to the amount of work that is performed at each iteration. If individual iterations are very computationally expensive, this can result in load imbalance since any discrepancy in the number of iterations performed across processors will have a large effect on relative runtimes.

Uniformity refers to how long each iteration takes in comparison to others. Uniform iterations each take the same amount of time to run, and therefore an even spread of uniform tasks is generally an efficient scheduling—load balancing is easy. Semi-uniform iterations are iterations that may be dependant on other properties of the system—the loop index for example—which makes load balancing more difficult. The most difficult type are non-uniform iterations, where there are differences in runtimes between iterations that cannot be predicted before actually running the program.

Locality of reference could be an issue if certain iterations need to be performed in the same memory space. This could result in a clustering of more tasks on some processors and also communication inefficiencies.

Another property of interest is task dependence. Iterations could be highly dependent on the results of previous iterations. The ideal situation is when tasks are independent. Things become more difficult, yet still manageable, if tasks are dependent in a predictable fashion. Algorithms where iteration dependence is unpredictably dynamic are the worst-case for parallel translation. Tasks where communication is either absent, predictable, or unpredictable are similarly best, difficult, and worst-case situations, respectively.

Beyond loop characteristics, there are other factors that can affect load balancing. These include the costs of communication, synchronization, and start-up of the parallel environment. Also, variation in processor speeds, architectures (distributed vs. shared memory for instance) and processor topology can all have effects on the resulting load balancing.

We now examine the two general types of scheduling algorithms, and how the properties we've mentioned relate to them.

Static Scheduling

Static scheduling is the process of dividing available work according to a scheme decided upon at compile time. That is, we know enough about the type of iterations we are dealing with to decide on an optimal or near-optimal strategy before even running our algorithm. This is most often the case when each iteration is uniform or semi-uniform, and the dependence of iterations on one another is non-existent or predictable.

The typical static schedule is a partitioning of jobs to processors that is as evenly distributed as possible. However, if the jobs have runtimes or costs associated with them that vary, and vary predictably, then the static scheduling problem reduces to the problem of "putting objects into bins." That is, we have a set of different-sized objects (jobs) that we'd like to put into containers (processors) such that once all the objects have been put into containers, the level of the containers is nearly equal in all of them. This problem is NP-hard but linear programming and other strategies can be used to efficiently gain good results.

For certain types of problems it can be worthwhile to devise entirely new static scheduling algorithms that presume a more advanced knowledge of the structure of the tasks being run. An example of such a problem-specific static scheduler is given in [3].

Dynamic Scheduling

Dynamic scheduling solves the same problem as static scheduling, only the work division is done at runtime. While dynamic scheduling implies some performance hit, it may be the only viable option when the running time of iterations is non-uniform, and especially if the variance in task costs or runtimes is large.

Within dynamic scheduling, there are two further subdivisions of the schedule type, based on how jobs are distributed to processors. The first type is centralised load balancing, where jobs sit on a queue on a master processor and are pushed or pulled to the other processors which are designated as workers. The second type is distributed load balancing, where there is no central job queue, and processors get their jobs from a wide range of other processors. We now discuss more specific examples of each type.

Centralised Load Balancing

The simplest type of dynamic scheduler is the self-scheduling algorithm. Here each processor repeatedly takes one of the remaining iterations or jobs from a central queue, and runs it, until no further jobs remain. This works well for shared-memory systems, however on distributed-memory systems, such as those that covSEARCH runs on, the execution time would be very slow due to an excess of expensive communications in the form of requests for jobs and small job transmissions.

This immediately suggests the next scheduling algorithm, fixed-size self-scheduling, where a fixed number of iterations (making up a chunk) are taken at once by each processor. Here there is a trade-off between maximizing chunk size to decrease communications, and the fact that with larger chunks there is a greater chance of poor load balancing, for instance if all processors finish at the same time, except for one which has just started a final large chunk of work to do.

Under guided self-scheduling (GSS) [4] we again take chunks of tasks, but each chunk is of differing (generally decreasing) size. The idea is to have processors working on the largest chunks first, then progressing towards smaller and smaller chunks as the entire program nears completion. This works to decrease overhead costs at first by using large chunk sizes, while ensuring that at the end discrepancies in processor utilisation are not large. The usual scheme for the chunk sizes is for the i^{th} chunk to to contain K_i iterations where $K_i = \lceil \frac{R_i}{p} \rceil$.

