Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2010) 1065–1073

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Parallel computation of phylogenetic consensus trees☆

Andre J. Aberer[a], Nicholas D. Pattengale[b], Alexandros Stamatakis[a,*]

[a]The Exelixis Lab, Department of Computer Science, Technische Universität München, Germany
[b]Department of Computer Science, University of New Mexico, USA

Abstract

The field of bioinformatics is witnessing a rapid and overwhelming accumulation of molecular sequence data, predominantly driven by novel wet-lab sequencing techniques. This trend poses scalability challenges for tool developers. In the field of phylogenetic inference (reconstruction of evolutionary trees from molecular sequence data), scalability is becoming an increasingly important issue for operations other than the tree reconstruction itself. In this paper we focus on a post-analysis task in reconstructing very large trees, specifically the step of building (extended) majority rules consensus trees from a collection of equally plausible trees or a collection of bootstrap replicate trees. To this end, we present sequential optimizations that establish our implementation as the current fastest exact implementation in phylogenetics, and our novel parallelized routines are the first of their kind. Our sequential optimizations achieve a performance improvement of factor 50 compared to the previous version of our code and we achieve a maximum speedup of 5.5 on a 8-core Nehalem node for building consensi on trees comprising up to 55,000 organisms. The methods developed here are integrated into the widely used open-source tool RAxML for phylogenetic tree reconstruction.

Keywords: RAxML, Majority Rule Tree, Multi-core Architecture, Phylogenetics

1. Introduction

Novel wet-lab sequencing techniques such as pyrosequencing [1] are driving a rapid accumulation of molecular sequence data that in turn poses new challenges for the development of scalable Bioinformatics tools. This data trend along with the recent paradigm shift to multi-core processors is necessitating scaling [re]considerations and parallelization of existing methods. Such is certainly the case for phylogenetic tree reconstruction, a field concerned with inferring evolutionary relationships between organisms based on their molecular data. A *phylogenetic tree* is an unrooted binary tree where currently living organisms for which molecular data is available are located at the tips; the inner (ancestral) nodes of phylogenetic trees represent extinct common ancestors.

While the actual tree reconstruction problem under the widely used Maximum Likelihood criterion [2] is NP-hard [3] and thus, computationally challenging, significant progress has been achieved with programs such as for instance RAxML [4] or GARLI [5]. In fact, RAxML [4] now allows for reconstruction of phylogenies that contain

☆Part of this work is funded via the Emmy-Noether program by the German Science Foundation (DFG).
*Corresponding Author
Email address: stamatak@cs.tum.edu (Alexandros Stamatakis)

1877-0509 © 2010 Published by Elsevier Ltd.
doi:10.1016/j.procs.2010.04.118

more than 10,000 organisms [6] (organisms are also called taxa in this context). One of the prohibitive factors when scaling to datasets of 10,000 or more taxa is in post-analysis. Post-analysis can refer to multiple things. For instance, it is often the case that reconstruction finds multiple equally plausible trees. This situation can be resolved by building a *consensus tree*, a summary tree that attempts to capture the information agreed upon by the set of plausible trees. The consensus tree that is most often assembled is the so-called *majority rules extended (MRE)* tree (also called greedy consensus trees [7], see Section 3), which has been popularized due to longstanding implementations in PHYLIP [8] and PAUP [9].

Another post-analysis task for which computing MRE trees is important is that of *bootstrapping* [10]. Bootstrapping is a technique for assessing the extent to which the underlying data (e.g., molecular sequences) support a reconstructed tree. Bootstrapping has not traditionally involved MRE consensus, however recently proposed *bootstopping criteria* [11] call for repeatedly computing MRE trees. Bootstopping criteria provide on-the-fly (i.e., runtime) guidance for determining when enough bootstrap replicates have been reconstructed.

