2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-1199-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPSW59300.2023.00035

Parallel Inference of Phylogenetic Stands with Gentrius

Anastasis Togkousidis[1,2], Olga Chernomor[3], Alexandros Stamatakis[4,1,2]
[1]Computational Molecular Evolution Group, Heidelberg Institute for Theoretical Studies
[2]Institute for Theoretical Informatics, Karlsruhe Institute of Technology
[3]Center for Integrative Bioinformatics Vienna (CIBIV), Max Perutz Laboratories,
University of Vienna and Medical University of Vienna, Vienna Bio Center (VBC),Vienna, Austria
[4]Biodiversity Computing Group, Institute of Computer Science, Foundation for Research and Technology - Hellas
Email: anastasis.togkousidis@h-its.org

***Abstract*—Multi-locus datasets are frequently used to infer phylogenies instead of using single locus. Missing data constitute a common challenge in such datasets as they can lead to stands, that is, sets of trees that are compatible with the incomplete per-locus trees. Under many common criteria the trees from one stand have identical score. Hence, identifying stands and determining their sizes is of crucial importance for a robust phylogenetic analysis. Recently, Chernomor *et al.* published Gentrius, a branch-and-bound algorithm that enumerates all stand trees given a set of unrooted incomplete locus trees. Despite its efficiency, the pattern and proportion of missing data in multi-locus datasets can still induce extremely long execution times.**

Here, we introduce the parallel version of the Gentrius algorithm. Our parallelization deploys a thread-pooling mechanism that maintains threads that finish early in busy-wait mode, such that they can contribute to solving long-running tasks. Thereby, we substantially reduce load imbalance and attain high parallel efficiency. Our performance assessment up to 16 cores yields linear parallel speedups on both, simulated, and empirical data. The parallel version of Gentrius is available as open source code under GNU GPL at https://github.com/togkousa/iqtree2/tree/terragen. All data we used for our analyses, are available for download at https://cme.h-its.org/exelixis/material/gentrius-parallel.tar.gz.

***Keywords*-phylogenetic stands, phylogenetic terraces, multi-locus data, parallel computing, shared memory parallelism**

I. Introduction

Standard phylogenetic methods infer trees from multiple sequence alignments (MSAs), for instance using the Maximum Likelihood (ML) criterion [1]. It is common practice to construct MSAs from a single gene/locus, comprising the sequences of distinct species. One may assume that this genome region constitutes a valid proxy for the evolutionary history of the respective species. However, different genes might have evolved under different models and parameters [2] and exhibit a distinct evolutionary history than the underlying species tree. Thus, phylogenetic inferences conducted on distinct genes might yield incongruent tree topologies.

Two alternative approaches to reconstructing phylogenies are the so-called supermatrix and supertree methods. In the first case, per-gene MSAs are assembled/concatenated into a large supermatrix that is typically divided into disjoint partitions, often representing the distinct genes/loci. The phylogenetic tree is then inferred from this supermatrix, where each gene is typically assumed to evolve under its own model of evolution while all genes share the same underlying tree topology. Second, in supertree methods separate gene trees are initially inferred for each gene and, then, are reconciled into a species tree by maximizing appropriate criteria, such as the reconciliation likelihood [3] or the quartet-consistency score [4].

Multi-locus datasets often exhibit patches of missing data. That is, a species may have no data present in a specific locus, either due to sampling issues, or because the target locus is simply absent in species genome. The presence and absence of data can be represented by a *presence-absence species per locus matrix* (PAM). The pattern and amount of missing data in PAM can lead to the so-called *stands* - collections of all species trees compatible with a set of corresponding induced per locus subtrees (Section II-A). Under many scoring criteria typically used in phylogenetics (e.g. likelihood, parsimony in supermatrix, or quartet-consistency score in supertree approaches), all trees from the stand have identical scores. In such cases, stand is called a *terrace*. The concept of terraces was first used implicitly in [5] and was explicitly described in [6], [7] for supermatrix and in [8], [9] for supertree approaches.

Missing data are rather common in multi-locus empirical datasets. In the RAxML Grove v0.7 database [10], we counted $7,295$ empirical, partitioned multi-gene datasets, $4,959$ (68%) of which had a non-zero proportion of missing data and $1,390$ (19%) a missing data proportion exceeding 30%. Identifying stands/terraces in such datasets constitutes an important step, both when searching tree space, and when post-analyzing the tree inference results. The presence of terraces indicates that a single inferred tree only represents one of many equally good solutions. The first systematic attempt for detecting stands was the implementation of the SUPERB algorithm [11] in a python tool called *terraphy* [12]. Thereafter, Biczok *et al.* [13] developed two efficient and independent `C++` implementations of the same algorithm. The main limitation is that the original SUPERB algorithm is defined on rooted trees. In practice, almost all 'classic' phylogenetic inference tools and methods return unrooted trees. In order to consistently root these unrooted trees, tools implementing the SUPERB algorithm

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00035

require the input dataset to contain at least one so-called *comprehensive taxon*, that is, a taxon that has data for *all* partitions of the MSA.

Recently, Chernomor *et al.* developed the Gentrius algorithm [14], a novel approach for enumerating trees on a stand from a set of unrooted, incomplete trees (i.e., trees not containing all taxa). Gentrius is a branch-and-bound type algorithm (Section II-B). It improves upon two aspects over the previous methods; it directly operates on unrooted trees and does not rely on the presence of a comprehensive taxon to conduct the enumeration. Gentrius is implemented in IQ-TREE 2 [15] in C++ and is executed sequentially. However, the computational complexity of building stands for unrooted trees is intractable [16]. Moreover, the number of trees on the stand can be also exponential [17]. Both these aspects, depending on the input tree and PAM, contribute to excessive running times to obtain even a lower bound on stand size.

To this end, we developed a parallel shared memory version of Gentrius that deploys a thread-pooling approach. That is, active threads continuously create and push new tasks into a queue, while inactive threads - those that have finished their jobs - remain in waiting mode until a new task becomes available in the queue. Essentially, each thread enumerates a fraction of the overall trees on stand, and the entire stand size is determined by summing up those individual calculations. Our experimental results (Section IV) yield linear parallel speedups up to 16 cores on both simulated and empirical data.

