Karlsruhe Institute of Technology

Load-Balance and Fault-Tolerance for Massively Parallel Phylogenetic Inference

Master's Thesis of

Klaus Lukas Hübner

at the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Reviewer:	Prof. Dr. Alexandros Stamatakis
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dr. Alexey Kozlov
Second advisor:	M.Sc. Demian Hespe

01 January 2020 – 30 June 2020

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 30 June, 2020

Klaus Lukas Hübner

.....

(Klaus Lukas Hübner)

Abstract

Upcoming exascale supercomputers will comprise hundreds of thousands of CPUs. Scientific applications on these supercomputers will face two major challenges: Hardware failures, and parallelization efficiency. We extend RAxML-ng, a widely used tool to build phylogenetic trees, to mitigate hardware failures without user intervention. For this, we increase the checkpointing frequency. We also detect failures, redistribute the work among the surviving ranks, restore a consistent search state, and restart the tree search automatically. RAxML-ng now supports fault tolerance in the tree search mode, using multiple starting trees, and multiple alignment data partitions. RAxML-ng can handle multiple failures at once as well as multiple successive failures. There is no limit on the number of failures that can occur simultaneously or sequentially. We also support mitigating failures which occur during the recovery of a previous failure or during checkpointing. In contrast to the previously available manual recovery scheme, a recovery is initiated automatically after a failure, that is, the user does not have to take any action. We benchmark our algorithms for checkpointing and recovery. In our experiments, creating a checkpoint of the model parameters requires at most 72.0 ± 0.9 ms (400 ranks, 4,116 partitions). Creating a checkpoint of the tree topology requires at most 0.575 ± 0.006 ms (1,879 taxa). The overall runtime of RAxML-ng increases by a factor of 1.02 ± 0.02 when using the new checkpointing scheme and by a factor of 1.08 ± 0.07 when using the new checkpointing scheme and ULFM v4.0.2u1 as the MPI implementation. Restoring the search state after a failure requires at most 535 ± 19 ms. We simulated up to ten failures, which causes the overall runtime to increase by a factor of 1.3 ± 0.2. We also describe multiple approaches on how to store the MSA data, which has to be re-read after a failure, redundantly in memory to avoid disc-accesses after a failure. RAxML-ng synchronizes thousands of times per second. How equally the load balancer distributes the work across the CPUs therefore directly influences the overall runtime. We find that some ranks require up to 30 % more time to process their portion of the work than the average rank does. We also find that a single rank sometimes requires the most time to process the current portion of work in 30 % of all iterations. We identify the site-repeats feature (an algorithmic optimization that avoids redundant computations) as the cause of this imbalance. We also present algorithms to solve the multi-sender h-relation problem and the unilaterally-saturating b-matching problem. The multi-sender h-relation problem is a variant of the h-relation problem in which each package can be received by any CPU in a set of valid sources. The unilaterally-saturating b-matching problem is a variant of the b-matching problem in bipartite graphs. In the b-matching problem, a function $b(v)$ defines an upper-bound for the number of matching edges each vertex v might be incident to. The matching is called unilaterally-saturating, if for one of the two sets of the bipartite graph, each vertex is incident to at least one matching edge.

Zusammenfassung

Zukünftige Exascale Supercomputer werden aus hunderttausenden CPUs bestehen. Wissenschaftliche Anwendungen auf diesen Supercomputern werden mit zwei großen Herausforderungen konfrontiert: Hardwareausfälle und effiziente Parallelisierung. Wir erweitern RAxML-ng, ein weitverbreitetes Softwarewerkzeug um Phylogenetische Bäume zu bauen, um die Funktion Hardwareausfälle ohne Eingreifen des Anwenders zu behandeln. Dafür erhöhen wir die Frequenz in welcher an Kontrollpunkten eine Sicherung des Zustands der Suche erstellt wird. Wir stellen Fehler automatisch fest und verteilen im Fehlerfall die Arbeit an die überlebenden Ranks, stellen einen konsistenten Suchzustand wieder her und starten die Baumsuche neu. All dies geschieht, im Gegensatz zum bisherigen Manuellen Wiederherstellungsschema, ohne Einwirken des Nutzers. RAxML-ng unterstützt nun Ausfallsicherheit bei der Baumsuche, mit mehreren Startbäumen und mehreren Partitionen der Alignmentdaten. RAxML-ng kann mehrere gleichzeitig und nacheinander auftretende Ausfälle behandeln. Es gibt dabei keine Obergrenze für die Anzahl der Ausfälle die gleichzeitig oder hintereinander auftreten dürfen. Wir unterstützten zudem die korrekte Behandlung von Ausfällen die während dem Wiederherstellungsvorgang eines vorhergehenden Ausfalls oder während dem Speichern des Suchzustands an einem Kontrollpunkt auftreten. Wir messen die Laufzeit unserer Algorithmen für das Erstellen von Kontrollpunkten und der Wiederherstellung des Suchzustands. In unseren Experimenten dauert das Erstellen eines Kontrollpuntes der Modelparameter höchstens 72.0 ± 0.9 ms (400 Ranks, 4,116 Partitionen). Das Erstellen eines Kontrollpunktes der Baumtopologie dauert höchstens 0.575 ± 0.006 ms (1,879 Taxa). Die Gesamtlaufzeit von RAxML-ng erhöht sich um den Faktor 1.02 ± 0.02 wenn wir das neue Kontrollpunktschema verwenden und um den Faktor 1.08 ± 0.07 wenn wir das neue Kontrollpunktschema und ULFM v4.0.2u1 als MPI Implementierung verwenden. Den Suchzustand nach einem Hardwareausfall wiederherzustellen dauert höchstens 535 ± 19 ms. Wir haben bis zu zehn Hardwareausfälle simuliert, wodurch sich die Laufzeit um den Faktor 1.3 ± 0.2 verlängert hat. Wir beschreiben zudem mehrere Ansätze um die MSA-Daten, welche nach jedem Ausfall neu gelesen werden müssen, redundant im Arbeitsspeicher der Ranks vorzuhalten und so die Festplattenzugriffe zu vermeiden. RAxML-ng synchronisiert tausende Male pro Sekunde. Wie gleichmäßig der Lastenverteilungsalgorithmus die Last auf die CPUs verteilt beeinflusst also direkt die Gesamtlaufzeit. Wir stellen fest, dass manche Ranks bis zu 30 % mehr Zeit benötigen um ihren Teil der Arbeit zu erledigen als ein Rank im Durchschnitt benötigt. Wir stellen zudem fest, dass manchmal ein Rank in 30 % aller Iterationen die meiste Zeit benötigt um sein Arbeitspaket abzuarbeiten. Wir identifizieren die Site-Repeats Funktion (eine Algorithmische Optimierung welche redundante Berechnungen vermeidet) als den Ursprung dieser Ungleichverteilung. Wir stellen weiter Algorithmen vor, welche

das multi-sender h-relation Problem und das unilaterally–saturating (einseitig sättigend) b-matching Problem lösen. Das multi-sender h-relation Problem ist eine Variante des h-relation Problems in welchem jedes Datenpaket von mehren CPUs empfangen werden kann. Das unilaterally–saturating b-matching Problem ist eine Variante des b-matching Problems in bipartiten Graphen. Beim b-matching Problem, definiert eine Funktion $b(v)$ eine Obergrenze für die Anzahl der Matchingkanten zu welcher jeder Knoten inzident sein darf. Wir nennen ein Matching unilaterally-saturating, falls für eine der beiden Knotenmengen des Bipartiten Graphen jeder Knoten inzident zu mindestens einer Matchingkante ist.

Contents

List of Figures

List of Tables

Part I.

Introduction

1. Introduction

1.1. Motivation

Phylogenetics is the study of the history of the evolution of species [28]. Phylogenetic analysis is important in the fields of cancer research [96], viral infectious research [67, 106], wild life conservation [34], drug discovery [9], and of course for inferring the tree of life [38, 117]. Phylogenetic analysis on today's large datasets requires multiple days of CPU time [70] and terabytes of memory [55, 77]. Most available High Performance Computing (HPC) systems do not have this much memory available on a single node. We therefore have to parallelize tree searches on large datasets. Additionally, we will obtain our results faster if we are using parallelization.

Algorithms which perform phylogenetic tree searches on HPC systems, synchronize thousands of times per second (see Appendix A.2.4). Each rank (see Section 1.4) has to wait for all other ranks to finish their current share of work at each synchronization point. To reduce the runtime of the algorithm, it is therefore important that we distribute the work evenly.

Parallelizing a program over an increasing number of nodes (see Section 1.4) poses additional difficulties. Failing hardware is projected to be one of the main challenges in future exascale systems [97]. It is reasonable to expect that a hardware failure will occur in exascale-systems every 30 to 60 min [16, 25, 99]. To the best of our knowledge, no phylogenetic inference software is currently able to handle hardware-failures without user intervention.

1.2. Scientific Contribution

RAxML-ng [69, 102] is the successor of RAxML, one of the most used and cited tools for phylogenetic inference. RAxML-ng has been used to infer a phylogenetic tree on over 12,000 cores in parallel, for example on bird genomes (unpublished). Its predecessor ExaML, a dedicated predecessor for supercomputers, has been used to infer a phylogenetic tree on over 4,000 cores in parallel [55]. We expect the need for highly parallel runs to increase as the size of molecular datasets doubles every 18 months [41].

We designed and implemented an improved failure-mitigation strategy for RAxML-ng. Currently, RAxML-ng uses a checkpoint/restart scheme. We can create checkpoints only at certain points in the algorithm (see Section 6.2). Multiple hours can pass in-between checkpoints. We increased the checkpoint frequency and made them more fine-grained (see Section 6.2). We also added the ability to handle rank failures without user intervention. We detect failures, redistribute the work among the surviving ranks, and restart the tree search

without user intervention (see Section 6.3). We benchmark our algorithms for checkpointing and recovery (see Section 6.2.3 and Section 6.3.3).

When recovering after a rank failure, we redistribute the work among the surviving ranks. The ranks which obtain additional work have to load the data they need for their computations from disk. We describe three approaches on how to eliminate this disk access by storing the data redundantly in the memory of the compute nodes (see Chapter 7). To the best of our knowledge, no research into low-latency access and redundant storage without replacement of failed ranks has been published to date (see Section 7.2.2).

We assess how equally RAxML-ng's load balancer distributes the work across all ranks. For this, we developed our own low-overhead profiling code and measured how long each rank requires for each portion of work between two synchronization points (see Section 4.5). We found that, over the runtime of the algorithm, some ranks require up to 30 % more time than the average rank does (see Section 4.5.2). We proceed to show, that it is possible that a single rank requires the most time to process the current portion of work in 30 % of all cases (see Section 4.5.4).

We also present algorithms to solve the multi-sender h-relation problem and the unilaterally-saturating b-matching problem. The multi-sender h-relation problem is a variant of the h-relation problem (see Section 7.2.2.1) in which each package can be received from any PE in a set of valid sources (Section 7.2.5). The unilaterally-saturating b-matching problem is a variant of the b-matching problem in bipartite graphs. In the b-matching problem, a function $b(v)$ defines an upper-bound for the number of matching edges each vertex v might be incident to. The matching is called unilaterally-saturating, if for one of the two sets of the bipartite graph, each vertex is incident to at least one matching edge (see Section 7.2.6). To the best of our knowledge, there do not exists any published algorithms to solve these two problems.

1.3. Structure of this Master's Thesis

We first introduce phylogenetic tree inference (Chapter 2) and how we parallelize a phylogenetic tree search (Chapter 3). We then proceed to profile the phylogenetic tree search of RAxML-ng (Chapter 4). In Chapter 5 and Chapter 6 we describe our implementation of a failure tolerant phylogenetic tree search in RAxML-ng. In Chapter 7 we then discuss approaches to redundant in-memory storage of static data across a distributed memory system. We propose to use these techniques to eliminate the disk accesses which ranks need to perform when they load new data to work on after a rank failure. We conclude with a discussion (Chapter 8) and discuss future work in Chapter 9.

1.4. Nomenclature: CPUs, Ranks, Nodes, and PEs

A word about nomenclature. Modern HPC systems comprise multiple computers connected over a network (see Section 4.2). We call each of these computers a "node". Each node has its own main memory, that is, we are considering different nodes to be distributed memory machines. Each node may have multiple CPUs and each CPU has multiple physical cores. All cores on a single node access the same main memory using a common address space and therefore constitute a shared memory system. A single process can run on one or more cores, possibly even on multiple CPUs of the same node. A single process will never run on multiple nodes.

In a distributed memory system, the processes communicate over messages which they pass over a network. The Message Passing Interface (MPI) is a standard describing message passing primitives, for example broadcast or reduce (see Section 3.4). A collective MPI operation is an operation in which each process of the application participates. When using MPI, multiple processes on the same node communicate via messages, too. If we want to leverage communication via the shared memory, we have to implement this using for example PThreads (see Section 3.1). We call each MPI process a "rank". In this thesis, we will not spawn multiple threads on a single rank. Consequently, one rank will always run on one CPU core. Multiple ranks might run on different cores of one CPU.

In theoretical computer science, the concept of Processing Elements (PEs) exists. In this thesis, a PE is equal to an MPI rank, that is, one process running on one core of one node, communicating with other PEs via messages. We will use the terms rank or PE depending on the context.

2. Phylogenetic Tree Inference

Phylogenetics is the study of the history of the evolution of a species [28]. Phylogenetic analysis is important in the fields of cancer research [96], viral infectious research [67, 106], wild life conservation [34], drug discovery [9], and of course for inferring the tree of life [38, 117]. Reconstructing past events is always hard, especially when most of them happened millions of years ago. Up until decades ago, scientists had to rely on comparative anatomy and embryology to construct phylogenetic trees [28], for example, how many legs does an animal have in comparison to others. As scientists in the 21st century are the lucky few who can rely on computers to perform what we once did manually – only faster and better.

2.1. What are Phylogenetic Trees?

Today, phylogenetic researchers rely on molecular data, mainly Deoxyribonucleic Acid (DNA) sequences, for reconstructing phylogenetic trees. Typically, we can obtain only the genetic sequence of species that survived until this day, not of the ones which are already extinct. By using molecular sequences instead of morphological traits, we have more data at hand and can therefore build phylogenies with higher statistical confidence. Additionally, bacteria and viruses lack morphological traits. With molecular sequence data, we can still build phylogenies for them.