While the code for phylogenetic inference in RAxML has been parallelized with Pthreads and MPI [12], the bootstopping tests have not (prior to the present paper) been parallelized and thus represent a performance bottleneck for reconstructing very large trees in terms of number of taxa. A concrete endeavor that could benefit from the techniques in this paper is the iPlant tree of life project (`www.iplantcollaborative.org`), which aims to reconstruct trees containing 100,000-500,000 taxa

In this paper we present the first parallelization of the MRE algorithm and provide a detailed performance study on current multi-core architectures of up to 32 cores on datasets containing as many as 55,000 taxa. We also detail sequential optimizations of the MRE algorithm that render RAxML the current fastest exact implementation for computing such trees.

2. Related Work

To the best of our knowledge, this paper represents the first parallelization of the MR and MRE algorithms in general, and for multi-core architectures in particular. With respect to sequential implementations Sul and Williams recently presented a program called HashCS [13] for computing strict consensus and MR consensus trees (but not extended MR trees) and conducted a comparative performance study with consensus tree algorithms implemented in other phylogeny tools such as PAUP* [14] or MrBayes. They show that, the HashCS implementation is the fastest currently available implementation for computing strict and MR consensus trees. Note however, that the algorithm of HashCS is approximate. That is, it is not guaranteed to yield the correct result (see p. 105 in [13]) because not all collisions in the hash table are resolved.

3. Sequential Implementation & Optimization

The implementation of consensus tree methods in RAxML was initially motivated by our work on bootstrap convergence criteria [11]. Specifically, we decided that it was worth the incremental effort to enhance RAxML such that it can compute majority rules (MR) and majority rules extended (MRE) consensus trees. Throughout this paper we mainly focus on the construction of MRE consensus trees, which are computationally significantly more challenging and harder to parallelize.

In this Section we briefly describe the main algorithmic steps of the original implementation as well as the optimizations of the sequential code.

The pertinent structural property of phylogenetic trees, upon which consensus tree algorithms operate, are *bipartitions* of trees. A bipartition of an unrooted binary tree is obtained by removing a branch of the tree, thereby splitting the leaves of the tree into two disjoint sets. An unrooted binary tree contains exactly $2n - 3$ bipartitions (one for each of its edges). An array containing all $2n - 3$ bipartitions of a tree fully defines its topology (in fact, only $n - 3$ of the bipartitions are necessary as n bipartitions are *trivial* in the sense that one side of the bipartition consists of a single leaf). One way of storing a bipartition in memory is as a bit-vector where indices correspond to taxa, and two indices have differing value when the two corresponding taxa exist on opposite sides of the edge inducing the bipartition. Since this representation can lead to two representations of the same bipartition (due to complementarity), we make the representation canonical by deciding upon a distinguished taxon that always takes the value 0.

A *consensus method* takes a set of trees as input, and produces a single summary tree as output. The *strict consensus tree* for a collection of trees only contains the bipartitions that are present in all trees (and as such is often not binary). The *majority rules (MR) consensus tree* consists of the bipartitions that occur in more than half of the input trees. As shown in [15], this set of bipartitions is guaranteed to be pairwise *compatible* (defined in Section 3, but in essence means that two bipartitions can structurally coexist in the same tree), and thus define a (like strict consensus, a not necessarily binary) tree [16]. The *majority rules extended (MRE) consensus tree* consists of all bipartitions in the MR tree, along with bipartitions added greedily (according to the number of trees in which they appear) under the constraint that each added bipartition is compatible with all those bipartitions already added to the consensus tree, until either the consensus tree is fully resolved (binary) or the sorted list of bipartitions is exhausted.

To describe the implementation, optimization, and parallelization of the MRE algorithm, we break it down into four phases:

1. **Tree Parsing:** Loading the collection of trees into memory and parsing them.
2. **Extraction and Addition of Bipartitions:** Extracting bipartitions from each parsed tree and inserting them into a hash table.
3. **Selection of Candidate Bipartitions:** Selecting candidate bipartitions for the final MRE tree and storing them in an array according to their frequency of occurrence and compatibility with the bipartitions that have already been added to the array.
4. **Reconstruction of the MRE tree:** Using the array of bipartitions to build the MRE tree and print it to file.