In this paper we present the parallelization of the Gentrius algorithm. The sequential algorithm and its practical applications are discussed in the original manuscript. The paper is organized as follows: In Section II, we present the underlying idea of the Gentrius algorithm, while in Section III we introduce our parallelization approach and its implementation via thread-pooling. We conduct a thorough parallel performance evaluation of Gentrius in Section IV. We conclude in Section V and indicate directions of future work.

II. Overview of Gentrius

A. Background

Data availability in multi-locus datasets is typically summarized via a binary PAM. Its elements are 1's and 0's, indicating the presence or absence of data for each species and locus. Let X be the full set of species/taxon labels in the PAM. Consider any binary tree T on X and let $Y_i \subseteq X$ be the subset of available taxa for locus i. Hence, $X = \bigcup_{i=1}^{m} Y_i$, where m is the number of loci. Further, let T_i be a tree on Y_i. We say that T displays T_i if $T|Y_i = T_i$, where the vertical bar denotes the subtree induced by restricting/pruning T to the taxa contained in Y_i. Two trees are *compatible* if there exists a tree that displays both of them. In other words, T'_1 and T'_2 are compatible if there is a single tree T'_0 from which both T'_1, T'_2 can be derived by a sequence of edge contractions. Such a tree exists *if and only if* two trees have identical induced subtrees for their common taxa. A stand is defined as the set of all trees on X that are compatible with all trees T_i, each one defined on Y_i for locus i.

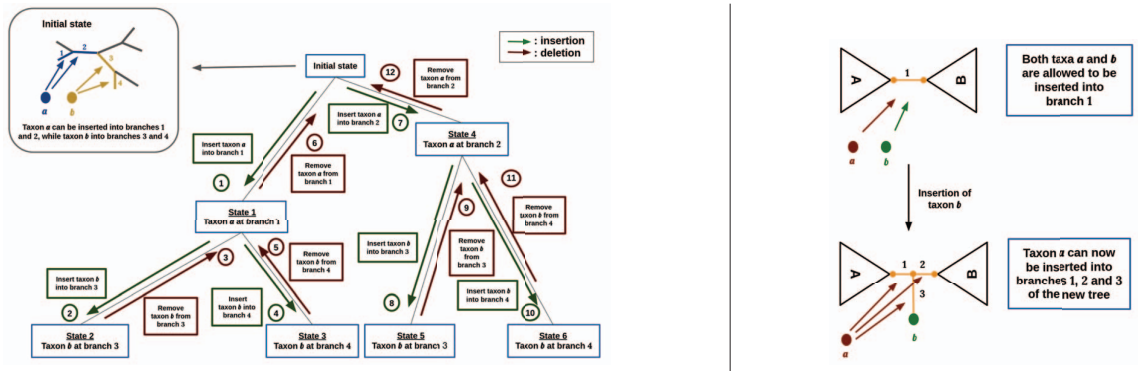
Gentrius is a deterministic, branch-and-bound algorithm that generates all trees on a stand, given a set of incomplete, unrooted subtrees. One of these subtrees serves as the initial tree, which we will call the *agile tree*. The algorithm uses the principle of *stepwise taxon insertion*, meaning that taxa which are not present in the initial agile tree are inserted sequentially into it, until it comprises all taxa contained in the set of subtrees. The incomplete subtrees are also called *constraint trees*, since they restrict taxon insertion positions to comply with the compatibility condition. There might be multiple edges where the non-present taxon can be inserted into the agile tree. For the enumeration of all stand trees, Gentrius tests all possible edges by successively inserting, removing, and re-inserting taxa into the agile tree. The standard output of the Gentrius algorithm is the number of trees on the stand and their topologies in the Newick tree format.

A second input option of Gentrius is to provide a complete species tree inferred with any phylogenetic method, together with a PAM. Gentrius will then extract the set of *induced subtrees* from the species tree. Each induced subtree is obtained by removing the taxa with zero entries for the corresponding locus in the PAM. This set of induced trees now constitutes the initial set of constraint trees described above and serves as a starting point for the algorithm.

A new taxon can be inserted only into *admissible branches*. A branch is called admissible, if after the taxon insertion the agile tree extended thereby is *pairwise compatible* with each constraint tree. We can effectively identify the set of admissible branches for each taxon using the so-called double-edge mapping approach which is described in detail in the Supplementary material of [14]. For each agile tree/constraint tree pair, Gentrius constructs their common subtree and maps the branches of the agile tree and constraint tree onto the branches of the common subtree. The mappings follow precise rules to comply with the compatibility criteria. Both mappings are surjective, meaning that for every branch in the common subtree, there is at least one branch in the agile tree (and the constraint tree as well) that is being mapped onto it. Multiple branches, though, from the agile/constraint tree can be mapped onto a signle branch in the common subtree. For taxon insertion, Gentrius maps its incident branch from the constraint tree onto a branch $\hat{b}$ in the common subtree. Next, the set of admissible branches is determined via the inverse mapping of $\hat{b}$ onto the agile tree. In case a taxon occurs in many constraint trees, the intersection of the corresponding individual branch sets is taken. After each taxon insertion or removal, these mappings are updated. We define a *state* of the algorithm to be the current agile tree, together with the set of constraint trees, the common subtrees, and the corresponding mappings at a given point in time. An *intermediate state* is a state in which the current agile tree is still incomplete (i.e. some of the taxa have not been inserted yet).

B. Algorithm

The core of the Gentrius algorithm is the recursive pseudocode function in Algorithm 1. The functions, in the order



(a) (b)

Fig. 1: Schematic overview of the Gentrius workflow. **(a)** A simple example where only two taxa, a, and b are missing from the initial tree. The admissible insertion branches for taxon a are $\{1, 2\}$ and $\{3, 4\}$ for taxon b, respectively. There is no overlap between the two insertion branch sets. Gentrius initially inserts taxon a and then proceeds to taxon b. Arrows $1 - 12$ illustrate the sequence of alternating states, generated by the recursive function of Algorithm 1. **(b)** A simple example showing that we cannot know a priori the set of admissible branches for all taxa. Here, taxa a and b can be both inserted into branch 1 of the initial tree. The two sets of admissible branches overlap. Following the insertion of taxon b, taxon a can now be inserted into branches $\{1, 2, 3\}$ of the new tree, without violating the compatibility condition.

they appear, are:

- `generateStandTrees()`: The main recursive function. Its arguments are the state of the algorithm, the list of taxa to be inserted and an integer `step` which indicates the recursion depth.
- `getNextTaxon()`: Takes as input the state of the algorithm, the list of taxa to be inserted and the integer `step`. It returns the next taxon to be inserted into the agile tree.
- `getAllowedBranches()`: Takes as input the state of the algorithm and the taxon to be inserted. Returns the set of admissible branches based on the double-edge mappings.
- `extendTaxon()`: Takes as input the state of the algorithm, the taxon to be inserted and the admissible branch. It inserts the taxon into the agile tree, updates the double-edge mappings and returns the new state.
- `removeTaxon()`: Takes as input the state of the algorithm and the taxon to be deleted. It removes the taxon from the agile tree, updates the double-edge mappings, and returns the new (previous) state.