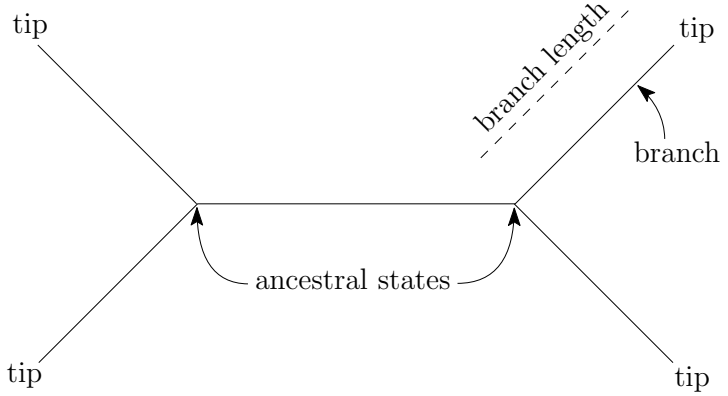


Figure 2.1.: Nomenclature of Phylogenetic Trees. A tree consists of a tree topology and branch lengths. Tips (leafs) have a degree of 1 (one neighbour). Ancestral states (inner nodes) have a degree of 3. Branches (edges) connect two nodes and have a length.

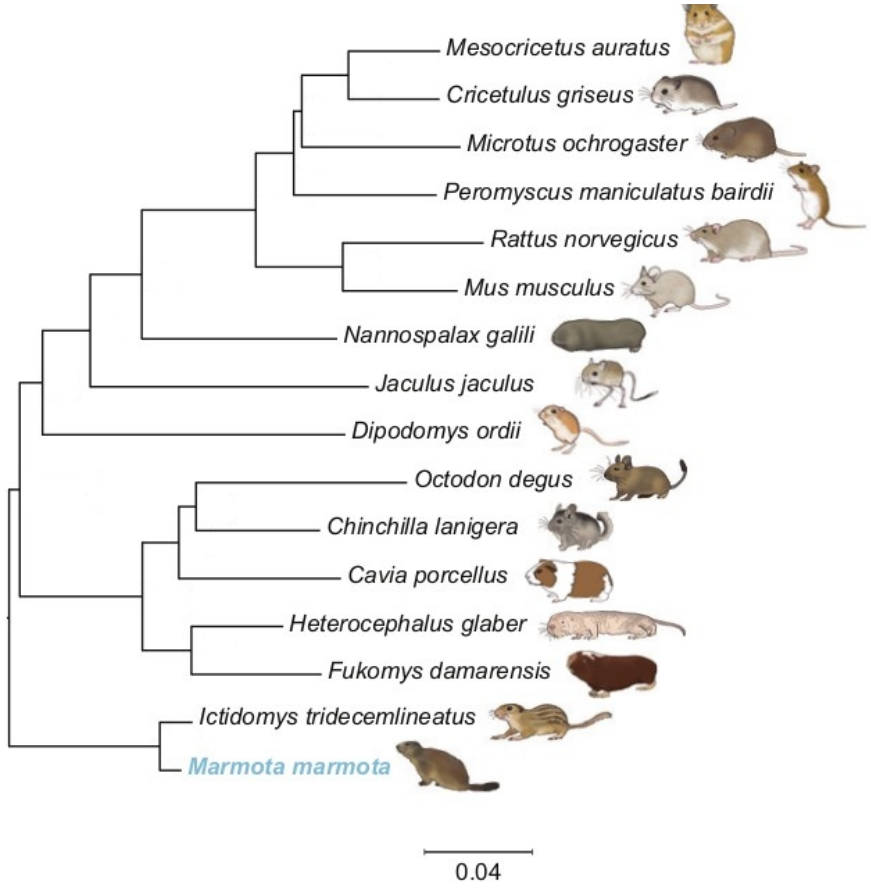


Figure 2.2.: Phylogenetic tree of *rodentia*. It was derived from multiple whole-genome alignments of protein-coding and non-coding sequences of rodent genomes [43].[1]; branch support values removed for simplicity. The taxa (here: rodents) are written at the tips.

By assuming that all life on earth originated from a common ancestor [23], we can draw a tree of life. The root of this tree is this common ancestor, the leaves are currently living species. Phylogenetic inference is the process of computing this tree. A phylogenetic tree consists of the topology, the branch lengths, and the sequences at the inner nodes.

When we compute phylogenetic trees, we are always assuming binary trees. In a binary tree, each node has either one (tip, leaf) or three (ancestral state, inner node) adjacent nodes (see Figure 2.1). We can simulate multifurcations by using multiple bifurcations which we connect with branches with a branch (edge) length of zero. Additionally, in most mathematical models we assume the tree to be unrooted [26, 101].

[1]Figure taken from `https://de.m.wikipedia.org/wiki/Datei:Phylogenetic_Tree_of_Rodentia.jpg`

2.2. Likelihood-Based Tree Inference

When researchers first started to compute phylogenies, they developed methods which assumed that more changes between two DNA sequences mean that more time has passed since they diverged from their common ancestor [118]. This assumption does not account for different parts of the DNA sequence mutating at different rates [73]. We will call this phenomenon rate heterogeneity among sites. A site is a position in the Amino Acid (AA) or DNA sequence. One cause of rate heterogeneity are different DNA repair efficiencies and DNA replication fidelities in different parts of the genome [10]. It is also possible, that mutations are reversed through a contrary mutation. In this case, both mutations cannot be observed in the existent sequences. This will lead to an underestimation of the distance between these two sequences [101].

Likelihood based phylogenetic inference tries to find *the* most likely tree among all possible trees [32]. That is, the tree (model) whose probability is the greatest, given the sequences (data). The input sequences have to aligned. We call this a Multiple Sequence Alignment (MSA). Sequence alignment has the goal to insert gaps of varying lengths into the sequences, such that those regions which share a common evolutionary history are aligned to each other. On possible heuristic for computing an MSA is to minimize the number of differences between the aligned sites of the MSA [18].

The likelihood of a tree does not represent the probability that this tree is the correct one. Phylogenetic tree inference models evolution over time and accounts for multiple mutations at the same position in the sequence as well as different rates of mutation along the sequence [33]. Multiple studies [45, 71, 116] showed that Likelihood-based methods of phylogenetic inference are able to reconstruct the true tree on simulated sequence data. Multiple open-source tools are available to perform likelihood-based phylogenetic tree inference, for example PhyML [45], FastTree [87], IQ-TREE [79], and RAxML/ExaML as well as its successor RAxML-ng [69, 102].

To search for *the* most likely tree, we must be able to evaluate the likelihood of a given tree, optimize the branch-lengths to obtain the maximum score for a particular tree, have a probabilistic model of nucleotide substitution, and efficiently search the space of valid tree topologies [101]. Finding *the* most likely tree is $\mathcal{NP}$-hard [22].

2.2.1. Calculating the Likelihood of a Given Tree and MSA

A probabilistic model for nucleotide substitution has to provide the probability of a sequence x^1 evolving into another sequence x^2 over a given period of time t. Both sequences must be aligned to each other (see Section 2.2). For computational simplicity, we assume, that different nucleotides x_i, x_j of the sequence evolve independently of each other. This assumption enables us to compute the likelihood of the whole sequence site by site by multiplying over the transition probabilities, that is:

$$P\left(x^1 \rightarrow x^2 \mid t\right) = \prod_i P\left(x_i^1 \rightarrow x_i^2 \mid t\right)$$

For each site, a function $P_{i,j}(t)$ describing the probability of mutation from nucleotide i to j is given with $i, j \in \{A, C, T, G\}$. We assume a Markov-process, that is, the probability $P_{i,j}(t)$ does not depend on previous mutations. We also assume time reversibility for these nucleotide transitions, that is, in the steady state, the number of transitions from state X to Y and from state Y to state X are the same. Let $\pi \in \{\pi = [0,1]^4 \mid \sum_{k=1}^{4} \pi_k = 1\}$ be the stationary frequencies of the Markov chain. The following then holds [26, 101]:

$$\forall i, j \in \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\} : \pi_i P_{i,j}(t) = \pi_j P_{j,i}(t)$$

When computing the likelihood of a substitution, it is therefore not important which sequence is the ancestor. The likelihood is the same independently of the direction of the transition.

Consider a set of n sequences x^j for $j = 1, \ldots, n$ which we will denote as x^*. These sequences have to be aligned to each other (see Section 2.2). Let T be a tree with n leaves with sequence x^j at leaf j. We will write t_* for the edge lengths of the tree. Given our model of evolution, we can define $P(x^* \mid T, t_*)$, that is, the probability of observing sequences x^* with tree topology T and branch lengths t_* [26]. We can now compute the likelihood of a phylogenetic tree, given the tree topology, the sequences at the tips, and the model of evolution. The model of evolution includes the nucleotide probabilities at the virtual root as well as the transition probabilities between nucleotide states. The virtual root can be any arbitrary node we choose to calculate the likelihood score of the given tree. As our transition probabilities are reversible, we will obtain the same likelihood score independently of where we place the virtual root.

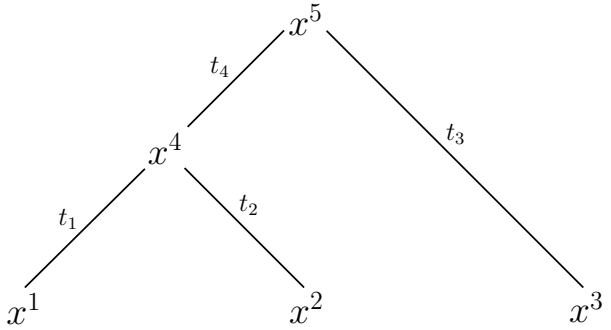


Figure 2.3.: An example phylogenetic tree with sequences $x^1, \ldots, x^5$. Sequence x^5 is the virtual root. Its probability $P(x^5)$ could be, for example, based upon the observed nucleotide frequencies. The probability of sequence x^j mutating into sequence x^i over time $t_{j,i}$ is given by $P(x^i \mid x^j, t_{j,i})$. It depends on the time that passed and the model of evolution we assume. We compute the likelihood of the tree by multiplying the probability of a sequence at the virtual root $P(x^5)$ with the probabilities of each transition $P(x^i \mid x^j, t_{j,i})$.

Let us consider the tree shown in Figure 2.3. We compute the likelihood of the tree by multiplying the probability of a sequence at the virtual root $P(x^5)$ with the probabilities of each transition $P(x^i \mid x^j, t_{j,i})$. That is:

$$P\left(x^1, \ldots, x^5 \mid T, t_*\right) = P\left(x^1 \mid x^4, t_1\right) \cdot \left(x^2 \mid x^4, t_2\right) \cdot P\left(x^3 \mid x^5, t_3\right) \cdot \left(x^4 \mid x^5, t_4\right) \cdot P\left(x^5\right)$$

We do not know the ancestral sequences if we are not using simulated data. To obtain the probability $P\left(x^1, \ldots, x^3 \mid T, t_*\right)$ of the known sequences for the given tree, we can sum efficiently over all possible ancestors x^4, x^5 using the Felsenstein pruning algorithm [31]. Given this method of evaluating the likelihood of a tree, we can search for *the* maximum likelihood tree. The maximum likelihood tree is the tree with the topology T and the branch lengths t_* which maximizes $P\left(x^* \mid T, t_*\right)$ [26].

2.2.1.1. Model of Evolution

The model of evolution consists of the probability of a sequence at the virtual root as well as the transition probabilities. We can, for example, estimate the probability of a sequence at a virtual root by computing the nucleotide frequencies in the data and assume that this was also the frequency at which the nucleotides were present at the time of the common ancestor [26].

We can model the rate at which different nucleotides mutate into each other using a variety of models. These models mainly differ in their degree of freedom. We could, for example, assume, that all nucleotides occur equally often and all transitions are equally likely. This simple model, known as the Jukes-Cantor model [56], has zero degrees of freedom. We do not need to optimize its parameters. We could also model each nucleotide frequency and transition separately. To ensure time-reversibility, however, the transition probabilities have to be symmetric, that is, $\pi_{\mathrm{A,G}} P(\mathrm{A} \rightarrow \mathrm{G}) = \pi_{\mathrm{G,A}} P(\mathrm{G} \rightarrow \mathrm{A})$. Because reversibility has to be maintained this model has 8 free parameters, which we can optimize [110].

The rate of mutation is not the same at all sites (see above). To account for this, Yang suggests a site-dependent variable, r_u, that scales all the t_* at the site u [121]. For given r_u, we can then compute the likelihood of a sequence as

$$P\left(x^* \mid T, t_*, r\right) = \prod_{u=1}^{N} P\left(x_u^* \mid T, r_u t_*\right)$$

We call this rate-heterogeneity. Since we do not know the values of r_u we have to integrate over all possible values, assuming that they are Γ distributed [26].

2.2.2. Overview of the Optimization Procedure

Finding *the* most likely tree is $\mathcal{NP}$-hard [22]. Even approximation is difficult as the number of possible tree topologies $\prod_{i=3}^{n}(2i-5)$ grows super-exponentially with the number of sequences [33]. There are for example $8x10^{21}$ possible rooted topologies for a set of 20 taxa [122]. Heuristics are thus needed to approximate the global maximum of the likelihood function. This Section will give an overview of heuristic used by RAxML-ng [68, 103, 105].

A tree is a tree topology with associated branch lengths. A phylogenetic tree is a tree with an associated evolutionary model (see Section 2.2.1.1). A tree search consists of multiple rounds of optimizing the tree topology, the branch lengths, and the evolutionary model.

RAxML-ng optimizes the tree topology by using Subtree Pruning and Regrafting (SPR) moves (see Section 2.2.2.1). The general idea of SPR-rounds is to move a subtree to a different position and keep the resulting topology if this move improved the likelihood of the tree. RAxML-ng repeats this procedure until the likelihood score does no longer improve. RAxML-ng uses Newton-Raphson, BFGS [36], and Brent [15] optimization methods for branch length and evolutionary model optimizations [101]. The algorithm of the tree search is described in the following Sections. An Overview is given in Algorithm 1.