The fraction of execution time spent in the different phases largely depends on the tree size. For trees with 2,500 organisms it spends an approximately equal proportion of time in each phase, while for larger trees with 30,-55,000 organisms, run times are dominated by the selection of candidate bipartitions (phase 3.) that entails the compatibility check and requires more than 95% of total run time.

Tree Parsing. The tree parsing procedure for the Newick tree format (see `http://evolution.genetics.washington.edu/phylip/newicktree.html`) is straightforward, but has benefitted from some focused modifications. In order to prepare for parallelization and to accelerate the parsing phase, we replaced the parsing routine that directly reads and parses the tree file, by a function that first loads the trees into main memory and then parses them as strings to avoid unnecessary I/O overhead. In addition, an optimization of the taxon name lookup procedure yielded significant speedups. In order to check for consistency in the taxon names, i.e., if a taxon name in the tree is contained in the set of taxon names, and also to consistently enumerate taxa, the parser needs to look up every taxon name that is encountered in a tree. The original implementation used a linear $O(n)$ time taxon name lookup which was replaced by a simple constant time lookup using a hash table. These optimizations yielded significant speedups of more than an order of magnitude, especially for trees with more than 1,000 taxa.

Extraction and Addition of Bipartitions. Once an input tree has been parsed, we extract the $n-3$ non-trivial bipartitions and store them in a hash table as suggested in [17]. We essentially use the bit vector representation of a bipartition as a hash key. As described in [17], bipartitions of a tree can be extracted in $O(n)$ time. The pertinent RAxML structure is a hash table entry:

```
struct ent {
  unsigned int *bitVector;
  unsigned int *treeVector;
  ...
  struct ent *next;
};
```

where `bitVector` is a representation of the bipartition such that all leaves on one side of the bipartition (the side that contains the first taxon) have value 0 (respectively 1), and `treeVector` is a (bit vector) where an index i has value 1 when this bipartition occurs in tree i, and is 0 otherwise. Note that, `treeVector` could essentially be replaced by a simple integer counter for computing MR and MRE. However, other functions that operate on bipartitions of trees in RAxML require to also store the tree which generated a bipartition; thus, we decided to keep the code as simple and generic as possible. In order to obtain the frequency of occurrence of a specific bipartition in the tree set, one needs to count the bits that are set in `treeVector`. This can be done by using one of the fast counting routines discussed at `http://gurmeetsingh.wordpress.com/2008/08/05/fast-bit-counting-routines/`. Based on extensive computational experiments with bit counting functions, we find that functions that use lookup tables perform best on modern CPUs.

Selection of Candidate Bipartitions. Once the bipartitions of all trees are stored in the hash table, and the respective frequency of occurrence of every bipartition is computed (using a fast bit count on `treeVector`), it is trivial to select the bipartitions that form the MR consensus tree. One simply needs to iterate through all bipartitions in the hash table and retain every bipartition (or rather a pointer to every bipartition) in an array, that occurs in more than half of the input trees, i.e., whose frequency of occurrence is > 0.5. At most $n - 3$ bipartition will be stored in this array in the case that the MR tree is a fully resolved binary tree.