Algorithm 1 generateStandTrees (state, list_of_taxa, step)

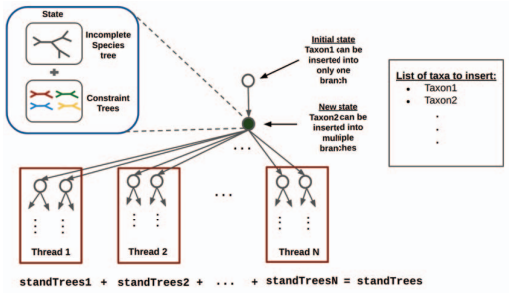
```
1: taxon ← getNextTaxon (state, list_of_taxa, step)
2: branches ← getAllowedBrances (state, taxon)
3: for branch in branches do
4:     new_state ← extendTaxon (state, taxon, branch)
5:     generateStandTrees (new_state, list_taxa, step + 1)
6:     if no more taxa then
7:         standTrees ← standTrees + 1
8:     end if
9:     state ← removeTaxon (new_state, taxon)
10: end for
```

An example of the Gentrius workflow is provided in Figure 1a. Under this simple scenario, Gentrius inserts taxa a and b, which are initially missing from the incomplete agile tree, into all possible combinations of admissible branches, starting from taxon a and continuing with taxon b. Here, the algorithm counts four stand trees in total.

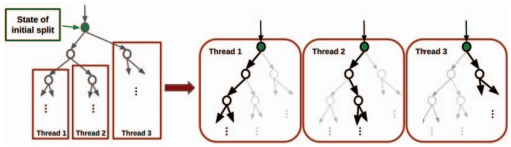
Based on Figure 1b, we can argue that it is impossible to know the set of admissible branches for all taxa, a priori. Chernomor *et al.* argue that the choice of the initial agile tree and the order of taxon insertion substantially affect the efficiency of Gentrius [14]. The efficiency is determined by the number of all generated intermediate states and the number of *dead ends* encountered. A dead end is defined as an intermediate state where at least one of the remaining taxa cannot be inserted into the agile tree without violating the compatibility condition. In this case, the algorithm removes the last inserted taxon from the agile tree and re-inserts it into a distinct admissible branch, if such a branch exists. Thus, an "unlucky" choice of the initial tree and an unfavorable taxon insertion order can induce visiting numerous irrelevant intermediate states leading to dead ends. While the algorithm will nonetheless correctly generate all stand trees, the execution time will be noticeably longer.

To accelerate the enumeration of stand trees, Gentrius uses two heuristics. The first one is to chose a "good" initial tree. Gentrius selects the constraint tree, which shares the largest number of taxa with all remaining constraint trees. The second heuristic is called *dynamic taxon insertion*. At each step, Gentrius selects the taxon with the smallest number of admissible branches, to be inserted next. To illustrate the validity of both heuristics, we tested them on dataset `emp-data-42370`. The number of stand trees on this dataset is $2,448,225$. When both heuristics were applied, Gentrius visited $547,786$ intermediate states and 0 dead ends. This translates into 14 seconds of serial execution time[1]. By deactivating the initial tree selection

[1]on an 11th Gen Intel(R) Core(TM) i7-1165G7 processor @ 2.8 GHz



(a)



(b)

Fig. 2: (a) First division of the total process into multiple parallel threads. The green state is the state of the initial split. Parallel execution is initiated when the algorithm encounters the first taxon to be inserted into multiple branches. **(b)** An alternative splitting scheme where the number of admissible branches in the initial split state (green state) exceeds the number of threads.

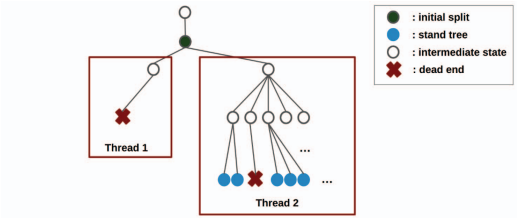


Fig. 3: An example of an unbalanced work distribution among threads. Thread 1 chooses the path that leads to the left subtree that only consists of a single dead end. The right subtree comprising a substantially larger amount of work is assigned to Thread 2. This imbalance occurs as we can not predict the structure of the workflow tree a priori.

heuristic and starting from a random constraint tree, Gentrius visited $6,829,128$ intermediate states and 0 dead ends in 50 seconds (3.5x slowdown). By deactivating the *dynamic taxon insertion* heuristic and randomly shuffling the taxon order, Gentrius visited $30,124,986$ intermediate states and $1,547,640$ dead ends in 174 seconds (12x slowdown).

In the worst-case scenario, the number of trees on a stand can be exponentially many [17]. Moreover, generating even a single tree from the stand is also computationally intractable [16]. To prevent excessive runtimes Gentrius employs three stopping rules:

1) when the algorithm counted more than N stand trees
2) when more than M intermediate states have been visited
3) when the execution requires more than T hours

The default values for these stopping rules are: $N := 10^6$ stand trees, $M := 10^7$ intermediate states and $T := 168$ hours. The algorithm terminates when either, all stand trees have been enumerated, or when one of the stopping conditions is satisfied.

III. Parallel Gentrius

In this section we introduce a parallelization scheme for Gentrius which is based on a thread pooling approach. Thereby, we attain "good" load balance and a substantial parallel speedup for Gentrius.

A. Underlying idea

Figure 1a illustrates the tree-like structure of the Gentrius workflow. Intuitively, we can split the entire process into two threads by assigning the left subtree of the workflow - rooted at State 1 - to one thread and the right subtree to a second thread. The final set of stand trees will then be the union of the disjoint trees counted by each thread independently.