Algorithm 1 Overview of the RAxML-ng search heuristic

```
1:  Optimize evolutionary model
2:  Optimize all branch lengths on starting topology
3:  Initial SPR rounds with increasing maximum distance
4:  Optimize evolutionary model
5:  repeat                                                    ▷ Fast SPR rounds
6:      Fast SPR iterations (no branch optimization)
7:      Insert nodes whose regrafting lead to the top 60 trees into BN
8:      for Node N ∈ BN do
9:          Prune and regraft N again, scoring in slow mode (with branch length optimization)
10:         Possibly insert resulting topology in the list BT of the 20 best scoring topologies
11:     end for
12:     for Topology T ∈ BT do
13:         Perform full branch length optimization on T
14:         Possibly update current best scoring topology T_best
15:     end for
16: until T_best not improved
17: Optimize evolutionary model
18: repeat                                                    ▷ Slow SPR rounds
19:     Slow SPR iterations; Possibly update list of 20 best scoring topologies BT
20:     for Topology T ∈ BT do
21:         Perform full branch length optimization on T
22:         Possibly update current best scoring topology T_best
23:     end for
24:     if T_best not improved then
25:         Increase rearrangement distance (maximum distance of an SPR move)
26:     end if
27: until T_best not improved and maximum rearrangement distance reached
28: Optimize evolutionary model
```

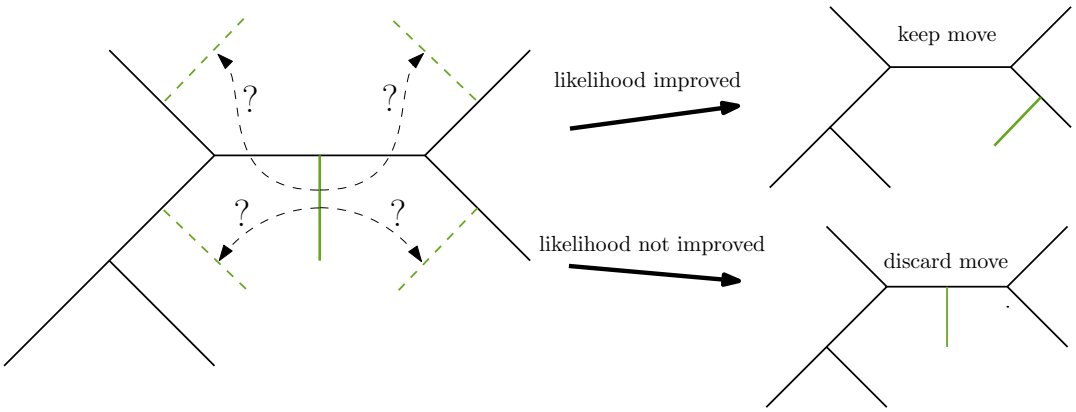


Figure 2.4.: Rearranging a subtree in a single iteration of a Subtree Pruning and Regrafting (SPR)-round. We consider only moves with distance of 1 in this example. If we improved the likelihood-score with the new topology, we conduct another iteration. Otherwise, the SPR-round is finished.

2.2.2.1. Subtree Pruning and Regrafting (SPR) Rounds

The RAxML-ng optimization procedure starts with a tree already containing all sequences. It uses either a random tree or a non-deterministically created parsimony tree as starting point for its likelihood optimization. RAxML-ng then refines this initial tree topology using SPR-moves.

One SPR iteration consists of removing (pruning) a subtree s from the currently best scoring tree and reinserting (regrafting) it into a neighbouring branch (see Figure 2.4). Over the course of one iteration, RAxML-ng tries all possible moves which are within the maximum rearrangement distance. We evaluate this new tree topology using the old branch lengths (fast mode) or after optimizing the branch lengths around the insertion node of the subtree (slow mode). If this new tree topology has a better likelihood-score than the original tree, we apply the SPR move. We continue the optimization using this new tree. We also keep the 20 top-scoring trees even if they do not improve the likelihood.

After each SPR iteration, we perform a full branch length optimization on the list of best scoring trees. If we find a new maximum likelihood tree, we keep it. If we found a higher-scoring tree in this SPR iteration, we conduct another one. The SPR round is finished, if we did not find any improvement to the current tree topology. During its tree search, RAxML-ng performs multiple slow and fast SPR-rounds with different rearrangement distances (see Algorithm 1).

2.2.2.2. Branch Length and Model Parameter Optimization

Next to the tree topology, RAxML-ng also optimizes the branch lengths and evolutionary models. As the transition probabilities are reversible (see Section 2.2.1), we can place a virtual root at any branch b_i or node of the tree. We can then use this to optimize each branch length

individually to maximize the likelihood. RAxML-ng repeatedly optimizes all branch lengths until the likelihood no longer improves. It is guaranteed, that the likelihood will constantly improve and eventually converge throughout this process [101].

The model of evolution also has free parameters which we have to optimize. This includes the nucleotide base-frequencies, substitution probabilities, and parameters for rate-heterogeneity (see Section 2.2.1.1). RAxML-ng uses Newton-Raphson, BFGS [36], and Brent [15] optimization for branch length and evolutionary model optimizations [101]. See Algorithm 1 for the points during a tree search where RAxML-ng optimizes the evolutionary model and the branch lengths.

Part II.

Profiling MPI-parallelized Phylogenetic Inference

3. Parallelization of Likelihood-Based Tree Inference

Phylogenetic inference on large datasets requires multiple days of CPU time [70] and terabytes of memory [55, 77]. Most available HPC systems do not have this much memory available on a single node. We therefore have to parallelize large tree searches. Additionally, we will obtain our results faster if we are using parallelization. Phylogenetic tree searches spend 85 to 98 % of their total runtime evaluating the likelihood-score of a given tree [2]. In this Chapter, we will describe the current parallelization strategy of RAxML-ng.

3.1. Parallelization Modes in RAxML-ng

RAxML-ng supports parallelization at three levels. At the single thread level, it uses parallelism as provided by the x86 vector intrinsics (SSE3, AVX, AVX2). At the single node level, RAxML-ng leverages the available cores by parallelization using PThreads. If we run RAxML-ng on a distributed memory HPC system, it uses parallelization via message passing (using MPI) [83, 104]. We can enable all three levels of parallelism at the same time. This is for example useful when running on a shared memory HPC system in which each multi-socket node comprises several multi-core CPUs, each supporting vector parallelism. In this thesis we do not consider PThreads parallelization. Instead, we run a separate MPI rank on each physical core of each multicore processor.

3.2. Parallelization Across Columns of a Multiple Sequence Alignment (MSA)

RAxML-ng is parallelized across the sites (columns) of the MSA (see Section 2.1). We can compute the likelihood of one sequence mutating into another sequence over a given time using the following formula (see Section 2.2.1):

$$P\left(s^1 \rightarrow s^2 \mid t\right) = \prod_i P\left(s_i^1 \rightarrow s_i^2 \mid t\right)$$

We can consequently evaluate all sites independently and multiply the resulting likelihoods at the end. We can parallelize the likelihood computations across the sites and compute their product using an allreduce operation (see Figure 3.1). This requires a single synchronization in

each likelihood calculation. When optimizing the branch lengths, for example with Newton-Raphson, we have to compute the first and second derivative. RAxML-ng parallelizes the calculation of the derivatives across sites, too. This requires two further allreduce operations.

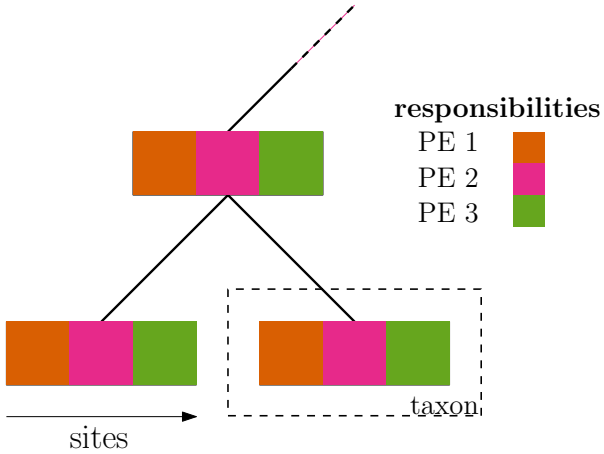


Figure 3.1.: Parallelization of Likelihood Computations. The load balancer assigns each PE a share of sites for which it has to compute the likelihood score. A PE always computes the likelihood score of its sites for the whole tree topology. The PEs then synchronize and compute the product of the likelihood-scores over all sites via an allreduce call.

3.3. Load Distribution, Partitions, and Site Repeats

Calculating the likelihood requires approximately the same time for each column of the MSA. That is, the workload on a PE is linear in the number of sites the load balancer assigns to it. This does not hold, when we expand our model to account for the fact that different parts of the genome evolve according to different evolutionary models (see Section 2.2.1). This is called partitioned analysis. Each partition consists of a set of sites with an associated evolutionary model consisting of transition probabilities, base frequencies, and branch length scalers. This allows different regions to evolve at different rates [94].

Managing an additional partition on a PE comes at a computational cost. The load balancer thus tries not to distribute a single partition among unnecessarily many PEs. This enables each PE to only keep those models updated that the PE needs for its local likelihood computations [94]. If two PEs were to have the same number of sites assigned to them, but these sites were drawn from a different number of partitions, the PE with more partitions would require more time to finish its likelihood computations. We want to avoid such an imbalance, as this causes every PE to wait for the slowest PE at every synchronization point.

It might happen, that two or more sites which belong to the same partition are identical inside a subtree. The likelihood-score of these sites will then be exactly the same. We consequently have to compute the likelihood-score only once and can then reuse it for all

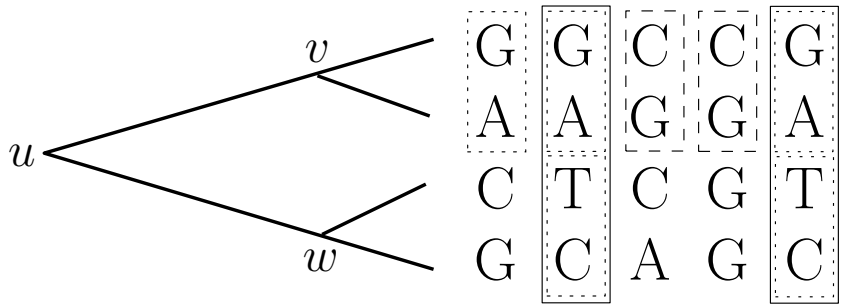


Figure 3.2.: Site Repeats. Subtree v has the site-repeat patterns `G|A` (dotted) and `C|G` (dashed). Subtree w has the site-repeat pattern `T|C` (dotted) and three not-repeated patterns. Subtree u has the site-repeat pattern `G|A|T|C` and three not repeated patterns. When we are evaluating a subtree, we have to compute the likelihood score only once for each site-repeat pattern and not repeated pattern. When evaluating v we therefore have to conduct likelihood-calculations of 2 patterns; 4 patterns when evaluating w, and 4 patterns when evaluating u. Without considering site-repeats, we would always have to compute the likelihood-score of 5 patterns.

identical sites in the current subtree. We call this technique "site-repeats" (see Figure 3.2). We could send the result of these likelihood computations over the network to other PEs which have the same patterns. We would, however, need to do this on each likelihood evaluation. This incurs too big of an overhead compared to re-computation to be feasible [94]. We therefore compute the likelihood for each pattern once on every PE and then reuse the result on this PE. This technique has been shown to speed up the overall runtime by a factor of 2 and decreases the memory used by up to 50 % [54].

3.4. Message Passing Primitives in RAxML-ng

RAxML-ng uses only one type of MPI operation during tree search: `MPI_Allreduce`. Consider an associative operation $\oplus$. Given data x_i on each PE i, a reduction computes [92]

$$\oplus_{i \leq p} = x_1 \oplus x_2 \oplus \cdots \oplus x_p$$

The difference between reduce and allreduce is, that allreduce will ensure that the final element is available at all PEs [112]. RAxML-ng uses `MPI_Allreduce` with addition as the reduction operator to compute the Log-Likelihood (LLH). Likelihoods tend to get very small. It is therefore more numerically stable to compute the log-likelihood, that is, the logarithm of the likelihood function. As the likelihood function is strictly increasing, maximizing the likelihood is equivalent to maximizing the log-likelihood. RAxML-ng also uses `MPI_Allreduce` to compute the derivatives used in branch length optimization. RAxML-ng does not perform any other collective operation during the tree search. In other parts of the program, for example during checkpointing, RAxML-ng also conducts broadcasts and other MPI operations.

4. Profiling RAxML-ng

RAxML-ng conducts thousands of `MPI_Allreduce` operations per second (see Appendix A.2.4). Every one of these operations causes all MPI ranks to synchronize. This means, that all ranks have to wait for the slowest one. We profile RAxML-ng v0.9.0 (see Section 4.2) to quantify how this synchronization causes slowdowns. If the load balancer distributes computations ("work") unequally across ranks, some ranks will work longer than others. This causes the faster ranks to wait for the slowest one at each synchronization point. An imbalanced work distribution will therefore increase the overall runtime.

4.1. Measuring MPI Performance

We can measure the performance of MPI programs for example using the Profile Layer of MPI (PMPI). For instance Freeh *et al.* [39] and Rountree *et al.* [91] use PMPI for profiling. With PMPI, MPI allows the user to rewrite all `MPI_`$*$ functions. We can use this to add any functionality we desire, for example profiling code [113]. This approach is restricted to measuring the time spend inside of MPI calls and the time in-between them. It allows us to profile during production with minimal overhead. This would, in principle, enable us to implement dynamic rebalancing of the workload to reduce the runtime of RAxML-ng.

Another approach to profiling is to instrument the code using a compiler wrapper. For example Score-P [63] and Scalasca [123] provide such wrappers and associated helper programs. Using profiling libraries, for example Caliper [12], we have an even more fine-grained control over which parts of the program to profile. With these methods we can profile any parts of the code, not only those between MPI calls. They, however, incur a higher overhead if we profile too many code sections.

We choose to implement our own instrumentation for profiling. All MPI calls in RAxML-ng are already wrapped in the `ParallelContext` class. This enables us to profile them with only a few modifications to the codebase. Using our own instrumentation, we can also profile other parts of the code (see for example Section 6.2.3). We chose to write custom code instead of using a profiling library like Caliper [12], because this allows us to control the granularity and format of the measurement. In the experiments in Section 4.5.1 we want to measure and store how long a rank is working in a histogram with exponentially growing bins. In Section 4.5.2 we want to track how long a rank is working in a histogram of fractions/multiples of the median work duration. We verified the results obtained using our custom profiling with benchmarks performed using Arm MAP[1] and Scalasca [123]).

[1] `https://www.arm.com/products/development-tools/server-and-hpc/forge/map`

4.2. Hardware and Software Used

We conduct all experiments in this thesis on the ForHLR II supercomputer located at the Steinbruch Center for Computing (SCC) in Karlsruhe. It comprises 1,178 worker nodes. Each node is equipped with two sockets of Intel Xeon E5-2660 v3 (Haswell) Deca-Core CPUs. These CPUs run at a base clock rate of 2.1 GHz (max. 3.3 GHz) which results in a theoretical maximum throughput of 832 GFLOPS per node. Each CPU has 64 KiB L1-cache (per-core), 264 KiB L2-cache (per core), 25 MiB L3-Cache (shared), and a 2,133 MHz bus as well as 64 GiB RAM. All nodes are connected to each other via an InfiniBand 4X EDR interconnection [108]. In each experiment we describe how many nodes we use.