Computing the MRE tree is only marginally more complicated, but significantly more compute intensive. One starts by constructing the MR tree, i.e., by adding those bipartitions that occur in more than 50% of the trees to the array of bipartitions of the consensus tree. Next, all bipartitions not occurring in the MR tree are sorted (in descending order) by their frequency of occurrence. Finally, the sorted list is scanned, and a bipartition is added to the bipartition array if it is compatible with *all* other bipartitions that already form part of the consensus tree. This operation is of time complexity $O(n^2)$, where n is the number of taxa, and hence dominates the run time of this step. However, we have found that the order in which the array of bipartitions that already form part of the consensus tree is traversed for checking compatibility has a notable effect on execution times. It turns out that, the number of pairwise compatibility checks between bipartitions is greatly reduced if the array is traversed from the end, i.e., starting at the most recently added entry. This reversal of the compatibility check order yielded a run time improvement for the bipartition selection phase of approximately 90% with respect to the original implementation on a tree with 2,554 taxa. The speedup is obtained because bipartitions that are located at the end of the array have a lower frequency of occurrence than bipartitions that are located at the start of the array. As such, there is a higher probability that the bipartition under consideration occurs together in a tree with bipartitions earlier in the array, and thus are compatible. It follows that the probability that the candidate bipartition to be added is incompatible with the bipartitions located at the end is higher and we therefore, on average, need to conduct less compatibility checks per candidate bipartition. Thus, the biggest gains are achieved by very diverse input tree sets that do not give rise to a fully resolved binary MRE tree. Bipartitions with low occurrence frequencies can often be rejected after the first compatibility check against the accepted bipartition with lowest frequency.

All of the steps just described (for computing MRE) are trivial, with perhaps the only exception being compatibility checking. For compatibility checking, we initially followed a very straightforward approach. We utilized the well known property that for two bipartitions $A|B$, $C|D$ to be compatible, it must be the case that at least one of the intersections $A \cap C$, $A \cap D$, $C \cap B$, $B \cap D$ is empty [18]. If we have stored A and C in the canonical form described above, the computation of the intersection of $B \cap D$ can be omitted because B and D will both contain the distinguished taxon and their intersection will therefore never be empty.

This formulation of compatibility gives rise to a straightforward implementation using bit-vector based bipartitions. Since the function for compatibility checking dominates the execution times of the $O(n^2)$ time pairwise compatibility check, we optimized it by testing different implementation options for the compatibility function. The optimization of this function yielded an additional run time improvement of approximately 50% on trees with more than 37,000 taxa and of 70% on trees with 2,554 taxa for the compatibility check.

Reconstruction of the MRE tree. We have found that when large trees are involved, a nontrivial amount of time can be spent transforming the consensus tree in bit-vector bipartition form back into a Newick representation. Our basic approach for this transformation, without parallelization, is resemblant of the approach taken in HashCS [13]. However, we describe that process here to yield Section 4 understandable. First, we sort the bit vector bipartitions according to number of bits set in ascending order. Then, for each bipartition, we search the sorted list for the first occurrence of a bipartition that has a superset of its bits set with respect to the bipartition under examination. In this manner we build for each bipartition, a list of the bipartitions that are its most strict subset bit vectors. It is then straightforward to traverse these lists in depth-first order and output the tree. Fortunately, the process of searching the list for the first occurrence of a superset is a parallelizable task, and is described in Section 4.

The sorting and the ascertained compatibility of all of the consensus bipartitions allow for a rapid test on whether bipartition A is a superset of bipartition B. Due to the sorting, A is either a proper superset of B or the set shares an empty intersection with B (direct consequence of the compatibility check). Thus, for the superset test we only need to check, if any element of A is also contained in B. This optimization yields a run time improvement of 70% for the tree reconstruction phase on a tree set with more than 38,000 taxa.

4. Parallelization

In the following we describe the parallelization of the four phases of the algorithm using Pthreads and the busy-wait mechanisms implemented in RAxML [12] to synchronize threads (which otherwise are used for for orchestrating parallel likelihood computations).

Tree Parsing. The parallelization of the tree parsing routine is relatively straightforward. Initially, the master thread will read in the tree file and store it in memory. During a first pass over all trees in memory it computes an array of pointers to this memory area that indexes the first symbol of every tree topology. Thereafter, the tree start pointers are distributed to all threads (including the master thread) for parsing by using a lock on the number of trees that remain to be parsed. Thereby, we allow for better load-balance, since the time for parsing a tree, extracting its bipartitions, and adding them to the hash table varies depending on the actual tree topology. Each thread uses a separate tree data structure in its private memory area to store trees.