In principle, we parallelize by generalizing this idea to N_t threads. All threads begin operating concurrently and each thread independently parses the entire input set of constraint trees. This redundant input parsing and storage by all threads is necessary as each thread requires some degree of flexibility to insert and remove taxa *on its own* agile tree. Since Gentrius is a deterministic algorithm and the input is fixed, the first stages of execution are identical across all threads. They all select the same initial agile tree and begin by adding the exact same taxa to it. For representative datasets (i.e. with relatively small amount of missing data), the first couple of taxa can be inserted into a single branch only, due to the dynamic taxon insertion heuristic that is used to accelerate Gentrius. However, at some point, Gentrius reaches a state where the next taxon can be inserted into multiple branches. We define this to be the *state of initial split*, where the first division of the algorithm into independent subprocesses (concurrent threads) takes place. Our parallel algorithm assigns the set of admissible branches for a given taxon as uniformly as possible among threads. For example, if there are 5 admissible branches and four threads are being used, Gentrius will assign two branches to one thread and one branch to each one of the remaining threads, respectively. Thereafter, each thread proceeds by adding the remaining taxa independently. In other words, after this initial parallel split of the workload each thread operates in a distinct branch of the branch-and-bound algorithm. In cases where the very first taxon already needs to be inserted into multiple branches, the initial split takes place at the root of the workflow (branch-and-bound) tree. The idea is outlined in Figure 2a.

A substantially distinct splitting scheme example is provided in Figure 2b, where the state of the initial split has two admissible branches for inserting the next taxon. If parallel Gentrius is executed with three threads, the thread number exceeds the cardinality of this set. Gentrius assigns the right subtree of the workflow to one thread. The two remaining threads both operate on the left subtree and their task separation occurs at a later stage when they encounter the first taxon with two or more admissible branches. We define a *path* that connects two states on the branch-and-bound graph to be

a sequence of taxon insertions/removals that, when applied to the agile tree of a thread being in one state, it becomes topologically equal to the agile tree of a second thread in a different state. In Figure 1a, for example, if two threads are in States 2 and 4 respectively, for the first thread to reach the state of the second, taxa b and a shall first be removed from its agile tree (from branches 3 and 1 respectively) and, then, taxon a must be re-inserted into branch 2. Since double-edge mappings are automatically updated after each taxon insertion/removal, synchronization between the states of the two threads is achieved. This idea is crucial for the second part of our parallelization scheme.

As outlined in Section II-A through Figure 1b, the set of allowed branches for taxon insertion is highly dependent on the insertion position of the previous taxon. The main problem that arises from this a priori uncertainty (i.e., simply not knowing the structure of the workflow graph in advance) is, that the initial division of the load can be unbalanced. For instance, it is likely that we assign a part of the workflow tree with a high number of stand trees, intermediate states, and dead ends to one thread, and a separate path with substantially less work to another. Figure 3 illustrates such an extreme example.

To alleviate this load imbalance we devise a thread-pooling parallelization scheme. A thread-pool maintains active threads that have completed their task early, instead of terminating them. The concept we deploy is also known as work-stealing. In analogy to the initial split case, the division of the workload among threads requires them to be in the exact same state, and is accomplished by assigning a distinct subset of the next taxon's admissible branches to each thread. Our work-stealing mechanism relies on the same idea. The first step such that early finished threads can attain the state of the working threads is to remove all taxa from their agile tree up to the state of the initial split (state I_0), which is the last common and consistent state. Thereafter, these threads switch into busy-wait mode. A *task* consists of the following components:

1) a path from state I_0 to a desired intermediate state, including the set of taxa to be added, their exact insertion order, and positions
2) the remaining taxa, the very next taxon to be inserted and a precomputed subset of admissible branches

Having passed state I_0, working threads are allowed to create and push tasks into a task queue. Each time a working thread moves between intermediate states by adding or removing taxa, it keeps track of the path that connects I_0 with its current state I_c. For creating a task, it calculates the set of admissible branches for the next taxon and divides it in half. The first half is submitted into the task queue, together with the path that connects states I_0 and I_c. The working thread proceeds by executing the second half. An inactive busy-waiting thread in the pool detects the new task and switches into working mode. It sequentially inserts all taxa into the predefined branches on its own agile tree, to reach state I_c. The next taxon and the corresponding subset

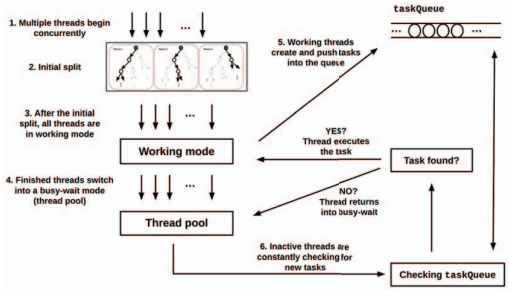


Fig. 4: Schematic overview of the parallelization scheme.

of admissible branches are already specified in the dequeued task. The new thread skips line 2 in Algorithm 1 (Section II-B) and directly proceeds with the recursive function. Figure 4 provides a summary of the entire parallelization strategy.

The following implementation details need to be considered. Our parallelization scheme requires each thread to exclusively work on its own copy of the agile tree. When threads assume independent tasks, they concurrently insert the exact same taxa into different branches of the agile tree, to generate all possible combinations of complete trees. This would be infeasible, on an agile tree that is shared by multiple threads, since concurrent insertion of the same taxon into multiple branches would cause an assertion to fail in the code due to inconsistency. Also, regarding the additional time required for one thread to reach the state of another, Gentrius processes hundreds of thousands of states per second[2]. For datasets comprising up to a few thousand taxa, this translates into a few milliseconds.

Finally, we have implemented some restrictions to the generation and submission of tasks to avoid task overload. There is an upper limit in the number of tasks that the task-queue can hold concurrently. We define this limit to be N_t+1, if $N_t < 8$, and $N_t/2$ otherwise (where N_t is the number of threads). The second restriction is that threads whose current state is deep down in the workflow tree, that is, threads that have less than three remaining taxa to insert, are not allowed to submit a new task, because counting the stand trees beneath their current state is fast and hence will not benefit from additional parallelization (work stealing). We selected the values for both thresholds based on the results of preliminary experiments.

B. Implementation

Our implementation combines the OpenMP API with the `C++` Thread library, for two reasons. First, OpenMP provides convenient parallelization features at an abstract level, that is, easy creation/destruction of threads and basic synchronization primitives such as locks and barriers. The Thread library, on the other hand, provides advanced synchronization primitives, such as conditional variables, which are of crucial importance in our design.