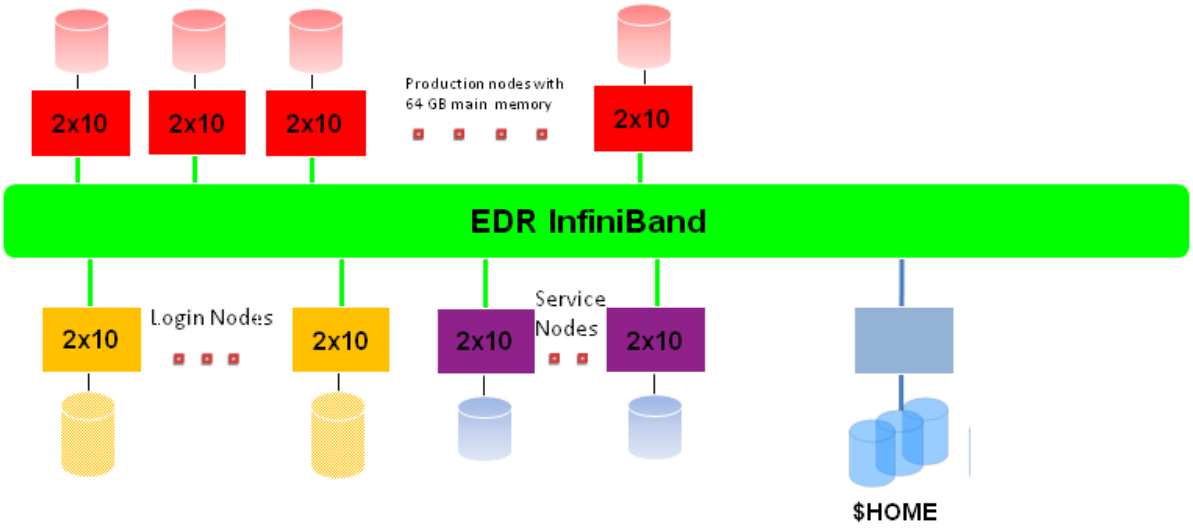


Figure 4.1.: ForHLR II architecture. A worker node comprises a two socket system with 10 CPU cores each. All worker nodes and the file server nodes are connected using EDR InfiniBand. Image taken from the ForHLR II's website[2]; simplified to only show the part of the infrastructure we use.

We store our input files on a Lustre distributed file system residing on a DDN ES7K RAID with 14 volumes. Each file is striped across 1 volume. We can read files from disk with a theoretical maximal performance of 2 GiB s^{-1} on a single node and 10 GiB s^{-1} across all nodes. Two file server nodes provide file access [107]. Each of them has identical hardware as the compute nodes. This is the default configuration on the ForHLR II. It is possible to, for example, increase the number of stripes or file servers serving the files. We do not use this feature for our experiments as we wanted to measure the performance in a typical use case, as most users of RAxML-ng will not hand-tune their file-system configuration.

[2] `https://www.scc.kit.edu/dienste/forhlr2.php`

All nodes are running Red Hat Enterprise Linux (RHEL) 7.x and Slurm 20.02.3. We use OpenMPI 3.1 and GCC 9.2 for our experiments where not mentioned otherwise. The RAxML-ng version we benchmarked was a few commits after the 0.9.0 release (`66ad9d2233` on branch `master`; 9th September 2019). Fault tolerant RAxML-ng is based upon `c2af275ae6` on branch `coarse` released on March 5th 2020.

4.3. Parameters Used

We conduct the profiling experiments with the following options when not noted otherwise. We only profile the phylogenetic tree search mode (see Section 2.2.2) and use parsimony starting trees. We document the random seeds we use in Appendix A.2. Tip-inner is turned off, pattern compression is turned on, per-rate scalers are turned off, site-repeats are turned on, the fast SPR radius is auto-detected, branch lengths scalers are proportional (ML estimate with NR-fast algorithm), the Single Instruction Multiple Data stream (SIMD) parallelization kernel is AVX2, and the number of threads per MPI rank is one. We analyse all datasets using one partition (see Section 3.3). See the RAxML-ng manual for details on these parameters.[3]

4.4. Datasets Used

For the experiments in this thesis, we selected empirical protein (AA) and DNA datasets with varying number of taxa (36 up to 815), alignment length (20,364 up to 21,410,970 sites), and partition count (1 to 4,116, see Table 4.1). The `fasta` and `model` files are available online.[4]

[3] `https://github.com/amkozlov/raxml-ng/wiki`
[4] `https://figshare.com/s/6123932e0a43280095ef`

Table 4.1.: Characteristics of the datasets used for evaluating RAxML-ng.

Designator	Data type	# taxa	# alignment sites	# unique patterns	# partitions	Reference
SongD1	DNA	37	1,338,678	746,408	1	Song *et al.* [100]
MisoD2a	DNA	144	1,240,377	1,142,662	100	Misof *et al.* [77]
XiD4	DNA	46	239,763	165,781	1	Xi *et al.* [119]
PrumD6	DNA	200	394,684	236,674	75	Prum *et al.* [88]
TarvD7	DNA	36	21,410,970	8,520,738	1	Tarver *et al.* [109]
PeteD8	DNA	174	3,011,099	2,248,590	4,116	Peters *et al.* [82]
ShiD9	DNA	815	20,364	13,311	29	Shi and Rabosky [98]
NagyA1	AA	60	172,073	156,312	594	Nagy *et al.* [78]
ChenA4	AA	58	1,806,035	1,547,914	1	Chen *et al.* [19]
YangA8	AA	95	504,850	476,259	1,122	Yang *et al.* [120]
KatzA10	AA	798	34,991	34,937	1	Katz and Grant [58]
GitzA12	AA	1,897	18,328	18,303	1	Gitzendanner *et al.* [42]

4.5. Experiments

In this Section, we present the profiling results of RAxML-ng. We use the hardware and software we describe in Section 4.2 and the parameters we describe in Section 4.3. We summarize the datasets we use in Section 4.4. We analyse one tree search per configuration, measuring every `MPI_Allreduce` call and the time in-between `MPI_Allreduce` calls. We call the time in-between `MPI_Allreduce` calls "work packages". If we write a checkpoint between two `MPI_Allreduce` calls, we discard this measurement because we do not want to measure the checkpointing performance in this experiment. We measure thousands of `MPI_Allreduce` calls and therefore thousands of work packages per second (see Appendix A.2.4).

4.5.1. Absolute Time Required for Work and Communication

We measure the absolute time each rank takes to complete a code segment (see Figure 4.3). A code segment is either an `MPI_Allreduce` call (left) or a work package (right).

Each bar shows the data for a single rank. The colours are used to group the ranks by the physical node they run on. For example, the run on the ChenA4 dataset with 160 ranks (top-left) runs on 8 nodes, the run on ShiD9 using 20 ranks (bottom-right) runs on one node. Each bar depicts the distribution of all the measurements of the time required to process a work (right) or communication (left) package on this rank. A communication package is a

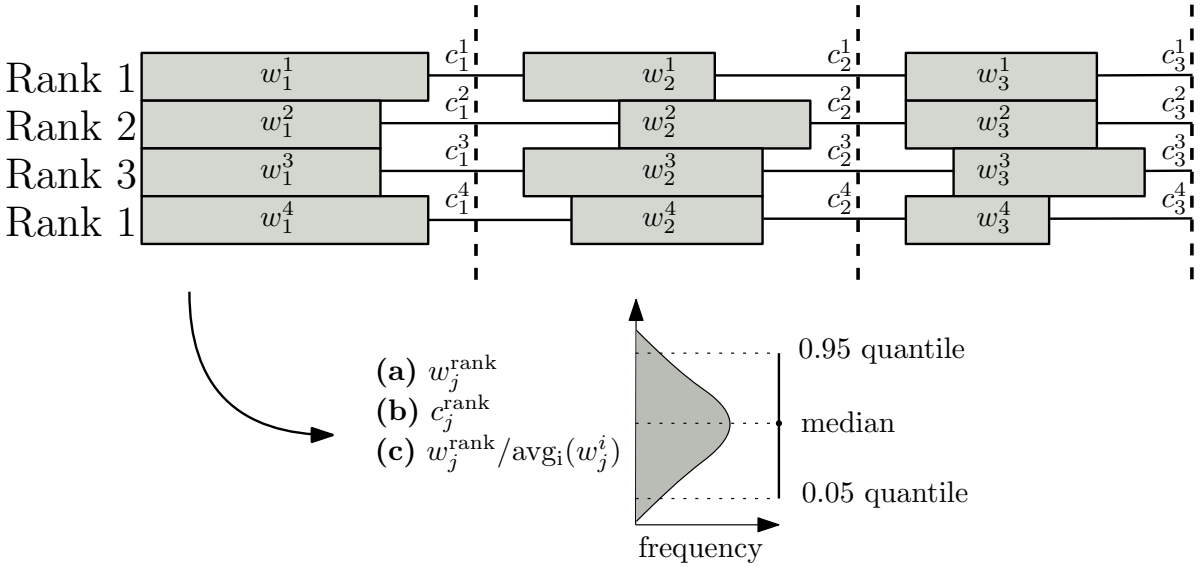


Figure 4.2.: All ranks process a work package (grey bars) each in parallel. When a rank finishes its work package, it enters the `MPI_Allreduce` call (horizontal line). Ranks wait for each other at a barrier (dashed line). The ranks exit the `MPI_Allreduce` non-synchronously and proceed with the next work package. Each rank measures the time it spends in each work package w_i^{rank} and communication package c_i^{rank}. As the ranks process thousands of work packages per second (see Appendix A.2.4), they can store their measurements only in histograms. **(a,b)** To show the time required for work/communication packages on each rank, the distribution on each rank is reduced to a single vertical bar. The upper end depicts the 0.95-quantile, the lower end the 0.05-quantile. For some measurements, other quantiles are used. The black dot depicts the median. The distribution of the time required to process work packages is not Gaussian! **(c)** For each measurement, we compute the average using an allreduce operation. We then compute how much longer each rank required than the average rank and store this Package-Specific Slowdon (PSS) in the histogram.

single `MPI_Allreduce` call. A work package is the time between two `MPI_Allreduce` calls. A bar ranges from the 0.01 to the 0.99 quantile of the times required on this rank. Black dots indicate the median time required (see Figure 4.2.a).

There is no way of knowing when exactly each rank enters or exits a code segment without synchronized clocks. If a rank finishes its work, we stop its work timer and start its `MPI_Allreduce` timer. The rank then immediately enters the MPI call. It waits inside the MPI call until all other ranks finished their work and arrive at the barrier of the MPI operation. For some runs, for example on the AA dataset `ChenA4` using 160 ranks (top-left), the time spent doing work is an order of magnitude higher than the time spent in `MPI_Allreduce` calls. For others runs, for example the run on the DNA dataset `SongD1` using 360 ranks (top-right), the time RAxML-ng spends in `MPI_Allreduce` calls and performing work is in the same order of magnitude. In some runs, for example on the DNA dataset `SongD1` with 360 ranks (top-right),

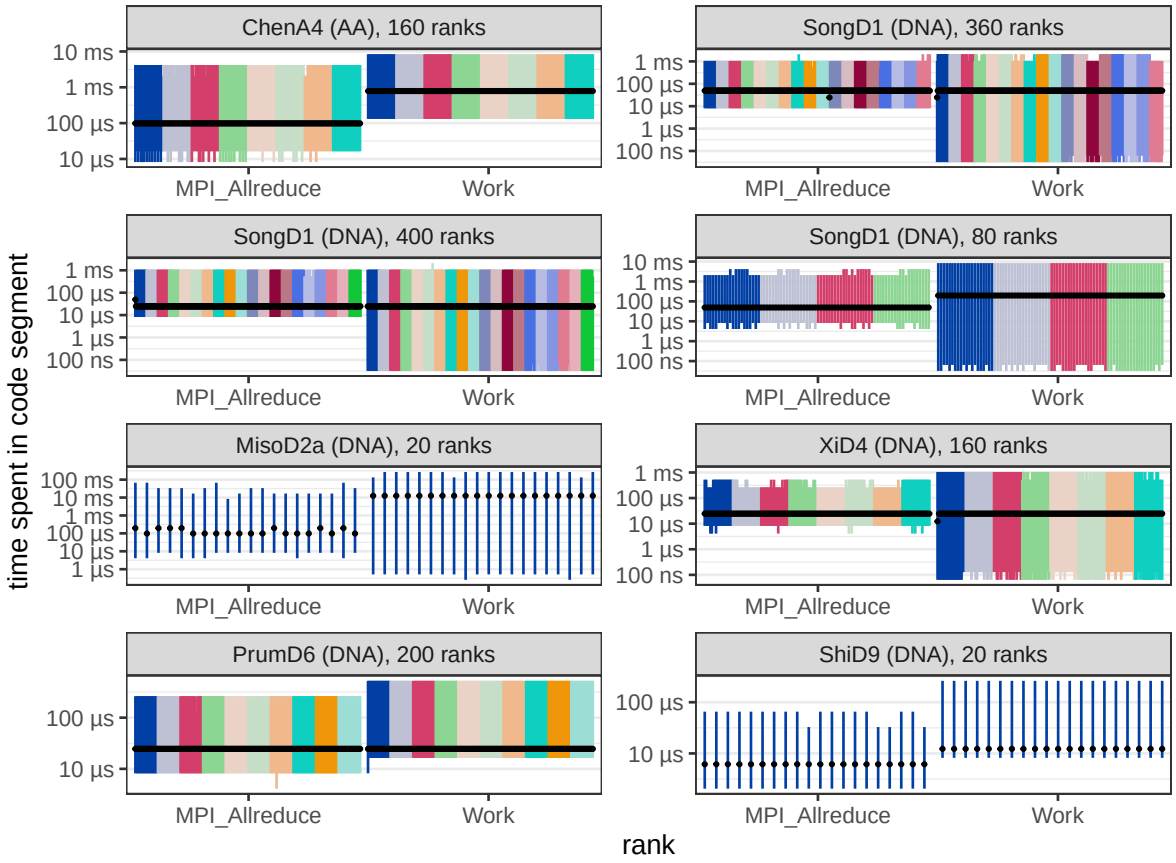


Figure 4.3.: Absolute time required for work and communication. Each bar depicts one rank. The colours group ranks by physical nodes. Each bar depicts the range between the 0.01 and 0.99 quantile of the time required on this rank. Black dots indicate the median. We bin the values into exponentially growing bins ([1 to 2) ns, [2 to 4) ns, [4 to 8) ns, ...). There are 20 ranks running on each node (one per physical CPU core).

the variance of the time taken performing work is greater than the variance of time required to communicate via `MPI_Allreduce` calls. The amount of work between two `MPI_Allreduce` calls varies. This is expected. RAxML-ng's algorithm is complex and has different phases (see Section 2.2.2). Depending on where in the algorithm we currently are, we execute different parts of the code during this work package. This can be anything between a likelihood evaluation on the current tree topology (see Section 2.2.1) and executing a SPR move (see Section 2.2.2).