Trapezoid scheduling [4] also uses successively smaller blocks, only in this case K_i depends not only on the number of remaining tasks, but also on the task cost variance. If the variance is high then a smaller number of tasks are incorporated into chunks, whereas a smaller variance will result in larger chunks being used. A variation on this schedule, weighted factoring, also takes into account the differences in computational power that occur in heterogeneous environments.

Distributed Load Balancing

With distributed load balancing there is no central job queue, but instead processors send to and receive from their neighbours. This can have advantages in distributed systems, where the cost to repeatedly communicate with a master processor may become a bottleneck, or in shared memory systems where synchronization is costly. This approach may also be beneficial in situations where inter-task dependencies are worked out on the fly.

Typical distributed schemes include round-robin and randomised work stealing. Round-robin schemes are where each processor attempts to get work from its neighbours, incrementing (modulo the number of processors) from one target to the next as work is completed. In random schemes, processors simply attempt to get work from randomly selected processors. It turns out that the random approach is often the best one [5].

3 The covSEARCH Algorithm

The overall covSEARCH problem is divided up into rounds, each round beginning with an initial tree (or group of trees) and ending with a tree that is deemed to be more biologically likely than the tree the round started with. The algorithm terminates when a tree of a certain likelihood threshold has been reached, or when there has not been sufficient progress in the last few rounds.

In its attempt to find a better tree, at the beginning of each round, covSEARCH generates a lot of random mutations on the current pool of trees. Each tree is checked to see if it has been investigated before, and it is then evaluated for approximate likelihood. If the tree then has a likelihood above some constant threshold, it is retained for further computation, otherwise it is discarded. Those trees that are kept have a branch-length optimisation run on them which is a more accurate likelihood measure (which also takes longer to compute), and these trees are kept in a pool from which the new most likely tree is selected at the end of the round.

On a sequential machine, this algorithm is pretty straight forward. The computations can be done in any order, with only a few minor effects to watch out for. We will look at the two extreme cases to illustrate some potential sequential optimisations. In the first case, when generating a new mutation on one of the current pool of trees, we would generate only one mutated tree. This tree would then be evaluated for likelihood, and if necessary, would have the branch-length optimisation run on it. If we do this, we are using the absolute minimum amount of memory. The second extreme strategy has us generating all necessary mutations at one time, and placing all of these mutations in a pool. In the next step, every tree would be evaluated for likelihood, and in the final step, all remaining trees would have the branch optimisation run on them. This method uses significantly more memory, however we may find minor speedup due to locality of reference effects. Large blocks of trees can be fetched from memory at once, and new code is loaded only infrequently. A balance between these two extremes will become more important in the parallel case.

The Current Static Schedule

CovSEARCH already has a parallel version implemented, but it suffers from (possible) sub-optimal speedup over the sequential variant. We will describe the current implementation, and some possible problems with it. When a round starts, the head node distributes the current best pool of trees to every node. From this pool, every node generates every mutated tree. Since the mutation algorithm is deterministic, we are guaranteed to generate the same group of mutations at every node. Each node does not generate the entire mutated pool right away, however, but rather some small non-constant portion of the mutated pool is generated (typical is around 300 trees of 600) at a time so as to alleviate memory concerns (we would assume). The of the nodes divide these trees evenly (where possible). From there, each node runs the likelihood and branch optimisation calculations on each of their trees. Each node reports back their own best tree to the head node for the selection of the trees for the next round.

There are three things potentially wrong with this load balancing schedule. First, we generate only a subset of the number of trees we will work on in one round, thus creating "sub-rounds". Each sub-round is as difficult to balance as each actual round with every node getting an unequal number of trees to compute every sub-round. This causes the potential for much more idle time at each node. Secondly, the size of each tree is very small, and unless we are working on very RAM limited machines (typically not the case in compute clusters), we need not limit ourselves to such small numbers of trees. In fact, even if the trees are 1MB each (and they are nowhere near that size), at 500 trees per round, we have plenty of memory to work with. Again, this point will come into play more apparently in load-balancing schedules other than this one. Finally, and this point comes more from observation, even if we give every node an equal amount of trees to compute, they may end up with a very unequal amount of computation to complete. This stems from the fact that the maximum likelihood calculation takes about two orders of magnitude less time than the branch length optimisation and that we have no idea how many trees from each node will end up having the branch length optimisation run upon them. We might expect, given this imbalance of run-times, that if a node is given very few trees that are "promoted" to branch-length optimisation, that it would complete much sooner than a node that is given many promoted trees. Indeed we find this is the case. Figure 2 shows a typical round of computation. Note that the amount of time that a node takes to complete a round is very strongly related to the number of branch-length optimisations it has to run.