[2]on an 11th Gen Intel(R) Core(TM) i7-1165G7 processor @ 2.8 GHz

We create/destroy threads using the OpenMP API. Moreover, threads going into busy wait mode, wait for a condition to be satisfied to assume a new task. To implement this functionality and facilitate inter-thread communication we use a combination of the `std::condition_variable` and `std::mutex` classes from the `C++` Thread library. This synchronization primitive blocks a thread, until a shared variable is modified by another thread (the condition). In our case, this shared variable is the task-queue. Each time a working thread submits a new task into the queue, it thereby notifies all inactive threads, and one of them will subsequently dequeue the task. To guarantee that the queue is accessed or modified by a single thread at a time, we use OpenMP locks.

Mixing the OpenMP API with the Thread Library is not problematic in our case, since the creation/destruction of threads is exclusively controlled by OpenMP, while the use of the Thread Library is limited thread synchronization via conditional variables. Furthermore, Gentrius remains portable in all currently compliant devices, because it is implemented in IQ-TREE 2 [15], which in turn is written in `C/C++` and requires OpenMP as a prerequisite to compile.

In the original sequential version of Gentrius, the number of stand trees, intermediate states, and dead ends is stored in global variables. Each time the algorithm generates a new state by inserting a new taxon, it updates the respective global variable and checks if one of the stopping rules is satisfied. To maintain this functionality in the parallel version, we deploy shared-memory atomic integer variables via the `std::atomic` template. We use them to protect the counters for stand trees, intermediate states, and dead ends. While the dead ends do not form part of the stopping rules, they are printed in the output. Note that, as it was mentioned in Section III-A, a single thread approximately visits hundreds of thousands of states per second, depending on the number of constraint trees and their corresponding topologies. This translates into hundreds of thousands shared-memory writes per second, or one write every few micro-seconds. In modern multiprocessing systems architectures, the cost of atomic primitives is estimated to require up to a few thousand CPU cycles [18]. This translates into a time requirement to update these counters in the order of hundreds to thousands nanoseconds. As the number of threads increases, the probability of lock congestion when updating these counters increases as well. To avoid lock congestion we modified the counter updates. Each thread only updates the global stand trees counter each time it has counted 2^{10} stand trees. For the global intermediate states counter the restriction is 2^{13} states, while for the global dead ends counter the increment interval is, again set to 2^{10}. Those settings have been empirically determined. We observed an average parallel speedup improvement of $2-5\%$ among datasets when 16 threads were used. For example, in dataset `emp-data-3802` the speedup improved by 4%. Each time a thread updates a global variable, it also checks whether the stopping rule for the corresponding variable is satisfied and, if so, notifies all threads to terminate. Evidently, this might lead parallel Gentrius to occasionally exceed the limits specified by the stopping rules. This does not constitute a problem, however, due to the capacity of Gentrius to process hundreds of thousand states per second. In practice, the stopping rules are exceeded by a few thousand stand trees/intermediate states or a few milliseconds of execution time, while the actual specified boundaries are in the order of millions stand trees/intermediate states and hours of execution time.

The required data structures for our analysis are stored in an object of type `Terrace` class, comprising the agile tree, the constraint trees and the double-edged mappings. Functionalities for taxon insertion/removal and mapping updates are also implemented in the `Terrace` class. The trees are implemented based on the respective IQ-TREE 2 template. As mentioned before, at startup, each thread initializes its own private `Terrace` object and operates on it consistently. Finally, a task is a structure of arrays containing the information about the taxa that need to be inserted.

IV. Performance Evaluation

To evaluate the performance of parallel Gentrius, we ran experiments on both simulated (Section IV-B) and empirical (Section IV-C) datasets, up to 16 threads. In Section IV-A we discuss the difficulties we encountered in our performance assessment, since the Gentrius stopping rules distort the results. In some cases, and depending on the dataset, one encounters slowdowns, speedup plateaus, sub-linear, or even super-linear speedups. We explain why this is the case via simple examples. In Section IV-D we present a short analysis on how stopping rules 1 and 2 affect the parallel efficiency. Finally, in Section IV-E we anecdotally test scalability when more than 16 threads are used. As test platform we used an Intel(R) Xeon(R) Platinum 8260 48 double-core processor running at 2.4 GHz.

Initially, we thoroughly verified that the sequential and parallel versions yield the exact same results for all datasets, that is, the same number of stand trees, intermediate states, and dead ends. For some datasets, we also verified that the stands generated by the serial and parallel versions are identical, that is, they comprise the exact same trees.

A. Variance in Speedups

In general, small datasets yield slowdowns in parallel Gentrius. We consider a dataset as being "small" when the serial execution takes less than 1 second, although based on Figures 6 and 7 we could argue that the distinction between small and big data is more fuzzy. Such small datasets typically comprise up to a few thousand stand trees. Unsurprisingly, the slowdown is caused by the thread creation and destruction overhead, as well as inter-thread communication and task distribution, which are time consuming in relation to the small sequential runtime.

Apart from these slowdowns, we observed speedup plateaus as well as sub-linear and super-linear speedups in parallel Gentrius. The main cause for this behavior are the stopping rules and the unbalanced structure of the Gentrius branch-and-bound tree. A simple example for such an unbalanced branch-and-bound tree is depicted in Figure 5a. In this simple

example, one encounters a speedup plateau, as the unbalanced workflow structure forbids the working thread to create and push new tasks into the queue. We recall that, for threads to create new tasks, our parallelization scheme requires them to be in a state with two or more admissible branches for the next taxon, while two additional conditions need to hold; the number of tasks in the queue should not exceed a pre-specified upper limit, and the thread creating the task should be in a state with more than two remaining taxa to be inserted into its agile tree. In Figure 5a, if we assume an execution with two threads, the first thread chooses the left path on the initial split state, faces a dead-end and directly switches into busy-wait mode. On the other hand, the second thread chooses the right path and, after a sequence of intermediate states where each intermediate state only comprises one admissible branch, the second thread reaches the deepest levels of the workflow tree with less than three remaining taxa, where task creation for work-stealing is disallowed. Hence, the second thread is unable to push new tasks into the queue, neither during the sequence of intermediate states, nor after attaining the task creation threshold. Irrespective of the number of threads being used, the parallel runtime will be almost equal to the serial runtime. In this case we have reached a speedup plateau. We have observed such speedup plateaus of around 3x and 5x in simulated datasets `sim-data-1511`, `sim-data-1792`, `sim-data-1795`. The serial runtime for those datasets is below 10 seconds.