The first rank per node sometimes requires the least time to finish a package. This is expected. We spend 85 to 98 % of the total runtime evaluating log-likelihood scores [2]. We do this, by computing the log-likelihood score of all sites independently and then computing their sum – first locally, then across all ranks (see Section 3.2 and Section 3.4). Consequently, the amount of work a rank has to perform to compute its local likelihood-score is linear to its

number of sites. The load-balancer assigns the least number of sites to the first processor on each rank. It thus has to perform the least work.

All ranks require about the same time to process their largest work packages in more than 99 % of cases. We cannot use Figure 4.3 to argue about small work packages or if any rank requires more time than other ranks to process the same work package. To investigate this, we have to measure the relative difference between the times required by different ranks. We do this in the next Section.

4.5.2. Relative Differences of Time Required for Work and Communication

We measure how much the time required to complete the same work or communication package differs between ranks. We measure the absolute differences ($t_{rank} - t_{fastest}$; see Appendix A.2.1) and the relative differences ($t_{rank}/t_{average}$) of the time required to process work packages and communication packages between ranks. To ascertain the time required by the fastest rank and the time required on average, we conduct one additional `MPI_Allreduce` call after each work package and its associated `MPI_Allreduce` operation. We do not measure the time required for this operation. In this Section, we describe the measurements of the relative differences, which we call Package-Specific Slowdon (PSS) for simplicity.

The time required for a work package varies by multiple orders of magnitude (see Section 4.5.1). We therefore investigate the PSS between each rank and the average rank (see Figure 4.4). That is, each rank computes $t_{rank}/t_{average}$ for each work or communication package. For example, a value of 1.1 indicates, that a rank requires 10 % more time to process the current package than the average over all ranks. We chose to compare against the average instead of against the fastest rank, as there are outliers when looking at the minimum time (see Appendix A.2.1). A bar ranges from the 0.05 to the 0.95 quantile of the PSS distribution of this node. Black dots indicate the median PSS (see Figure 4.2.c).

If a rank requires less time to finish a work or communication package than the average rank, the other ranks do not have to wait for it. This does therefore not increase the overall runtime. If a rank requires more time to finish than the average, other ranks have to wait for it at the next `MPI_Allreduce` call. We consequently want to avoid this situation. In all our measurements, there is at least one work package for which at least one rank requires *more* than 11 times as much time than the average rank. For all but one run[5], there is also at least one work package for which at least one rank requires at least 11 times *less* time than the average rank. We use binned histograms to store the PSS. We choose 11 times faster/slower as the largest/the smallest bin. The outliers might thus lie even farther out. We analyse the impact of these outliers on the total runtime in Section 4.5.3 and Appendix A.2.2.

Across all runs, no rank has a work-PSS of more than 2.75 on more than 5 % of packages. In half of the runs, the worst 0.95 quantile work-PSS across all ranks was less or equal to 1.25. In each run, the 0.95 quantile work-PSS was at least 1.15. Therefore, in each run, on at least one

[5]For this run, there is at least one work-package for which at least one rank requires 7 times less time than the average rank.

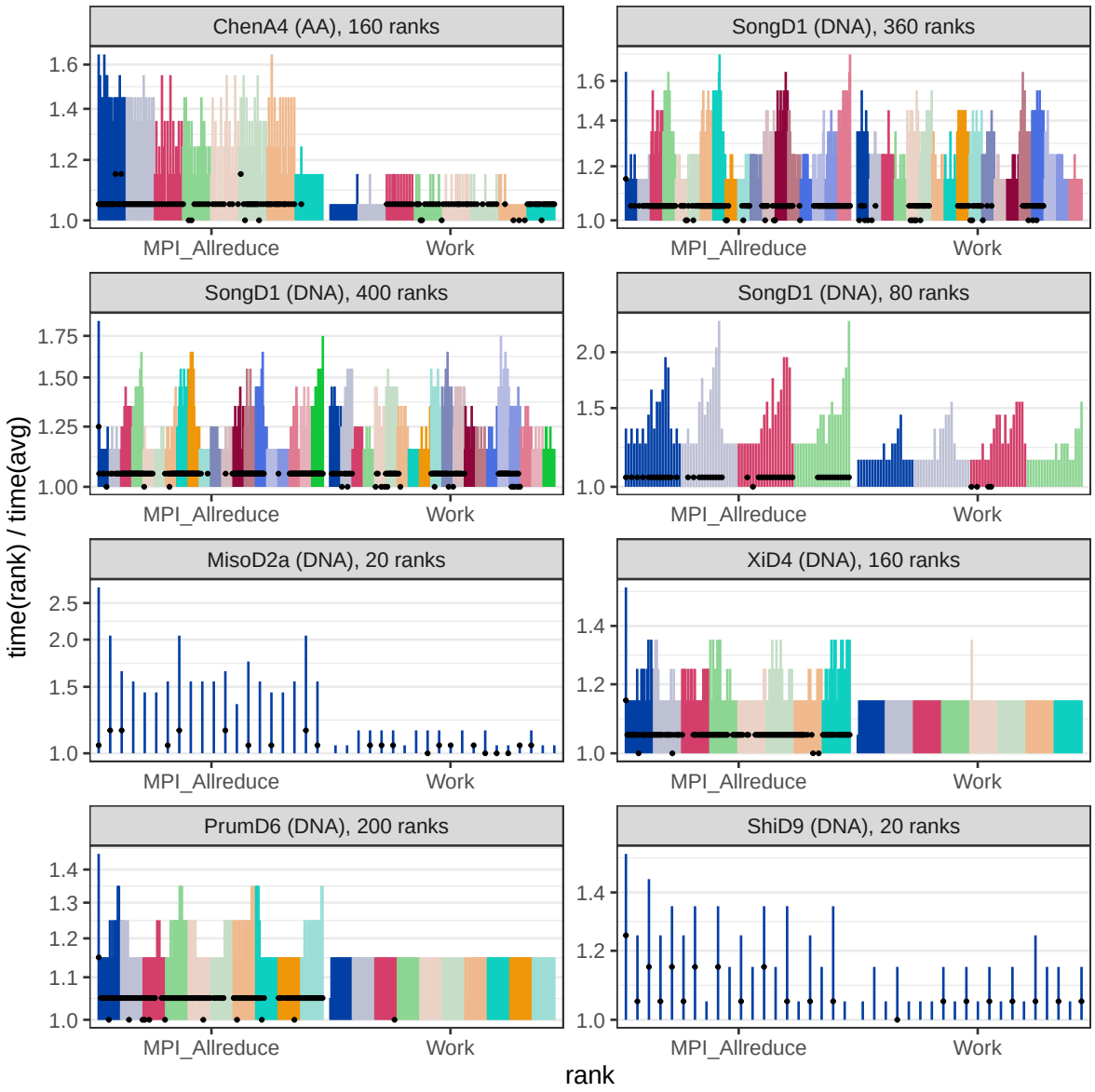


Figure 4.4.: Relative differences of the time required for work and communication packages (Package-Specific Slowdon (PSS)). That is, each rank computes $t_{rank}/t_{average}$ for each work or communication package. Each bar depicts the distribution of the PSSs of one rank. The colours group together ranks on the same node. The bar ranges from the 0.05 to the 0.95 quantile of the PSS. Black dots indicate the median of the PSS. For example: A bar ranging up to 1.6 means, that this rank required 60 % more time than the average rank for at least 5 % of the work/communication packages. The y-axis is truncated below 1.

rank, at least 5 % of work packages required at least 15 % more time to proceed than on the average rank. This points to an imbalance in the work distribution. From Figure 4.4 we cannot

extract the impact of this imbalance on the total runtime. It could be, that the imbalance only exists for small work packages and that large work packages are more balanced. In the next Sections we look into how the overall work volume (sum of all work packages) is distributed.

Overall, the variance of the PSS is larger for communication packages than for work packages. `MPI_Allreduce` calls require up to an order of magnitude less time than work packages (see Section 4.5.1). A rank which finishes with its work package will enter the following `MPI_Allreduce` call and wait there for all other ranks to finish their work. Consequently, a small relative difference in the time required to complete a work package will cause a large relative difference in waiting time inside the following `MPI_Allreduce` call. This explains the greater variance of relative differences for `MPI_Allreduce` calls vs work packages.

4.5.3. Overall Work per Rank

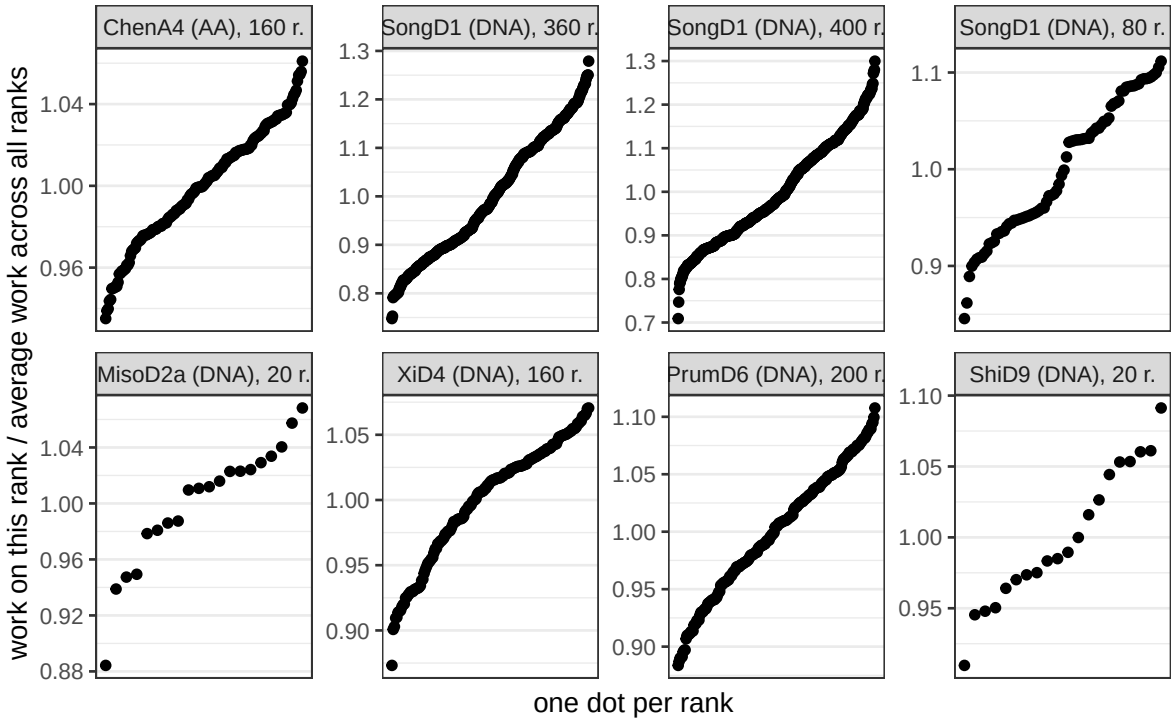


Figure 4.5.: Overall time spent working per rank, normalized by the average time across all ranks. We show the fraction of work on this rank divided by the average work across all ranks on the y-axis. We show on dot per rank. For example, a dot at $y = 1.04$ indicates, that a specific ranks requires 4 % more time to finish its work than the average rank does. "r.": ranks

We measure the sum of work performed on each rank. That is, we time each interval between two `MPI_Allreduce` operations and consider it as work package. We discard work-packages during which we write a checkpoint. We measure thousands of work packages per

second (see Appendix A.2.4), while a checkpoint is written only every few minutes to hours (see Figure 6.1). We therefore do not loose much information but prevent the time-intensive checkpoints from distorting our measurements. We then compute the total time each rank worked on the non-discarded work packages. We show this rank specific total work-time divided by the average work-time in Figure 4.5. For example, a dot at $y = 1.04$ indicates, that a specific ranks requires 4 % more time to finish its work than the average rank does.

For all runs, the maximum imbalance of overall work is below 30 %. That is, the slowest rank requires no more than 30 % more time to finish all their work packages than the average rank. For six of the eight runs, the imbalance of work is below 15 % and for half of the runs it is below 10 %. This shows, that there is an imbalance in the distribution of work between the ranks. In Section 4.5.5, we investigate the cause for this imbalance.

4.5.4. Which Ranks are the Slowest?

In Section 4.5.3, we find an imbalance in the distribution of work across the ranks of 15 to 30 %. The slowest rank requires 15 to 30 % more time to process all its work packages than the average rank does. This does not answer the question if the same ranks are the slowest ones for each work package. One rank could require the most time for every work package. It could also be, that while many ranks require a large amount of time for some work package, only some ranks are slower *on average*.

We thus count how often each rank requires the most time for processing a work package. We show this data in a Figure 4.6, with the rank count on the x-axis and the fraction of time a rank was the slowest to process a work package on the y-axis. For example, a dot at 0.03 indicates that a specific ranks requires the most time for 3 % of work-packages. We also look at the fraction of time each rank spends working compared to the time it spends inside an `MPI_Allreduce` call (see Appendix A.2.3).

We measure the time required for processing each work package on each rank. This measurement uses the local clock. The rank which requires the most time to conduct its work is not necessarily the last one to arrive at the barrier of the following `MPI_Allreduce`. This is, because the ranks did not exit the previous barrier synchronously. The time required for an `MPI_Allreduce` is up to an order of magnitude less than the time required to process a work package (see Section 4.5.1). We want to argue about the imbalance of work across the ranks and therefore neglect this difference.

For three of the eight measurements, a single rank is the slowest rank on at least twice as many work packages than any other rank. For example in the run on `ShiD9` with 20 ranks on a single node (bottom-right), one rank was the slowest rank for 30 % of work packages. All other ranks are the slowest rank in less than 10 % of the work packages. On five out of eight runs, at least one rank was the slowest rank for more than 5 % of the time. Taking into account the previous Sections, we can conclude, that there is a systematic imbalance. The same ranks require the most time to process a work package for a substantial fraction of all work packages and the sum of work is unevenly distributed across the ranks.