Extraction and Addition of Bipartitions. Once a tree has been parsed by a thread, the bipartitions are extracted and added to the global hash table whose consistency is maintained via a simple lock for every hash table entry. It turns out that this naïve locking mechanism is sufficient, particularly because the tree parsing times and bipartition extraction times are significantly larger than the actual addition of a bipartition to the hash table. Moreover, the efficiency of this parallel operation on the hash table is mainly limited by memory bandwidth and not by the simple locking mechanism we use, as is underlined by experiments for read-only hash table accesses that exhibit identical scalability.

Selection of Candidate Bipartitions. When constructing the MR tree, bipartitions are simply selected by frequency of occurrence. While this operation is not worth parallelizing, the extension of an MR into an MRE tree offers huge potential. As already mentioned, adding bipartitions with support ≤ 0.5 (which requires compatibility checking) dominates run times (e.g., over 95% of total runtime for trees with more than 37,000 taxa).

The difficulty in parallelizing this task is a direct data dependency: in order to accept a bipartition, a compatibility check must be performed against all bipartitions that have already been accepted so far. An approach with one thread per bipartition that is to be checked would require the threads to wait for all other threads working on bipartitions with higher frequencies.

Instead, we decided in favor of a master-worker scheme. We divide the whole of candidate bipartitions into batches (we address batch size below). The huge advantage of this approach is, that it functions without any locking (except for a minor case – the worker threads trying to acquire a new candidate for testing). For each batch the worker threads check compatibility of the candidate bipartitions against all consensus bipartitions accepted in previous batches. This check can be performed with the desired order reversal. When a worker accepts its candidate, it marks the candidate as being ready for a check against all bipartitions from the current batch that were accepted as consensus bipartitions. This check is then performed by the otherwise busily waiting master thread. This way the resolution of the dependency is delegated to only one thread, while the order reversal can be maintained. Note that the master does not add the accepted candidates to the whole of all consensus bipartitions until the current batch is fully processed. See Figure 1 for a graphical representation of the algorithm.

The major disadvantage to this approach is the performance with exactly two threads: here the only worker thread is too slow in providing pre-approved candidates for the master. However, this is not critical because of the rapid increase in core count. In general, load imbalance between master and worker threads can largely be avoided with a careful choice of the batch size. If the batch size is chosen too small, the worker threads will run out of work quickly and have to wait for the master thread to post-process the candidates. If on the other side, the batch size is too big, the master thread will spend too much time waiting for new proposals. A balancing between the expected amount of checks to be performed by each worker $\mathbb{E}[W]$ and that of the master $\mathbb{E}[M]$ is desirable. A good estimate for optimal batch sizes should also take into account, that bipartitions with lower frequency among the trees have a higher probability of getting rejected and thus hardly consume computational time. In order to reflect this circumstance, we measure the acceptance rate r (fraction of accepted candidate bipartitions) of the previous batch and use it as an

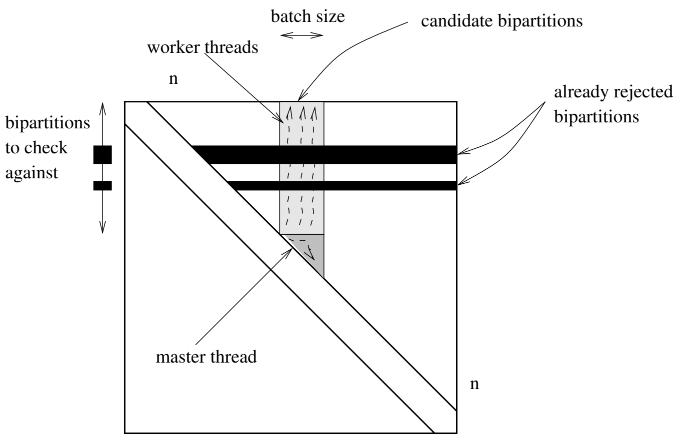


Figure 1: Parallel master-worker scheme for the compatibility check.

approximation. Together with the amount of workers AW and the number of bipartitions accepted so far b, the new batch size x can be derived as follows. Ideally we would like to achieve $\mathbb{E}[W] = \mathbb{E}[M]$, which can be rewritten as $\frac{x \cdot b \cdot r}{AW} = \frac{(r \cdot x)^2}{2}$. To obtain x one simply needs to compute: $x := \frac{2 \cdot b}{r \cdot AW}$.