Given this imbalance among the nodes, we devised the following metric to measure the amount of wasted time in a given round. For round times $t_1 \dots t_n$, and $M = max(t_1 \dots t_n)$, wasted time $= \sum_{i=1}^{n} \frac{M-t_i}{M}$. We found that the amount of wasted time was typically any from 5% to 10% of the time in a round. This would seem to suggest that we cannot do drastically better in terms of run-time with this type of set-up (where every node generates every tree, and operates on a subset of them).

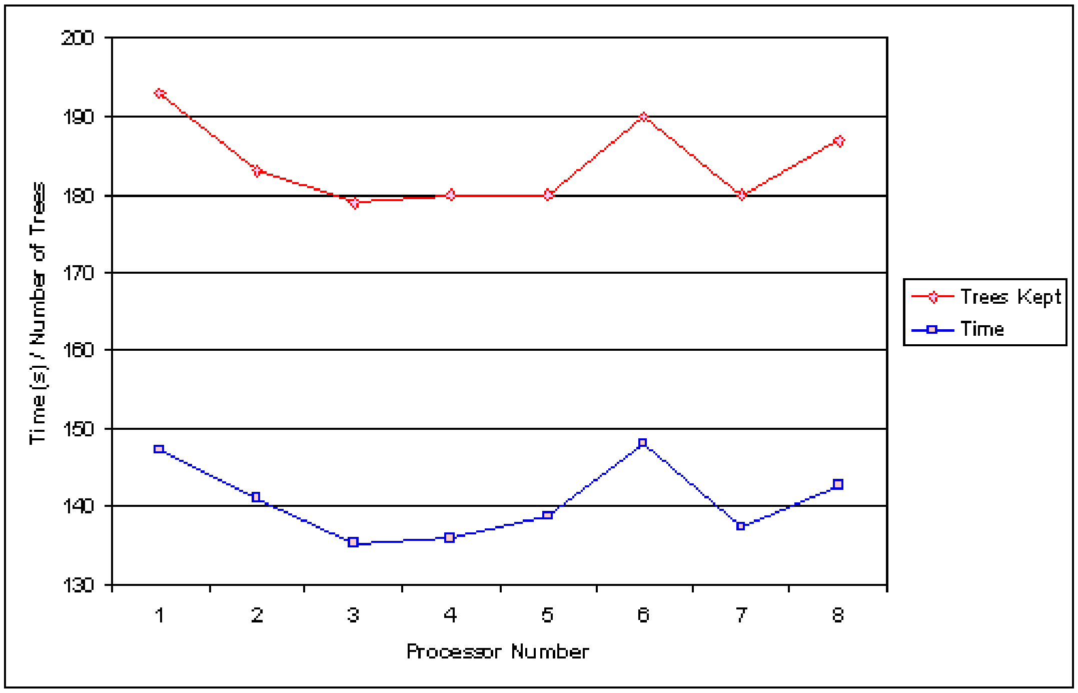

Figure 2: shows the amount of time taken by all processors in a typical round. Note that the number of trees that are kept and thus have the branch-length optimisation run on them is very strongly correlated to the amount of time that the round takes.

Two New Static Schedules

Based on the observations above, we came up with two new static schedules for load balancing.

Zero Communication

To understand the first static schedule, we will quickly review the current algorithm. There are two computation phases, the ML calculation, and the branch-length optimisation (BLO). In the current scheme, every node runs ML on $\frac{1}{P}$ of the trees, and some indeterminate number of these will go on to have the BLO run on them. Wasted time comes from having an imbalance in the number of BLO trees computed, and the ML takes about one fiftieth of the time of the BLO per tree. So our reasoning was to have every node calculate the ML on *every* tree, and then we would know exactly how many trees each node should compute the BLO on. So each node takes on P times as many short ML calculations in exchanged for much less overall wasted time per round. In fact, every processor (except perhaps the one) will have exactly the same number of computations to do. In practice, this worked well to balance the calculations, but it did not achieve any speedup.