A second example of an unbalanced workflow structure is shown in Figure 5b. In serial execution, after the initial split state, the algorithm first descends into workflow subtree A. Since this part of the workflow graph contains zero stand trees, the algorithm can exceed the maximum intermediate state limit (second stopping rule) and terminate. When executing parallel Gentrius with two threads, the second thread concurrently proceeds to the workflow subtree B and counts more than a million stand trees. We observed this branch-and-bound workflow structure in dataset `sim-data-5001` under the default stopping rule parameter settings. In serial execution the algorithm terminates after 113 seconds, having counted 10 million intermediate states but zero stand trees. However, when executed in parallel using two threads, the algorithm counts 1 million stand trees within 5 seconds, resulting in a 22.6x speedup. Notably, even when the threshold is raised to 100 million intermediate states, serial execution is still unable to count any stand trees and terminates after reaching this state threshold in 1104 seconds, leading to a 220x speedup.

We can also apply Figure 5b when the first stopping rule is satisfied, assuming that in serial execution Gentrius first descends into workflow subtree B and terminates, after counting 1 million stand trees. In parallel execution, however, the speedup depends on the number of threads executing tasks from the workflow subtree B, where more than 1 million stand trees are detected, compared to the number of threads descending into workflow subtree A. In such cases the user might encounter speedups or plateaus. Overall, this behavior is a result of the concurrent parallel descent into different

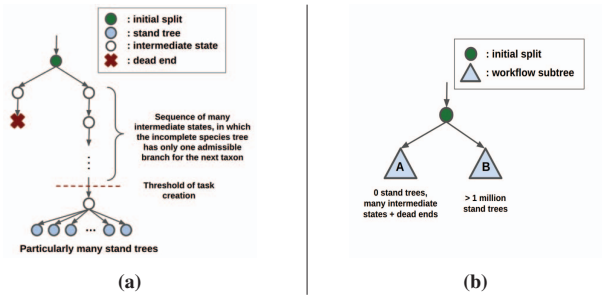


Fig. 5: (a) First example of an unbalanced workflow structure which induces a speedup plateau. **(b)** Second example of an unbalanced workflow structure, which induces a super-linear speedup or speedup plateau.

TABLE I: Adapted speedups of five datasets that reached the time limit under serial execution.

Dataset	Number of Threads				
	2	4	8	12	16
emp-data-5873	1.9	3.8	6.9	8.8	12.2
emp-data-43820	2.4	3.9	8.7	9.7	11.2
emp-data-55114	2.3	4.5	8.1	9.6	9.1
sim-data-4666	1.9	3.8	7.0	8.1	11.1
sim-data-4843	1.6	3.0	6.9	8.8	10.1

branches of the branch-and-bound workflow in conjunction with the stopping rules.

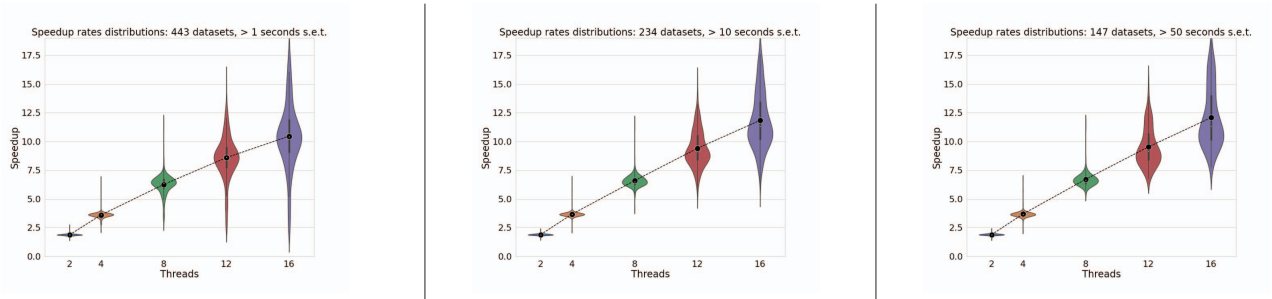
Finally, the third stopping rule sets an execution time threshold. We discuss one dataset that exhibited apparent unexpected behavior while running the main experiments on empirical data (Section IV-B), that is `emp-data-5873`. The stopping rule was set to five hours of execution time. The serial execution terminated, due to the stopping condition, after five hours $(18,000$ seconds) having counted $387,985,999$ stand trees. In contrast, parallel Gentrius managed to enumerate the entire stand, consisting of $485,240,625$ trees, without triggering any stopping rule; the execution with two threads required $11,333$ seconds. In this context, claiming that the speedup for two threads is $18,000/11,333 = 1.58$x underestimates the actual speedup. Because the direct comparison of execution times in such cases is inaccurate, we introduce the *adapted speedup* as a more appropriate metric. We thus define the *adapted* speedup for N threads (ASP_N) as

$$ASP_N = \frac{ST_N/T_N}{ST_1/T_1} = \frac{ST_N}{ST_1} \times SP_N$$

where ST_k, T_k denote the number of enumerated stand trees and the execution time when k threads are used, and SP_N is the standard speedup metric for N threads, calculated by simply dividing the corresponding execution times. Table I shows these adapted speedups for five datasets that reached the time limit threshold in serial execution.

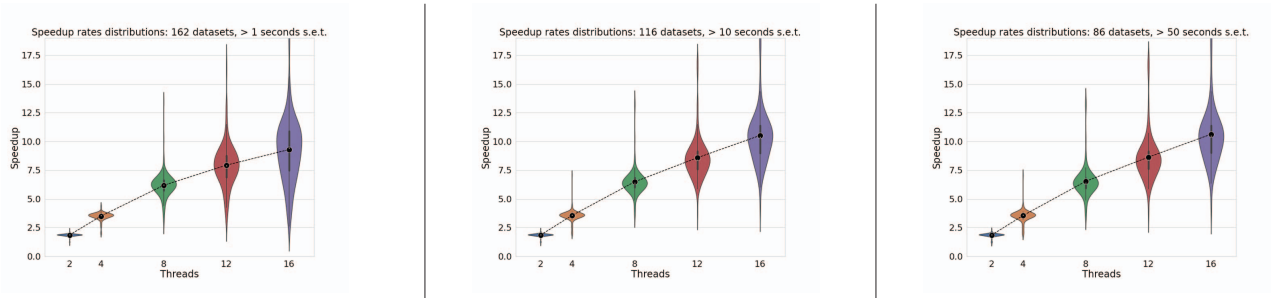
B. Simulated data

To assess the performance improvement on simulated data, we used the simulated instances from the original Gentrius



(a) **(b)** **(c)**

Fig. 6: Per thread speedup distributions on simulated data. The mean speedups are depicted by a dashed line (s.e.t. stands for 'serial execution time'). Speedup rates were calculated for: **(a)** 443 datasets have more than 1 second of s.e.t. **(b)** 234 more than 10 seconds s.e.t. **(c)**, 147 more than 50 seconds s.e.t.