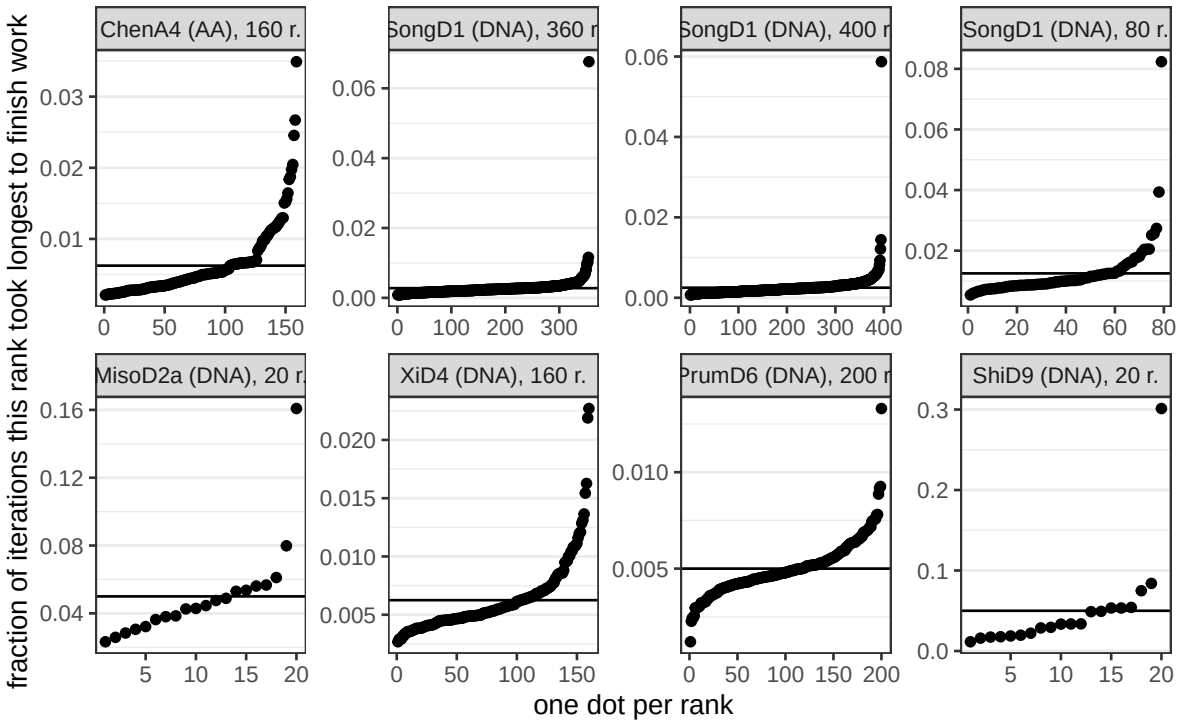


Figure 4.6.: How often a rank requires the most time to process a work package. The plot shows the fraction of work packages for which a rank is the slowest. For example, a dot at 0.03 indicates that a specific ranks requires the most time for 3 % of work-packages. The black vertical bars show the fraction a rank is expected to be the slowest one, that is, $\frac{1}{\text{nRanks}}$. "r.": ranks

4.5.5. Site-Repeats and Imbalance of Work

In Section 4.5.3 we showed, that there is an imbalance of work of up to 30 % in our measurements. In Section 4.5.4 we showed, that for some runs, a single rank requires the most time to process the current work package for 30 % of all work packages.

The question now is, what causes this imbalance? We hypothesize, that the site-repeat feature (see Section 3.3) is causing this. Remember that site-repeats are sites which are identical in different subtrees. If we consider site-repeats, we can omit redundant computations. As transferring the results of this computations over the network requires too much time, we can only consider site-repeats on the same rank. The current load balancer does not account for site-repeats. If different ranks have different amount of site-repeats, they can omit a different amount of computations which causes the work to be unevenly distributed.

To underpin our hypothesis with data, we compare the imbalance of work between runs with site-repeats turned on and off. We keep the dataset and number of ranks constant. We also look at how the time working to total runtime ratio changes when enabling site-repeats (see Appendix A.2.3).

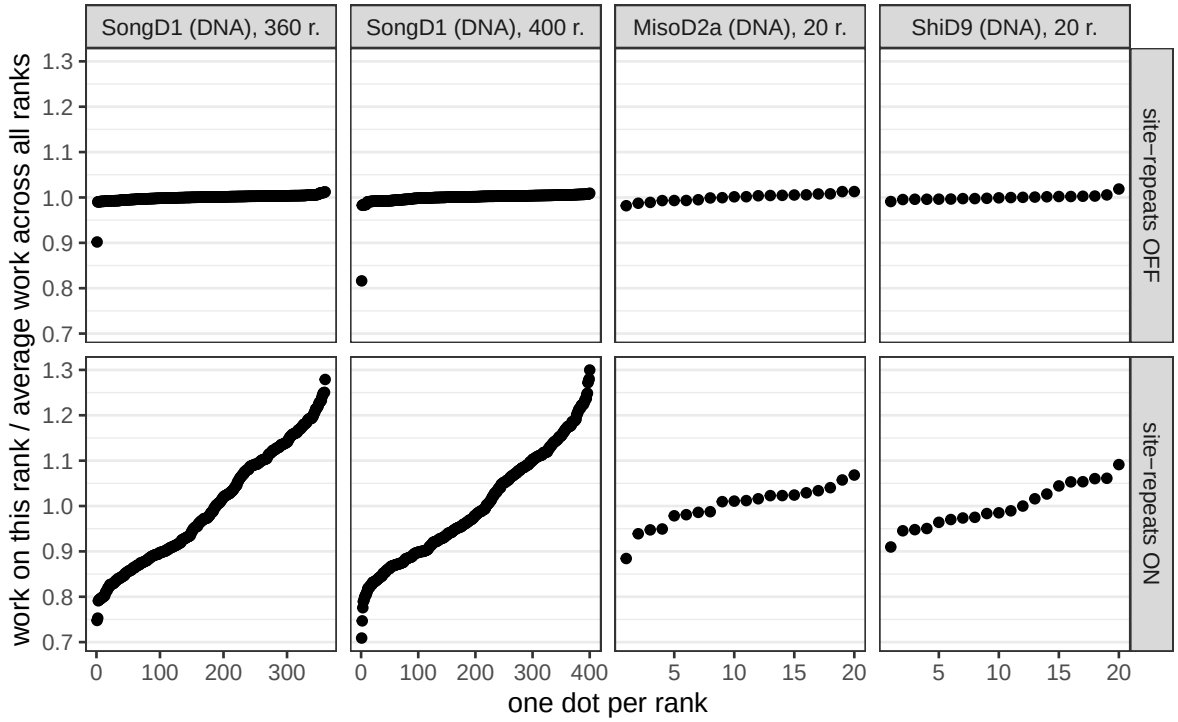


Figure 4.7.: Distribution of work with site-repeats `ON` and `OFF`. Each rank measures the total time it is working and normalizes this measurement with the average time a rank is working. With the site-repeats feature turned `OFF` (vs. `ON`), the variance is significantly smaller (one-sided F-test; $p < 0.002$ for ShiD9, $p < 10^{15}$ otherwise). "r.": ranks

Table 4.2.: Runtimes of RAxML-ng with the site-repeat based omission of redundant calculations `ON` vs. `OFF`. We use all 20 CPU cores on each node.

dataset	nodes	ranks	runtime SR off [s]	runtime SR on [s]	speedup
SongD1	18	360	5,633	1,352	4.17
MisoD2a	1	20	65,795	46,155	1.43
ShiD9	1	20	48,450	20,911	2.32
XiD4	8	160	7,718	5,582	1.38

Disabling site-repeats decreases the imbalance (variance) of work significantly (one-sided F-test; $p < 0.002$ for ShiD9, $p < 10^{15}$ otherwise; see Figure 4.7). Disabling site-repeats in production runs is nonetheless not a good idea. By disabling the site-repeats feature, we increase the runtime of RAxML-ng by up to 417 % in our experiments (see Table 4.2). Using the site-repeats feature thus speeds up the computation considerably, but also causes imbalance of work.

Writing a load balancer which takes into account the work saved by site-repeats is not trivial. Which sites are repeats of one another depends on the current subtree. The same site

can be a site repeat of multiple other sites, one (or multiple) for each subtree. We propose and implement a site-repeats aware load balancer by reducing the problem to judicious hypergraph partitioning in another publication [8].

Part III.

Failure Mitigation

5. Fault-Tolerant MPI

Failing hardware is projected to be one of the main challenges in future exascale systems [97]. In fact, it is reasonable to expect that a hardware failure will occur in exascale-systems every 30 to 60 min [16, 25, 99]. HPC systems may fail for the following reasons: Core hangs, kernel panic, file system bugs, file server failures, corrupted memory or interconnect, network outages, and air conditioning or power halts [46, 74]. Metrics to describe the resilience of hardware are the Mean Time Between Failure (MTBF) for repairable components and the Mean Time To Failure (MTTF) for non-repairable components. Both describe the average time one can expect a system to function after repair or replacement [74]. For the sake of simplicity in the following, we will subsume MTBF and MTTF under the term MTTF and assume negligible repair and replacement time. The current state of most MPI software is such that a failure on any rank will result in program termination. Regarding the frequency of failure we can therefore look at the set of ranks as a whole in terms of a number of serially connected single systems which fails as a whole if one component fails. The MTTF of an MPI program running on PEs $n_1, n_2, \ldots, n_j$ with independent failure probabilities is therefore equal to

$$\mathrm{mttf}(n_1, n_2, \ldots, n_j) = \left(\sum_{n_j} \frac{1}{\mathrm{mtbf}(\mathrm{n_j})} \right)^{-1}$$

As the number of cores that scientific software runs on increases, the MTTF decreases rapidly. Gupta *et al.* [46] reported the MTTF of four systems in the petaflops range containing up to 18,688 nodes (see Table 5.1). Currently, most compute jobs only use a part of these petascale systems which explains why current software are not constantly aborted because of rank failures. In the not so distant future, on the by then commonly available exascale systems, scientific software will run on tens of thousands of cores or more, therefore experiencing a core failure every few hours [16, 25, 99]. We can therefore no longer ignore the possibility of compute node or network failures, and are in need for failure mitigating software.

Table 5.1.: MTTF of petascale systems as reported by Gupta *et al.* [46]

System	Nodes	Cores	MTTF
Jaguar XT4 (quad-core AMD Opteron)	7,832	31,328	36.91 h
Jaguar XT5 (four socket dual-core AMD Opteron)	18,688	149,504	22.67 h
Jaguar XT6 (2 socket 16-core AMD Opteron-6274)	18,688	298,592	8.93 h
Titan XK7 (16-core AMD Opteron-6274 + K20x Nvidia GPU)	18,688	560,640	14.51 h

5.1. Techniques for Fault Tolerant MPI Programs

Despite the limited support for mitigation of hardware failures by MPI, some research on handling faults already exists. The three main techniques used to make programs failure tolerant are: Algorithm Based Fault Tolerance, restarting failed sub-jobs, and checkpointing/restart. Algorithm Based Fault Tolerance is used particularly in numeric applications [14, 115] but has the inherent problem that the algorithm in question has to be extensible to include redundancy. For example in matrix multiplication the algorithm can add more rows to the matrix. It can chose these additional rows such that they redundantly encode the contents of the original matrix. In case of failure, the algorithm can use this added redundancy to restore the lost data. We believe it is unlikely to be possible to extend RAxML-ng in this way based on over 15 years of experience in RAxML-ng development. Using restarts of failed sub-jobs as failure mitigation strategy is feasible in case the program at hand can be split up into separate small work packages which can easily be redistributed between nodes and managed by a (possibly distributed) work queue. MapReduce frameworks, for instance, implement this approach [75]. RAxML-ng, however, is an iterative optimizer which we cannot easily split up this way without a substantial rewrite, if at all. For checkpointing and restarting the program has to save its state to disk or memory at regular intervals. From these saved checkpoints the user or job scheduler can restart the program after a failure.

Checkpoint/restart approaches are further classified into system-level and application-level approaches. System-level approaches have the advantage of being (nearly) transparent to the application the programmer intends to checkpoint. This enables fault tolerance with minimal development overhead [47, 90]. But transparent checkpoint/restart systems are not aware of which parts of the allocated memory are relevant and which parts the program can recompute easily. In RAxML-ng, saving Conditional Likelihood Vectors (CLVs, which store cached intermediate results of likelihood computations) may take up to tens of gigabytes of memory per node, resulting in terabytes of memory for large runs [55, 77] we would have to checkpoint only to invalidate them after restoration. Application-level checkpointing is already implemented in RAxML-ng (see Section 6.1). We decided to increase the frequency of and granularity of checkpointing. The changes to the existing scheme are described in Section 6.2.

RAxML-ng can easily recompute a large portion of its allocated memory, for example the CLVs. Only the model parameters, branch lengths, and tree topology have to be stored at checkpoints. In common use cases, these make up a few megabytes only. We can therefore afford to save a full checkpoint to each rank's memory every time we perform one. This is called diskless checkpointing and has the advantage of being faster than writing checkpoints to disk [85].

In coordinated checkpointing, all ranks of the program create their checkpoints at the same time. This comes at the cost of an additional synchronisation point. Gavaskar and Subbarao recommend coordinated checkpointing for high-bandwidth, low-latency interconnections as they are common in modern HPC systems [40]. Because of this and the fact, that the ranks in

RAxML-ng are synchronizing thousands of times per second anyway (see Appendix A.2.4) we choose to conduct coordinated instead of uncoordinated checkpointing.

This leaves the question if we want to have spare cores available to replace failed nodes or shrink the number of nodes the job runs on upon failure. For example Teranishi and Heroux describe a framework for recovering from failures relying on available replacement processors [111]. But making sufficient replacement processors available constitutes a waste of resources in case there is no failure. Ashraf *et al.* look at the performance implications of replacing failed nodes versus shrinking the set of worker nodes. For their application they draw the conclusion that shrinking represents a viable alternative to replacement. The time required did increase to a smaller degree when looking at shrinking vs replacement as it did when looking at the number of failed nodes [7]. We therefore choose to not make spare nodes available and instead redistribute the calculations to the remaining nodes upon failure.

5.2. The new MPI Standard and User Level Failure Mitigation

The upcoming MPI standard 4.0 will have support for mechanisms allowing developers to mitigate failures of ranks or network components. Currently, there are two actively developed MPI implementations which already support failure mitigation: MPI Chameleon (MPICH) [44] starting with version 3.1 and User Level Failure Mitigation (ULFM) [11]. We chose ULFM as the MPI implementation to develop a failure-mitigating version of RAxML-ng because the authors are also working on the standardization of MPI 4.0 and we therefore hope to be as forward compatible as possible.

Researchers have used ULFM in scientific software before. For example Ali *et al.* implemented numeric linear equation and partial equation solvers which are failure tolerant [3]. Obersteiner *et al.* extended a plasma simulation [80], Laguna *et al.* a molecular dynamics simulation [72], and Engelmann and Geist a Fast Fourier Transformation [29] that gracefully handle hardware faults. Kohl *et al.* [65] implemented a checkpoint-recovery system for a simulation in the material sciences. After a failure, the system assigns the work of the failed PEs to a single PE. The load-distribution algorithm [95] then recalculates the data distribution with the reduced number of PEs. Next, the PEs exchange the data residing on the wrong (i.e. overloaded) PE over the network using point-to-point communication. Their algorithm does not handle redistribution of static data but only of data which changes over the runtime of the algorithm.