The reason that this takes longer than the original schedule, is because the processors

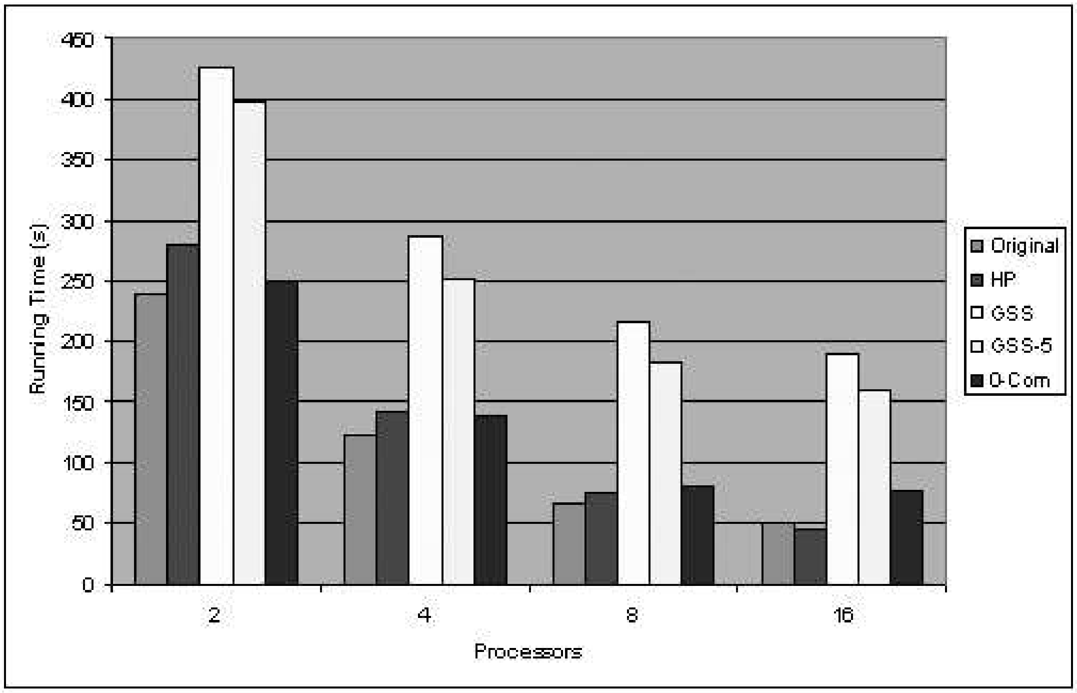

Figure 3: A comparison of the running times for the static and dynamic schedules implemented.

are fairly well balanced to begin with. There is only a max of 10–12% wasted time. In a round that takes 200 seconds for 8 processors, there is a difference of maybe 16 trees between the node with the fewest BLO calculations and the node with the most. This is perhaps 20 seconds (the max potential savings). However, it takes about 120 seconds to run all of the ML calculations, but only 16 seconds to calculate $\frac{1}{8}^{th}$ of them. The upside to this schedule is that wasted time dropped from 10–12% to around 1–2%.

Speedup of the Zero Communication Schedule is compared to the other schedules in figure 4.

Parallel ML

Bolstered by our success with the reduced wasted time, we figured we were on the right track. Given that the ML calculations took a lot longer than we thought they would we had to come up with a new schedule. The obvious idea would be to parallelise the ML calculations, and then do a rebalancing of trees, and then do the BLO calculations. It is here that we finally notice why we need to have access to all of the trees at one time: if we do not have all trees in memory, we would have to do several rebalancing rounds. Two rounds of communication are required to rebalance the trees after the ML calculations are done, and much effort was given to making these communication steps as small as possible. The first communication is an MPI_alltoall which simply broadcasts the number of trees that each node has that will require the BLO. Each node takes these numbers and calculates how many trees the *should* have. An algorithm is run on every node which tells every node who should be borrowing which trees from who. This way, every node knows what every other node is doing, and then every node knows which trees to send to which node.