(a) **(b)** **(c)**

Fig. 7: Per thread speedups on empirical data, the mean speedup values are depicted by a dashed line (s.e.t. stands for 'serial execution time'). Speedups were calculated for: **(a)** 162 datasets with more than 1 second s.e.t. **(b)**, 116 with more than 10 seconds s.e.t. **(c)**, 86 with more than 50 seconds s.e.t.

manuscript. The set comprises $4,997$ instances with varying sizes, different proportions of as well as distinct missing data patterns; more specifically, taxon numbers ranged between 50 and 300, locus numbers between 5 and 30, and the proportion of missing data ranged between 30% and 50%. We set the stopping rule parameters to $N := 10^9$ stand trees, $M := 10^9$ intermediate states, and $T := 5$ hours for all executions (both serial and multi-threaded). The reason why the values on rules 1 and 2 are equal is because, although it seems plausible to expect more intermediate states, in the vast majority of instances we end up counting more stand trees, due to the dynamic taxon insertion heuristic.

To avoid the difficulties described in Section IV-A and establish a fair base comparison, we filtered the datasets to extract instances that do not trigger any stopping rules. We did this by initially running parallel Gentrius for all datasets with 16 threads and keeping only those datasets for which we successfully calculated the entire stand. For these datasets we then re-ran the analysis using $N_t = \{12, 8, 4, 2, 1\}$ threads, making sure the time limit was not exceeded as the number of threads decreased. For the aforementioned reasons (see Section IV-A), small datasets were also excluded, resulting in 443 final datasets with serial execution times ranging from 1 second up to 4 hours.

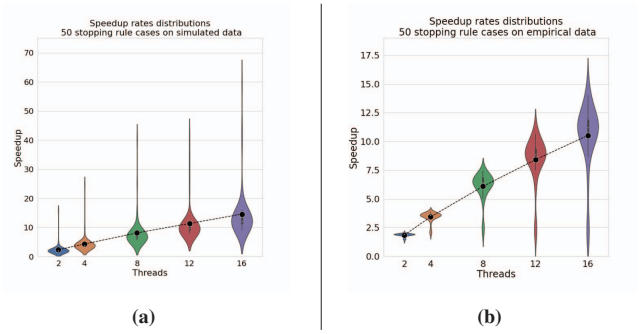
The results of our analysis on simulated data are summarized in Figure 6. In Figures 6b and 6c we have raised the threshold for small datasets from 1 second up to 10 and 50 seconds, respectively. We observe linear speedups with respect to the number of threads/cores used.

C. Empirical data

We used the RAxML Grove database as an empirical dataset source. We extracted $3,097$ datasets for our experiments. We applied the exact same pipeline as for simulated data to filter datasets and post-process results. This yielded 162 final datasets with serial execution times ranging between 1 second up to 3 hours. The results of our empirical data analysis are summarized in Figure 7. The results indicate a linear with respect to the number of threads/cores, when serial execution time is greater than 50 seconds.

D. Stopping rule cases

Next, we analyze the impact of stopping rules on parallel efficiency in more detail. We conducted short analyses of datasets that encounter stopping rules 1 or 2. We characterize these analyses as short, since we reduced the threshold value to 10 million stand trees and intermediate states respectively, so as to minimize the runtime requirements of these experiments. We collected 50 datasets of each type (empirical/simulated) and measured the speedups by dividing the serial by the



(a) **(b)**

Fig. 8: Speedup distribution on datasets that trigger stopping rules 1 or 2. Short analysis on: **(a)** 50 simulated datasets. **(b)** 50 empirical datasets.

TABLE II: Speedups for two large datasets, when more than 16 threads are used. The serial execution times for the two datasets are 11,199.6 (`emp-data-60587`) and 17,163.4 (`sim-data-4677`) seconds respectively

Dataset	Number of Threads		
	16	32	48
emp-data-60587	12.0	20.37	26.18
sim-data-4677	13.43	23.01	29.46

parallel execution times, although as we argued in Section IV-A this comparison might sometimes be misleading.

Figure 8 summarizes the results of our analysis. The distribution of speedups on both simulated and empirical data is substantially distorted, particularly so in the case of simulated data where we observe a super-linear speedup for a few datasets, for example in `sr_sim-data-44`. The tree-like workflow structure of this specific dataset turns out to be highly unbalanced, yielding a misleading impression that it must comprise a plethora of dead ends and no stand trees, when being analyzed with $N_t = \{1, 2\}$ threads. In contrast, when we use $N_t = \{4, 8, 12, 16\}$ threads, the algorithm terminates having counted 10 million stand trees, yielding speedups of 5x, 25x, 41x and 59x for 4, 8, 12 and 16 threads, respectively. This striking variance among results is an outcome of the unbalanced branch-and-bound workflow structure combined with how threads chose their paths along the workflow. The more threads are being used, the more efficient and faster the exploration for stand trees becomes for this specific dataset.

E. Scalability on more than 16 threads

To anecdotally test scalability when more than 16 threads are used, we selected two datasets with comparatively long sequential execution times. The two datasets are `emp-data-60587` and `sim-data-4677`, with sequential execution times of 11,199.6 and 17,163.4 seconds, respectively. The calculated speedups for executions using $N_t = \{16, 32, 48\}$ threads are shown in Table II.

V. Discussion

We described, implemented, evaluated, and made available an efficient parallel open-source version of Gentrius. Fast threads that finish their own tasks early can subsequently contribute to accelerating slower threads in completing their tasks via a thread pooling mechanism that implements work stealing. The parallel version of Gentrius now allows for seamless computation of stands on large datasets using off-the-shelf multi-core desktop and server systems.