In order to assess the quality of our estimate x, we multiplied it with a constant c, where $0.2 \leq c \leq 8$. However, any $c \neq 1$ lead to a performance decrease. Via experimentation we also found that, the batch size b should not be set to less than 5 candidates per worker.

Furthermore, when the acceptance rate r decreases, competition for candidates among the workers can influence performance. When processing small trees with many threads, we found that performance can be improved by over 30%, if the worker threads are assigned more candidates (inversely proportional to the acceptance rate) once the respective worker acquires the lock.

Reconstruction of the MRE tree. In the conversion of the bipartition representation to the Newick tree format, the search for direct supersets is quadratic in the number of consensus bipartitions and thus the most time-intensive part of the final tree reconstruction. Nonetheless, sorting and the fact that the search loop can be aborted when the direct superset has been found, lead to good average performance. Additionally, we parallelized the superset search in a similar manner as the insertion of bipartitions into the hash table by using naïve locking (see Sections 3 and 4).

Each thread computes the superset of one of the bipartitions at a time. As explained, we maintain lists with strict subsets of the bipartitions, where the respective subset is to be inserted. In the parallel case, this insertion must be locked. Once again we use the straightforward approach: each of the lists is locked by one lock.

5. Experimental Results

Experimental Setup. For our computational experiments we used a 32-core SUN x4600 with 64GB of main memory and a 8-core Intel Nehalem with 36 GB of main memory. The code was compiled using `gcc v4.3.2` and `gcc v4.4.1` respectively with optimization flags `-DNDEBUG -O2 -fomit-frame-pointer -funroll-loops`. As test datasets we used a collection of 10,112 bootstrap trees with 2,554 taxa, of 672 bootstrap trees with 37,831 taxa, and of 211 bootstrap trees with 55,593 taxa. The trees in these tree sets all originated from real-world biological datasets and were computed using RAxML. The source code and test datasets for the sequential and parallel versions of the code is available for download at `http://wwwkramer.in.tum.de/exelixis/parallelRF.tar.bz2`.

taxa	2,554	37,831	55,593
HashCS	49 secs	84 secs	75 secs
RAxML	25 secs	145 secs	167 secs

Table 1: Sequential execution times for MR tree building with HashCS and RAxML

batch size	10	100,	1,000	10,000	100,000	adaptive
37,831 taxa	8,732s	4,279s	4,499s	3,405s	5,904s	2,771s
55,593 taxa	18,215s	9,300s	9,675s	7,289s	13,006s	6,808s

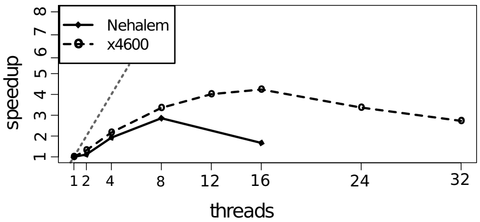
Table 2: MRE execution times in seconds on 32 cores with fixed and adaptive batch sizes.

Sequential Performance. In Table 1 we provide the sequential execution times in seconds of the exact MR implementation in RAxML and those of the approximate MR implementation in HashCS on the Nehalem. On the two larger datasets, HashCS is approximately 2 times faster than RAxML for a variety of reasons: HashCS, unlike RAxML, is stand-alone software for computing MR and strict consensus trees. Therefore, certain optimizations are more straightforward to implement in HashCS than in RAxML where software engineering considerations also need to be taken into account, such as for example the decision not to use a simple integer counter for the bipartition frequency. Moreover, the RAxML implementation has been more thoroughly optimized for computing extended MR trees. Also, some computational tricks of HashCS can not be applied to extended MR consensus calculations. In addition, we have observed that HashCS yields erroneous results on 64 bit architectures and that it can not automatically count the number of trees in the two larger datasets. We therefore compiled a 32 bit executable of HashCS on the Nehalem which yields correct results. One should keep in mind that the execution times for computing MR are very short compared to the actual tree inference times (for datasets of this size several thousands of CPU hours). Note that, the computation of MRE trees requires two orders of magnitude more time than MR because of the compatibility check.