ULFM reports failures by returning `MPI_ERR_PROC_FAILED` on at least one rank which participated in the failed communication. This rank then has to use `MPI_Comm_revoke` to propagate the failure notification to the other ranks. The next time a rank calls an MPI operation it will be notified that another rank revoked the communicator. Different ranks can therefore be in different parts of the code when they detect the failure. Next, all surviving ranks call `MPI_Comm_shrink` collectively, creating a new communicator with the failed ranks excluded [76].

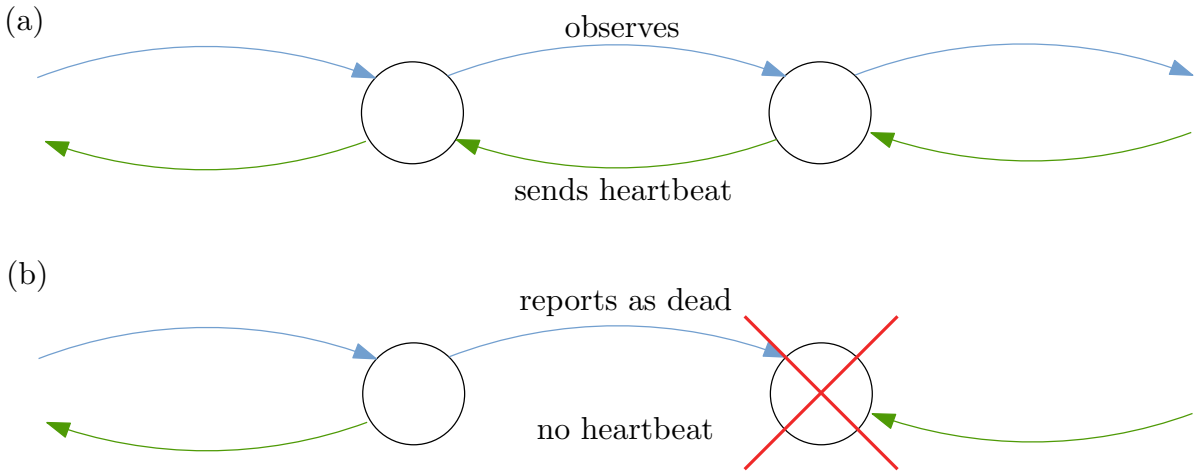


Figure 5.1.: Simplified principle of heartbeat-based failure detection. **(a)** during normal operation, each rank sends a heartbeat signal to its observer at regular intervals. **(b)** If a node fails, it will no longer send heartbeat signals. After missing three heartbeat signals in a row, its observer will report it as being dead.

ULFM detects hardware failures by a variety of detection mechanisms, depending, for example on the kind of network interconnection available. One of the basic mechanisms is that of regular heartbeat signals. In its default configuration, each rank sends a heartbeat signal every 100 ms to the rank responsible for observing it. If the rank misses three heartbeat signals in a row, its observer reports its failure (Figure 5.1). The MPI standard only mandates progress during MPI calls. For ULFM this means that for the failure detection to work at least one thread per rank has to enter an MPI function at regular intervals. If this is not the case, a rank cannot guarantee not to miss multiple consecutive heartbeat intervals which would cause it to be falsely reported it as being dead. We ran into this behaviour when testing ULFM with the unmodified (non failure-mitigating) version of RAxML-ng in preliminary tests. All three runs aborted because of false-positive failure reports. After consultation with the authors of ULFM, we adjusted some of ULFM's runtime settings as follows: Increase the heartbeat interval from 100 ms to 300 ms, the heartbeat timeout from 300 ms to 1 s, and enable a separate heartbeat thread on each rank. Separate heartbeat threads ensure MPI progress at all times. This is because they are responsible only for sending heartbeats and therefore can enter the MPI runtime at all times without having to wait for the program to call an MPI function. In response to our discussion on the mailing list, the ULFM team published a tutorial on this topic on the ULFM website.[1] After incorporating these changes into our configuration, we observed far fewer false-positive failure reports on ForHLR II (see Section 4.2). In theory, these changes come at the cost of performance. ULFM will require more time to detect failures

[1] `https://fault-tolerance.org/2020/01/21/spurious-errors-lack-of-mpi-progress-and-failure-detection/`

because of the increased heartbeat timeout. The latency will also increase because multiple threads are accessing MPI.

We measure the effect of the heartbeat timeout and heartbeat thread settings on the time ULFM required to recover from failures. We did this on the ForHLR II system by repeatedly simulating failures (Section 5.3) and measuring the time until a new communicator is created. (Figure 5.2). First, we measure the time required for failure recovery in the default configuration (300 ms no heartbeat thread) on 4 nodes. Enabling the heartbeat thread decreases detection time from 8 s (median) to 900 ms (median). As the heartbeat thread also decreases the probability for false-positive failure reports, we therefore decided to keep this setting. Next, we investigate the impact of setting the timeout to 1,000 ms on the failure detection speed. Contrary to our expectations, increasing the heartbeat interval does not change the result significantly (effect below standard deviation). Additionally, we want to look into the question if slow detection of failures is caused by the computations keeping the CPU cores crammed. We issue three runs with different heartbeat thread and timeout settings in which we do not use all available cores for computations, leaving them free for the ULFM runtime. This does not change the time ULFM required for failure recovery. We also perform one experiment with 400 ranks to get a feeling on how the failure detection scales. With the heartbeat thread enabled and a 300 ms timeout, 11.4 % of the recoveries require more than 2 s and 8.9 % of the recoveries require more than 90 s. These results are probably highly dependant on the HPC system used and should not be generalized. Laguna *et al.* reported that ULFM required 11 s to recover when using 260 ranks on another system [72].

To evaluate the real world impact of using a fault tolerant MPI implementation, we also compared the runtimes of RAxML-ng using ULFM with heartbeat thread enabled and disabled vs OpenMPI v4.0 as baseline on three different datasets and PE counts (Table 5.2). We chose OpenMPI v4.0 as reference because ULFM v4.0.2u1 is based upon OpenMPI v4.0 . The additional time required for using ULFM ranges between −0.6 and 8.6 % of the reference runtime. The additional slowdown induced by a separate heartbeat thread ranges between −0.7 and 2.2 % of reference runtime. The run using ULFM being faster than the run using OpenMPI might be due to measurement fluctuations, but we did not tested this hypothesis.

Table 5.2.: Performance impact of ULFM. We show the runtimes of unmodified RAxML-ng using OpenMPI v4.0 and ULFM v4.0.2u1 with heartbeat thread (hbt, a thread dedicated to sending heartbeat signals) turned ON and OFF. The slowdown is given relative to OpenMPI v4.0.

				slowdown [%]	
dataset	**nodes**	**ranks**	**OpenMPI [s]**	**ULFM hbt: ON**	**ULFM hbt: OFF**
ChenA4	8	160	5,582	0.1	−0.6
SongD1	18	360	1,437	1.9	2.5
ShiD9	1	20	20,911	6.4	8.6

We also observe that ULFM sometimes reports a single rank failure but `MPI_Comm_shrink` returns a communicator which differs among the ranks. Multiple ranks report their rank id as

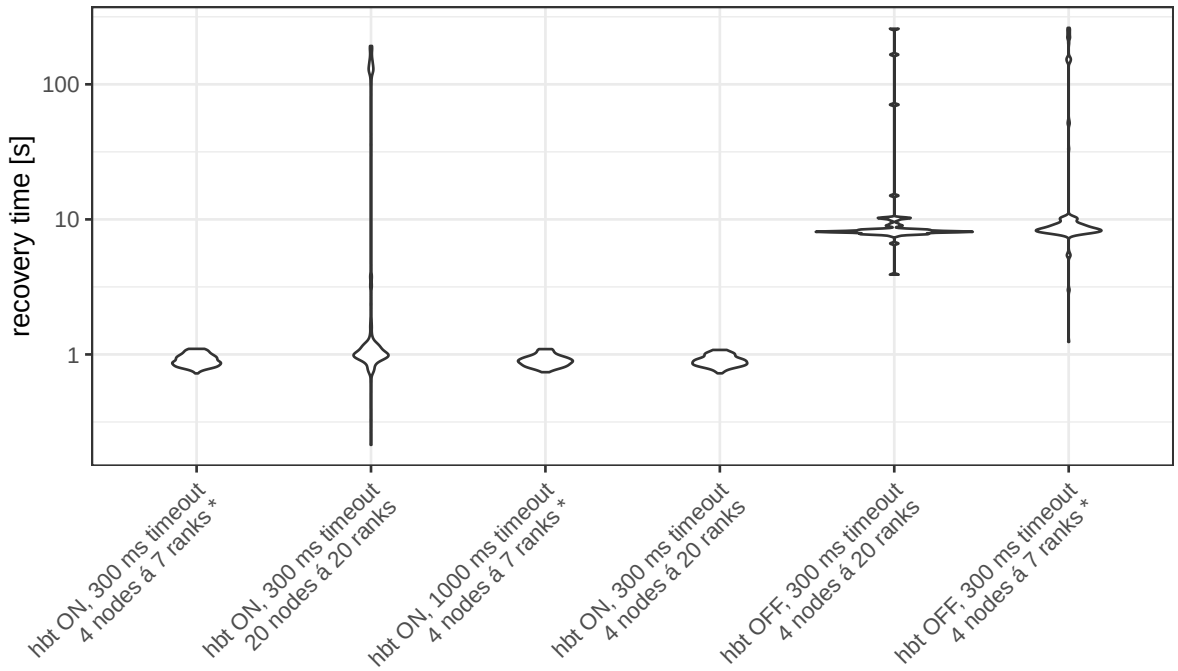


Figure 5.2.: Violin plot of the time required by ULFM to detect and recover from a node failure. This includes the time for for all ranks to agree on which nodes have failed (done by ULFM) and creating the new communicator. We measurd two heartbeat timeouts: 300 ms (default, false-positives) and 1,000 ms (no false-positives). Measurements marked with a * have additional nodes allocated which our code does not use, leaving them free for ULFM. hbt (heartbeat thread) indicates whether we enabled a thread responsible solely for sending heartbeat signals. Each measurement is performed at least 49 times.

0 and a world size of 1. We reported this behaviour on the ULFM mailing list and the authors of ULFM reproduced and confirmed the bug [13]. There is no patched version available yet.[2] For this reason we use OpenMPI v4.0 as default MPI implementation and simulated failures as described in the following section for some experiments.

5.3. Simulating Failures

We can simulate core failures in numerous ways without root access to the HPC machines. When using ULFM the heartbeat detection mechanism yields a straightforward approach. When not using a heartbeat thread (see Section 5.2) it suffices to put the program into a long sleep to simulate a failure. When using a heartbeat thread, sending the signal `SIGKILL` to the

[2] As of June 30, 2020.

rank's process will simulate a failure. The program cannot catch, block ore ignore `SIGKILL`. It can therefore not perform any cleanup operation [61]. Other possible signals include `SIGSEGV`, `SIGILL`, `SIGFPE`, `SIGBUS`, `SIGXFSZ`, `SIGPWR`, and `SIGXCPU`. None of these signals allow the receiving process to perform a cleanup operation. We tested all of these methods and ULFM detected all of them as rank failures with no noticeable difference, that is, the next MPI operation will fail with `MPI_ERROR_PROC_FAILED`. We were able to revoke the communicator using `MPI_Comm_revoke` and the new communicator we build using ULFM's `MPI_Comm_shrink` did not contain this "failed" node. We choose to simulate failures in experiments with ULFM by killing a process via signalling `SIGKILL` either via invocation of `kill -SIGKILL` [59] or via self-signalling using `raise(SIGKILL)` [60].

When using OpenMPI to avoid running into false-positive failure reports and inconsistent communicators (see Section 5.2) we exploit RAxML-ng's encapsulation of all MPI calls inside the `ParallelContext` class to simulate failures. A static method of this class is set as the parallel allreduce callback in the `C` part of RAxML-ng. This way, all parallel communication goes through `ParallelContext`. If we intend to simulate a failure during an MPI call, we split the communicator into a set of surviving and a set of of failed nodes using `MPI_Comm_split`. The split-off nodes in the failed group then terminate gracefully and the set of surviving nodes continue restoring the search state as if a real failure had occurred (see Section 6.3). Additionally, we can simulate a failure without loosing nodes. In this case, we do not split off nodes but rather reassign each node a new rank id and restore the search state as if a real failure would have occurred.

6. Implementing a Failure-Mitigating RAxML-ng Tree Search

The amount of available biological sequence data is increasing with enormous speed [64]. The need for phylogenetic inference on large MSAs is therefore growing. These phylogenetic inferences require more and more computing power. Single core performance does no longer increase according to Moore's Law [114]. We therefore need to apply horizontal scaling, that is, add more CPUs. Increasing the number of CPUs will decrease the MTTF of the system as a whole (see Chapter 5). Consequently, we need to gracefully detect and handle rank failures.

6.1. Current State - Checkpointing and Restart

Even before we implemented the modifications described here, RAxML-ng already supported saving the current tree search state to disk. In case of failure, the user can restart the program from the last checkpoint. Checkpoints, however, could only be issued at certain steps during the optimization procedure. This means, that, depending on the dataset, several hours can pass between two checkpoints. Thus, if a failure occurs in a large parallel run, possibly hundreds or thousands of CPU hours are lost.

The search state of RAxML-ng consists of the model parameters, the tree topology, and the branch lengths of the currently best know tree, that is, the tree with the currently highest likelihood score. The model parameters include the nucleotide frequencies, the transition matrices, and the heterogeneity rate parameters. The tree topology is the same on all ranks at all times. It is, however, not saved in memory as is. We need to reconstruct the currently best known tree from the tree topology of the currently evaluated tree and a sequence of roll-back SPR moves (see Section 2.2.2.1 and Figure 6.2). The model parameters of a partition are only stored at those ranks that have at least one MSA column of that partition assigned to them. It is therefore possible that the model parameters of a partition are only saved at one single rank. These can hence be lost if the rank fails.

6.2. Mini-Checkpointing

We want to support the mitigation of the failure of any set of ranks. For this, we need to store the model parameters of each partition at all ranks. This also represents the simplest solution, and we want to avoid pre-mature optimization before profiling. Each time an optimization

procedure updates the model parameters, we need to broadcast them to and update them on all other ranks. The model parameters can be changed by the following subroutines: Substitution rates optimization, base frequency optimization, alpha parameter (Γ-model) optimization, proportion of invariant sites (`+I` model) optimization, rates and weights optimization, branch length optimization, and branch length scaler normalization and optimization. An illustration is given in Figure 6.1. To differentiate this redistribution of model parameters from the regular checkpointing to disk, we call it "mini-checkpointing". RAxML-ng stores a copy of the latest mini-checkpoint in the main memory of each rank.