Apart from the fact that we provide a faster solution for counting trees on a stand, in some cases the multi-threaded execution identifies trees on a stand that the serial execution fails to detect in reasonable time. This is due to the parallel descent into the distinct branches of the branch-and-bound algorithm in conjunction with the implemented stopping rules. Thus, multi-threaded execution allows for a potentially faster and more efficient exploration of the workflow graph.

In terms of future work, we intend to explore different heuristics for the taxon insertions order that can potentially further increase parallel efficiency. Finally, we aim to redesign the data structure and functions for mappings between branches of trees, since, based on profiling results with `Valgrind`, updating these mappings to add or remove taxa consumes $15 - 30\%$ of total runtime, depending on the dataset.

Summarising, identifying stands constitutes a crucial step in species tree inference from multiple loci and investigation of the uncertainty due to missing data. Hence, efficient bioinformatic tools such as parallelised Gentrius implementation presented in this paper can be very helpful for pipelines and methods predicting the difficulty of a phylogenetic analysis such as recent systematic attempt by Haag *et al.* [19].

Acknowledgment

This work was financially supported by the Klaus Tschira Foundation and by the European Union (EU) under Grant Agreement No 101087081 (Comp-Biodiv-GR).

Funded by the European Union

References

[1] J. Felsenstein, "Evolutionary trees from dna sequences: A maximum likelihood approach," *Journal of Molecular Evolution*, vol. 17, p. 368–376, 11 1981. [Online]. Available: https://doi.org/10.1007/BF01734359

[2] Z. Yang, *Molecular Evolution: A Statistical Approach*. OUP Oxford, 2014. [Online]. Available: https://books.google.gr/books?id=T-LoAwAAQBAJ

[3] B. Morel, A. M. Kozlov, A. Stamatakis, and G. J. Szöllősi, "GeneRax: A Tool for Species-Tree-Aware Maximum Likelihood-Based Gene Family Tree Inference under Gene Duplication, Transfer, and Loss," *Molecular Biology and Evolution*, vol. 37, no. 9, pp. 2763–2774, 06 2020. [Online]. Available: https://doi.org/10.1093/molbev/msaa141

[4] M. Rabiee, E. Sayyari, and S. Mirarab, "Multi-allele species reconstruction using astral," *Molecular Phylogenetics and Evolution*, vol. 130, pp. 286–296, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1055790317308424

[5] A. Stamatakis and N. Alachiotis, "Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data," *Bioinformatics*, vol. 26, no. 12, pp. i132–i139, 06 2010. [Online]. Available: https://doi.org/10.1093/bioinformatics/btq205

[6] M. J. Sanderson, M. M. McMahon, and M. Steel, "Terraces in phylogenetic tree space," *Science*, vol. 333, no. 6041, pp. 448–450, 2011. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.1206357

[7] M. J. Sanderson, M. M. McMahon, A. Stamatakis, D. J. Zwickl, and M. Steel, "Impacts of Terraces on Phylogenetic Inference," *Systematic Biology*, vol. 64, no. 5, pp. 709–726, 05 2015. [Online]. Available: https://doi.org/10.1093/sysbio/syv024

[8] I. T. Farah, M. M. Islam, K. T. Zinat, A. H. Rahman, and M. S. Bayzid, "Phylogenomic terraces: presence and implication in species tree estimation from gene trees," *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/04/20/2020.04.19.048843

[9] M. Habib, A. H. Rahman, and M. S. Bayzid, "Terraces in species tree inference from gene trees," *bioRxiv*, 2022. [Online]. Available: https://www.biorxiv.org/content/early/2022/11/24/2022.11.21.517454

[10] D. Höhler, W. Pfeiffer, V. Ioannidis, H. Stockinger, and A. Stamatakis, "RAxML Grove: an empirical phylogenetic tree database," *Bioinformatics*, vol. 38, no. 6, pp. 1741–1742, 12 2021. [Online]. Available: https://doi.org/10.1093/bioinformatics/btab863

[11] M. Constantinescu and D. Sankoff, "An efficient algorithm for supertrees," *Journal of Classification*, vol. 12, p. 101–112, 3 1995. [Online]. Available: https://doi.org/10.1007/BF01202270

[12] "Terraphy tool," https://github.com/zwickl/terraphy, accessed: 2022-10-13.

[13] R. Biczok, P. Bozsoky, P. Eisenmann, J. Ernst, T. Ribizel, F. Scholz, A. Trefzer, F. Weber, M. Hamann, and A. Stamatakis, "Two C++ libraries for counting trees on a phylogenetic terrace," *Bioinformatics*, vol. 34, no. 19, pp. 3399–3401, 05 2018. [Online]. Available: https://doi.org/10.1093/bioinformatics/bty384

[14] O. Chernomor, C. Elgert, and A. von Haeseler, "Identifying equally scoring trees in phylogenomics with incomplete data using gentrius," *bioRxiv*, 2023. [Online]. Available: https://doi.org/10.1101/2023.01.19.524678

[15] B. Q. Minh, H. A. Schmidt, O. Chernomor, D. Schrempf, M. D. Woodhams, A. von Haeseler, and R. Lanfear, "IQ-TREE 2: New Models and Efficient Methods for Phylogenetic Inference in the Genomic Era," *Molecular Biology and Evolution*, vol. 37, no. 5, pp. 1530–1534, 02 2020. [Online]. Available: https://doi.org/10.1093/molbev/msaa015

[16] M. Bordewich, C. Semple, and J. Talbot, "Counting consistent phylogenetic trees is #p-complete," *Advances in Applied Mathematics*, vol. 33, no. 2, pp. 416–430, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0196885804000107

[17] S. Böcker, "Exponentially many supertrees," *Applied Mathematics Letters*, vol. 15, no. 7, pp. 861–865, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S089396590200054X

[18] F. Hoseini, A. Atalar, and P. Tsigas, "Modeling the performance of atomic primitives on modern architectures," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337901

[19] J. Haag, D. Höhler, B. Bettisworth, and A. Stamatakis, "From Easy to Hopeless—Predicting the Difficulty of Phylogenetic Analyses," *Molecular Biology and Evolution*, vol. 39, no. 12, 11 2022, msac254. [Online]. Available: https://doi.org/10.1093/molbev/msac254