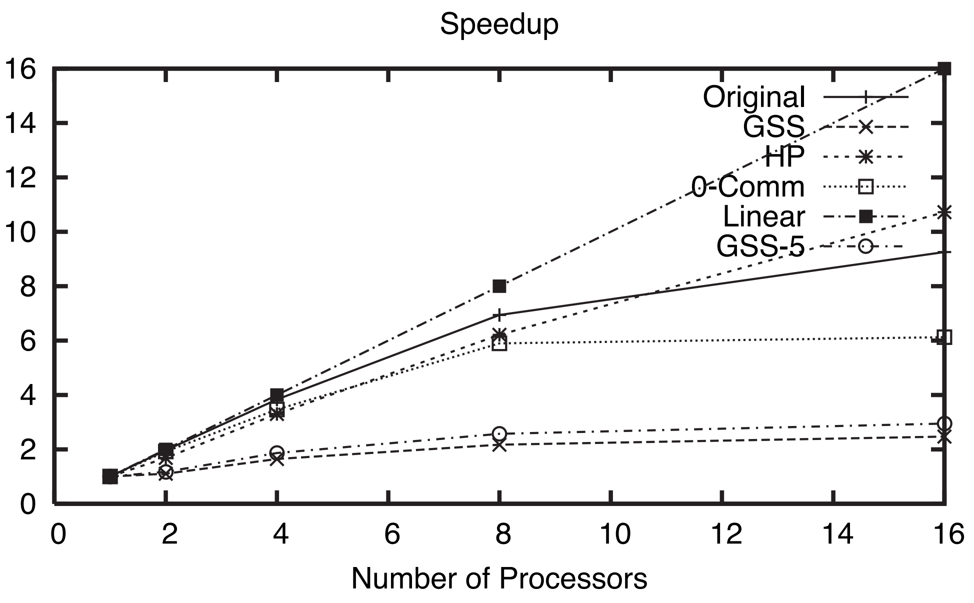

Figure 4: The speedup graph shows that none of our schemes do too well, but of the lot, HP seems to be the best for 16 processors.

For the next communication, we must notice that every node has generated every SPR tree, but they do not know at this point which of these trees have been kept by their neighbours, or which trees they will be borrowing, or giving away. Since all trees are present at every node, each node that is giving tress to another node must only communicate which trees they are giving away. This is done with a small number of MPI_Sends.

After these two communications, every node should have done an equal number of ML calculations, and an equal number of BLO communications. Unfortunately while we have the code implemented with at least 10-20 hours of debugging put into it, we were unable to get this schedule to work on Pepito - a machine which has proved to be very flakey indeed. We estimate, however, that of the 20 seconds we possibly could save, that this scheme would end up saving us 15–18 seconds depending on how large the communication overhead ends up being.

4 Implementation and Results

The Dynamic Schedule

Our first implementation idea in terms of dynamic scheduling was to implement the guided self scheduler discussed in 2. However, it was found that the existing algorithm did not simply generate all the tree permutations at once, but instead generated them in sets. This made computation and storage of the percentage of trees to leave unprocessed (for later computation under GSS) more difficult. Therefore a simplified approach (Half Processors or HP) was devised. Here, instead of creating chunks of decreasing size for later processing, we instead allowed the half of the processors that finished earliest to run equal-sized chunks of the remaining work. The idea being that perhaps the first half of the processors were finishing significantly earlier than the second half. Further, if some processors finished much earlier than the rest, they could run a second (or third etc.) chunk.

Hours of testing and gathering of results were undertaken using this scheduler, and it was found that the running times were very similar to those of the original algorithm. The reason for this soon became clear—the program was in fact not behaving the way we had at first thought, and our code was barely being executed at all. Briefly, the difficulty came from a "maximum tree set size" parameter that was used in generating the SPR trees (i.e. the work that was to be divided up). It was thought that this parameter indicated the number of trees that were returned, however it was found that this value was used as a type of seed, and influenced, but did not define, the number of trees returned.

Once the problem was determined, it did have an upside—the implementation of the GSS algorithm was in fact made more feasible. The GSS implementation was performed and the results are given in 3 . There was by this point not time to run the program on any larger data sets for all processor combinations, however the results yielded on a smaller data set are given. The major effect is unfortunately a significant slowdown, by roughly a factor of three.