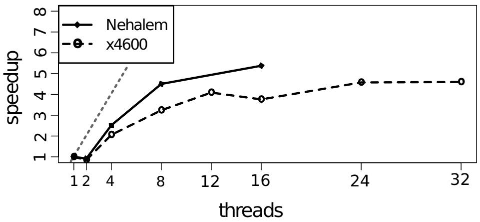
Parallel Performance. In Table 2 we provide total execution times in seconds as a function of batch size for MRE computations on the 37,831, and 55,593 taxon trees using 32 cores. The data clearly show that the adaptive criterion we have developed significantly outperforms MRE reconstructions using fixed batch sizes. Moreover, an unfavorable batch size choice can yield a two-fold slowdown.

In Figures 2(a) through 2(c) we provide speedup plots for computing MRE consensi on the 2,554, 37,831, and 55,593 taxon trees respectively for the Nehalem and Sun x4600 systems. On the Nehalem which has 8 physical cores we also exploited the hyper-threading capability which yielded a run time improvement of approximately 10%. Because of the sequential optimizations described in Section 3 the average sequential execution time for MRE on the 2,554 taxon dataset on the Nehalem was only 30 seconds. The parallel efficiency, especially on the Nehalem is acceptable given the inherent sequential dependencies in the compatibility check routine that dominates execution times.

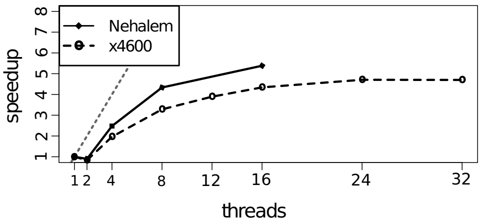
In Figure 2(d) we provide the per-phase speedups for the 55,593 taxon dataset. Note that, we have further split up phase 2 of the algorithm into bipartition extraction and hash table insertion times. The operations of phase 2 scale badly and even yield a slowdown because they are limited by memory bandwidth and irregular memory accesses. We also measured the absolute times (data not shown) for parallel read-only hash-table lookups as used for drawing support values on a given tree which behave similarly in terms of scalability. Therefore, we conclude that the bad performance for phase 2 is mainly because of memory bandwidth and latency issues, rather than due to the naïve locking mechanism we use. In these experiments the Nehalem is faster overall than the SUN x4600 when hyper-threading is disabled. The speedups on the Nehalem are more favorable because of the higher memory bandwidth. Since phase 2 only accounts for 1% of total execution time on the two large datasets the impact on scalability is negligible. As can be observed by comparing Figures 2(c) and 2(d) the overall speedup plot is congruent to the speedup of the compatibility check (phase 3).