This slowdown is most likely due to the fact that the naïve GSS algorithm will continue to divide up the work into chunks, up to the point near termination where each chunk may only contain one or two tasks to perform. For instance, if 600 trees are examined in a round (a low number) on 8 processors, the 20% or 120 of those that are turned into smaller chunks are in fact turned into roughly 25 chunks. This means that the total number of chunks used in GSS is 33, compared to just 8 in the original algorithm.

Increasing the number of chunks by a factor of four may be acceptable on a shared memory system, however on a distributed system such as pepito, the amount of overhead introduced became prohibitive. This overhead is not due only to the MPI communication overhead, but also the overhead of beginning the computational phase, which includes creation of new tree pool and test structures at each phase.

Another new schedule (GSS-5) was implemented in an attempt to alleviate some of the problems found in the original GSS implementation. This was simply a modification where a minimum value was set for the chunk size, and here a value of 5 was used as the minimum. The results yielded from this implementation are also shown in 3. A significant improvement over GSS is found here.

Finally, HP, the original modified scheduler, was re-worked to run properly here. The results for this implementation are also given in 3. This schedule in fact runs more quickly on 16 processors than the original algorithm, and by a significant margin. However, it runs more slowly than the original on fewer processors. The proposed reason for this is due to the "master's worker" problem, which is discussed below.

In the original implementation, the master sends off trees to all the worker processors. However, in order to be useful while the other processors are working, it runs its own instance of the worker class before trying to receive the results from the other processors. The master's worker is unfortunately run in a blocking manner, which means that if any of the other processors finish their tree computations before the master's worker, they cannot be given further work by a dynamic scheduler until the master's worker returns. It also means that the master's worker cannot be easily given more than one chunk of tasks to work on within a round without slowing down the handing out of further tasks to other processors.

As a result of the above problem, the HP algorithm most likely runs faster on 16 processors because the time lost due to the master's worker is spread over a larger number of processors. Recuperating this lost time would likely give further performance increases on smaller numbers of processors.

"More Master"

One other thing that we felt was worth mentioning here is the original scheme used to divide up the trees among the nodes. The original implementers for some reason unknown to us decided to give the root node a few more trees to compute. Possible reasons include that the root node was faster than the other nodes on the first machine this was implemented on (improbable), or that the original author thought that the other nodes should be done their work before the root node so as to avoid some possible race condition (this would just be sloppy coding). Firstly, the idea was not a good one because it would further imbalance the number of trees each node has. Second, and more importantly, the method used to give the root node more trees was actually *incorrect*. It turned out that this scheme (called the "more master" scheme), was causing a few trees to not be investigated. This is not only a potential memory leak, but it also could cause the program's output to be incorrect. To correct this, we simply removed the more master scheme.

5 Conclusion

Over the course of this paper we have learned quite a bit about the load balancing problems associated with a two-step calculation such as in covSEARCH. We found that the load-balancing was not bad to begin with, but that there is the possibility to waste maybe 10–15% less time every computation round. We have a few things which we would like to try in future work. Firstly, we would like to fork the master process into a separate thread in the head node so as to facilitate dynamic scheduling better (and to alleviate the problem of the other nodes having to wait for the master). We would also like to finish implementing the second static scheduling algorithm, as it seemed to be the best of both worlds: it parallelised both the ML and branch-length opts. Finally, we would like to see if there is any way to prevent every node from generating every tree – this is the largest remaining piece of the code that is not parallelised.

References

[1] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[2] Michael Lawrence Glenn Hickey. Randomized parallel phylogenetic tree search csci 6702: Parallel computing course project, 2004.

[3] G. Paller and C. Wollinski. Springplay : A new class of compile-time scheduling algorithm for heterogenous target architectures.

[4] Oscar Plata and Francisco F. Rivera. Combining static and dynamic scheduling on distributed-memory multiprocessors. In *ICS '94: Proceedings of the 8th international conference on Supercomputing*, pages 186–195, New York, NY, USA, 1994. ACM Press.

[5] Katherine Yelick. Cs 267: Applications of parallel computers load balancing. `http://www.cs.berkeley.edu/\~yelick/cs267/lectures/21/lect21-load.pdf`, 2004.