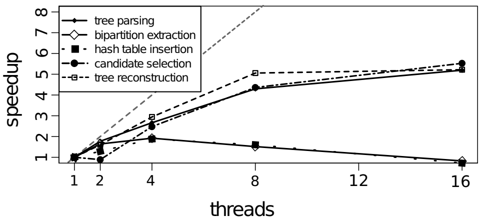
(a) Speedup plot for 2,554 taxon trees



(b) Speedup plot for 37,831 taxon trees



(c) Speedup plot for 55,593 taxon trees



(d) Per-phase speedup plot for 55,593 taxon trees

6. Conclusion and Future Work

We have presented the first parallelization and performance study for the computation of MRE trees on current multi-core architectures. In addition, we have described a series of sequential optimizations for computing MRE. We mainly focus on the parallelization of the compatibility check phase of the program which requires more than 95% of total execution time on large tree collections with more than 37,000 taxa. We proposed a suitable mechanism for partially resolving the inherent sequential dependencies of the compatibility check which lead to moderate speedups of 4-5.5 on a Sun x4600 with 32 cores and a Intel Nehalem with 8 cores and hyper-threading. Given the hard–to–resolve sequential dependencies, the speedups of this initial parallelization of MRE are rather promising and the Pthreads implementation will help to accelerate the post-analysis of large tree collections in day to day biological practice as well as the computation of our Bootstopping test. Future work will cover a refinement of our parallelization strategy to further improve scalability.

Finally, following our usual practice, we will progressively integrate the methods developed here into the standard release of RAxML such that a production-level parallel code for computing consensi on multi-cores will become available to the large user community.

References

[1] M. Ronaghi, Pyrosequencing Sheds Light on DNA Sequencing, Genome Research 11 (1) (2001) 3–11.
[2] J. Felsenstein, Evolutionary trees from DNA sequences: a maximum likelihood approach, J. Mol. Evol. 17 (1981) 368–376.
[3] B. Chor, T. Tuller, Maximum likelihood of evolutionary trees: hardness and approximation, Bioinformatics 21 (1) (2005) 97–106.
[4] A. Stamatakis, RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models, Bioinformatics 22 (21) (2006) 2688–2690. doi:10.1093/bioinformatics/btl446.
[5] D. Zwickl, Genetic Algorithm Approaches for the Phylogenetic Analysis of Large Biological Sequence Datasets under the Maximum Likelihood Criterion, Ph.D. thesis, University of Texas at Austin (April 2006).
[6] S. Smith, M. Donoghue, Rates of Molecular Evolution Are Linked to Life History in Flowering Plants, Science 322 (5898) (2008) 86–89.
[7] D. Bryant, A classification of consensus methods for phylogenetics.
[8] J. Felsenstein, Phylip - phylogeny inference package (version 3.2), Cladistics 5 (1989) 164–166.
[9] D. L. Swofford, Paup*. phylogenetic analysis using parsimony (*and other methods). version 4. (2003).
[10] J. Felsenstein, Confidence Limits on Phylogenies: An Approach Using the Bootstrap, Evolution 39 (4) (1985) 783–791.
[11] N. D. Pattengale, M. Alipour, O. R. Bininda-Emonds, B. M. Moret, A. Stamatakis, How many bootstrap replicates are necessary?, in: RECOMB 2'09: Proceedings of the 13th Annual International Conference on Research in Computational Molecular Biology, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 184–200.

[12] A. Stamatakis, M. Ott, Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study., in: M. Chetty, A. Ngom, S. Ahmad (Eds.), PRIB, Vol. 5265 of Lecture Notes in Computer Science, Springer, 2008, pp. 424–435.

[13] S.-J. Sul, T. L. Williams, An experimental analysis of consensus tree algorithms for large-scale tree collections, in: Proceedings of 5th Intl. Symposium on Bioinformatics Research and Applications (ISBRA'09), Springer LNBI 5542, 2009, pp. 100–111.

[14] D. L. Swofford, PAUP*: Phylogenetic analysis using parsimony (* and other methods), version 4.0b10, Sinauer Associates, 2002.

[15] T. Margush, F. McMorris, Consensus *n*-trees, Bulletin of Mathematical Biology 43 (1981) 239–244.

[16] P. Buneman, The recovery of trees from measures of dissimilarity, In D.G. Kendall and P. Tautu, editors, Mathematics the the Archeological and Historical Sciences (1971) 387–395.

[17] N. D. Pattengale, E. J. Gottlieb, B. M. E. Moret, Efficiently computing the Robinson-Foulds metric, J. Comput. Biol. 14 (6) (2007) 724–735, special issue on best papers from RECOMB'06.

[18] D. Gusfield, Efficient algorithms for inferring evolutionary trees, Networks 21 (1) (1991) 19–28.