If the program detects a failure during one of the above procedures or an SPR-round (see Section 2.2.2.1), it will restart the computation from the last mini-checkpoint. If the program detects a rank failure during mini-checkpointing, it needs to restart from the preceding optimization. This is because we cannot guarantee, that the current value of each model parameter is still available on a surviving rank.

Regular checkpoints write the model parameters, the tree topology, and the branch lengths to disk. The tree topology and the branch lengths are consistent on all ranks at all times. We therefore do not need to collect them prior to creating a checkpoint. We perform mini-checkpointing each time an optimization procedure updates the model parameters (see Figure 6.1). The mini-checkpoints are therefore also consistent on all ranks when we want to write a checkpoint to disk. This alleviates the need for collecting model parameters during checkpoint creation. Creating regular checkpoints is therefore a local operation that does not require network communication. This means that while writing checkpoints to disk, ULFM will not report failures. Thus, we do not need to handle rank failures. The checkpoint procedure will only fail if the master rank fails while writing to disk. In this case, the former checkpoint will still be valid. We can use it to restart the search. If any other rank fails while the master rank writes the checkpoint, we will detect this failure at the beginning of the next optimization round. We can then restart the computation from the checkpoint that was just written.

6.2.1. Problem Statement

To enable mitigation of rank failure during a tree search, the search state has to be available consistently at each rank at the time of failure. The search state consists of the model parameters, the tree topology, and the branch lengths. We define "consistent" in this context as a combination of values which were current at the same time in the past. If the program detects a rank failure it needs to restore its search state from the last (mini-)checkpoint. Next, it has to restart the search on all surviving ranks.

6.2.2. Algorithm

A major part of this Master's Thesis was to adapt RAxML-ng such that it satisfies the description given in the problem statement above (see Section 6.2.1). For this, we need to extend the checkpointing strategy to keep all model parameters current at all ranks at all times (see

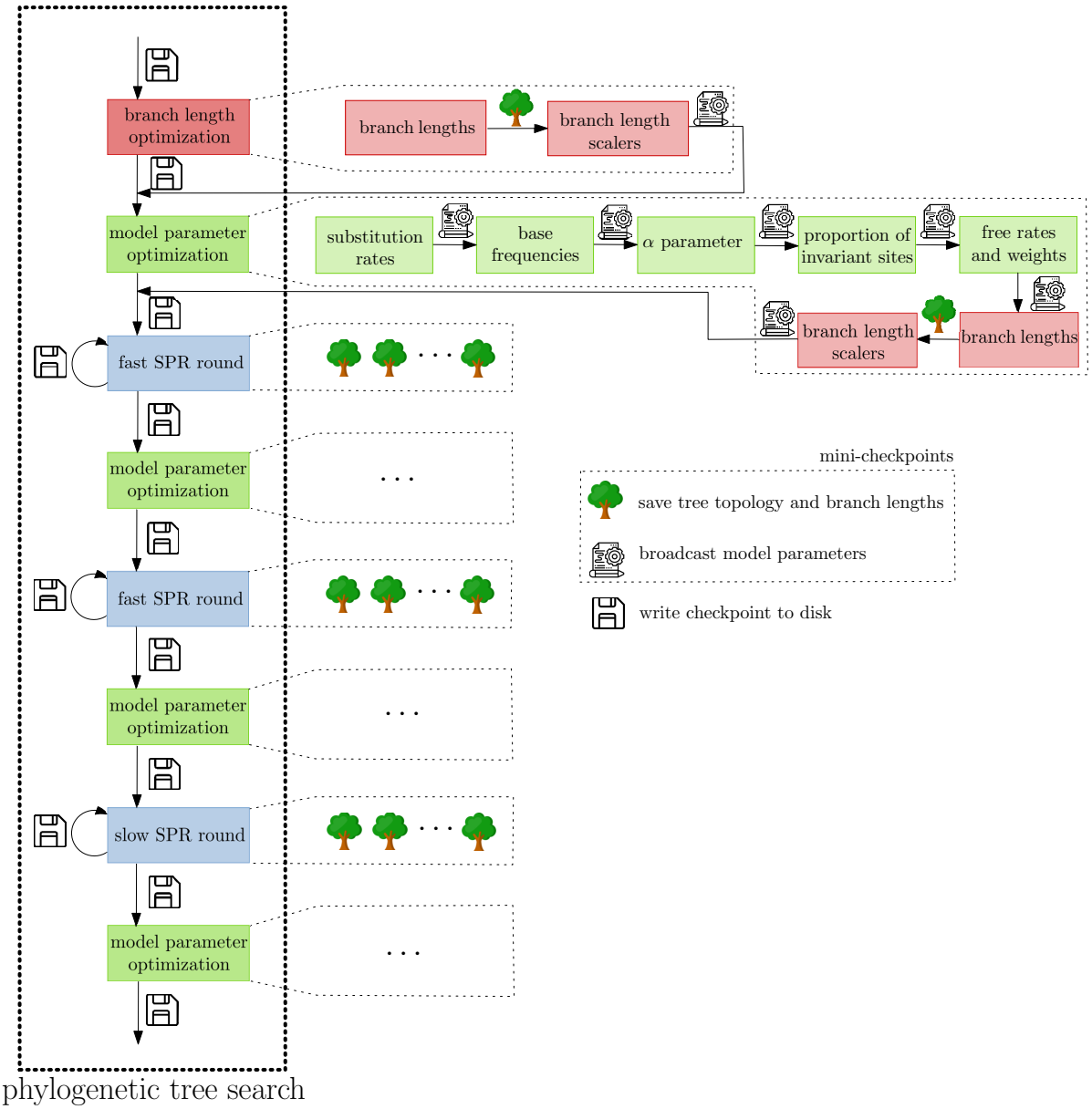


Figure 6.1.: Frequency of checkpointing. On the left, an overview of the search procedure is given. RAxML-ng writes checkpoints to disk before each step of the optimization procedure and when optimization is completed. These are the regular checkpoints which are already implemented. To have the up-to-date model parameters, the master rank has to collect them first. Depending on the dataset and number of ranks used, each of these phases can take multiple hours to complete. By introducing mini-checkpointing, we increase the frequency at which the model parameters are shared. The ranks now broadcast them after each sub-step, denoted by the respective model symbol. Additionally, the currently best-scoring tree is saved each time it is updated, that is after adjusting the branch lengths and during SPR rounds. Checkpoints are still written do disk, but do now not need to collect the model parameters, as they are already consistent on all ranks.

Section 6.2). We also increase the frequency at which we create an in-memory checkpoint of the topology of the currently best known tree. We use ULFM to detect rank failures and create a new MPI communicator containing only the surviving ranks in case of failure. RAxML-ng then redistributes the work to the surviving ranks, restores a valid program state, and restarts the tree search.

Mini-Checkpointing: Redistribution of Model Parameters

Each MSA partition has a set of model parameters associated with it (see Section 2.2.1). These model parameters comprise the transition matrix, the base frequencies, and in the scaled branch length scaling mode the branch length scalers. Each time an optimization procedure updates the model parameters of a partition, the program redistributes them to all ranks (see Figure 6.1). Each partition has one rank associated with it. This rank is responsible for sending this partition's model parameters to all other ranks. One rank might be responsible for multiple partition's model parameters, but each partition has only one rank which is responsible for broadcasting its models. To create a mini-checkpoint, each rank executes Algorithm 2, synchronizing at each broadcast. As a result, each rank has an up-to-date copy of each partition's model parameters. As long as at least one rank survives, RAxML-ng can thus resume the tree search.

Algorithm 2 Broadcast of Model Parameters

```
procedure BroadcastModelParameters
    for each rank do
        if rank is responsible for at least one model then
            Broadcast(all models this rank is responsible for)        ▷ temporary copy
        end if
    end for
    check for rank failure                                            ▷ using MPI_Comm_agree
    if no rank failure reported then
        Update working copy from temporary copy.
    else
        Rollback to previous mini-checkpoint.                         ▷ working copy unaltered
        Restart preceding optimization.
    end if
end procedure
```

Model parameters for common use cases are less than a few MiB in size. Thus, we expect the number of partitions m to have a negligible impact on the runtime of a single broadcast. It does, however, have an impact on the number of broadcasts that the program needs to execute. Each broadcast transmits at least one model. Therefore, we need to conduct a maximum of one broadcast per partition and the number of partitions is an upper bound for the number of broadcasts. A rank will broadcast all the model parameters it is responsible for in a single

broadcast. The number of ranks is hence another upper bound for the number of broadcasts. Let us assume there are p ranks and the time for a single broadcast is $T_{bcast}(p)$. The runtime of mini-checkpointing then scales with $\min(p, m) \cdot T_{bcast}(p)$.

Upon broadcasting, the ranks gather the received models in a temporary copy. If no rank failure occurs during broadcasting, the algorithm copies the temporary copies over to the working copies. This is a local operation. Thus, if a rank fails during this, all other ranks will still have the up-to-date models and can restart from them. Note that they will recognize the failure at the subsequent collective operation. If a rank fails during the broadcasting of model parameters, there is no guarantee that an up-to-date copy of all parameters is still available. We consequently need to repeat the preceding parameter optimization step. For this, we need to restore the data from the last mini-checkpoint, which is still valid. On success, the program rebroadcasts the updated model parameters.

Saving the currently best tree topology

All changes made to the tree topology happen at all ranks simultaneously (see Figure 6.2). Hence, we do not need to broadcast them. RAxML-ng does, however, not save the currently best tree topology in a trivial form. An SPR round modifies the tree topology, saving the moves needed to restore the best tree topology in a rollback list (see Figure 6.2).

During the evaluation of all possible moves we are at most one move away from the currently best tree. After we evaluated all moves once, we re-evaluate the moves that result in the 20 best-scoring topologies with full branch-length optimization. If a failure occurs during this, we would need to rollback multiple moves from both the rollback and the best-nodes list. To simplify recovery and avoid pre-mature optimization before profiling, we choose to copy the currently best-scoring tree to a separate rollback data structure each time it is updated. Note that, we do not rebroadcast the model parameters here; therefore this operation happens locally on each rank (see Figure 6.1).

6.2.3. Evaluation

We implement the redistribution of the model parameters. We evaluate the time required as a function of the number of models and ranks used. We use the same hardware configuration as described in Section 4.2. We measur the time for generating each mini-checkpoint separately and show the mean (dots) and standard deviation (error bars) in Figure 6.3. We expect the time required to broadcast models to increase only if the number of models *and* the number of ranks increase. This is, because the number of models and the number of ranks are both upper bounds for the number of broadcasts we need to send (see Section 6.2.2). If only the number of models *or* the number of ranks increases, we expect mini-checkpoints not to require substantially more time. If the number of models increases but the number of ranks stays small, the number of ranks will limit the number of broadcasts required. In the worst case, each rank will send one broadcast. If the number of ranks increases but the number of

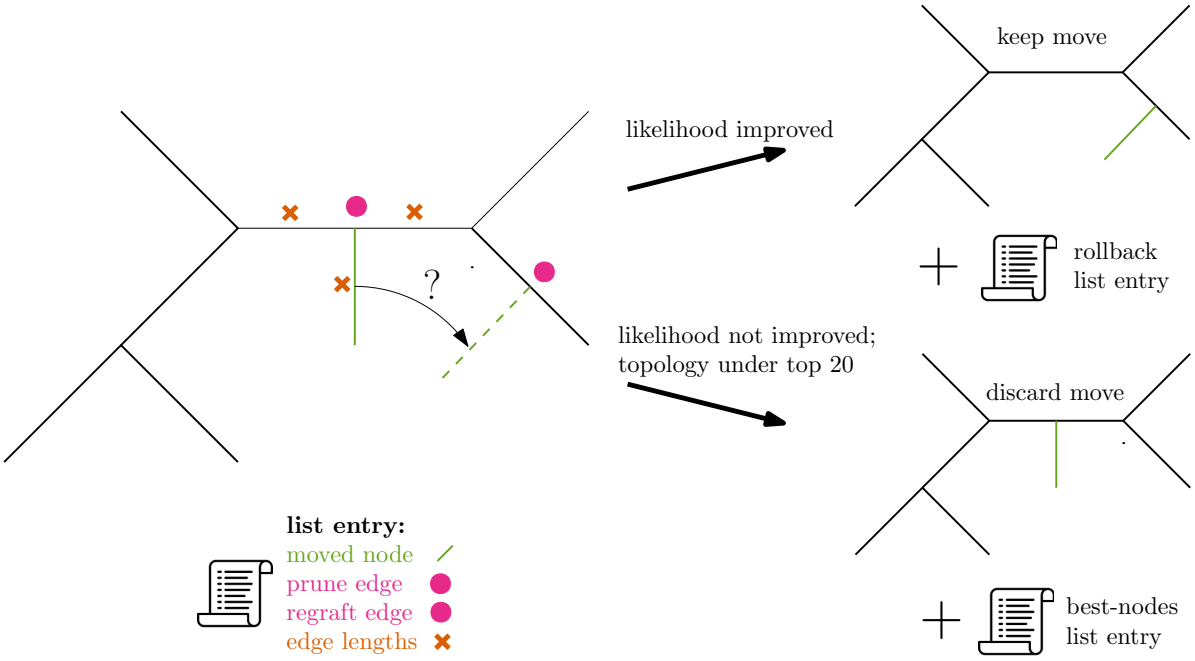


Figure 6.2.: Rollback mechanism of an SPR move. We are pruning and re-grafting the node connected to the green edge. We temporarily move the node to a new location. During this, we save the information needed to undo this move in local variables. We then evaluate the likelihood-score of the new topology. If the likelihood-score improves, we keep the move and save it to the rollback list. If the move does not improve the likelihood-score but the result is under the 20 best scoring topologies, we save it in the best-nodes list. We later re-evaluate all moves in the best-nodes and rollback lists with full branch length optimization to check if they yield a likelihood improvement.

models stays small, the number of models will limit the number of broadcasts required. A maximum of one broadcast per model is sent.

For all runs with either less than 100 ranks or less than 1,000 models, redistributing the model parameters requires at most 11.1 ± 0.2 ms. Only if the number of models *and* the number of ranks increases, the time required by model broadcasting increases. The run on the PeteD8 dataset has 4,116 models and uses 260 ranks. It requires 72.0 ± 0.9 ms per model parameter redistribution (see Appendix A.4.2). Creating a checkpoint of the tree topology requires at most 0.575 ± 0.006 ms (1,879 taxa, see Appendix A.4.1 and Appendix A.4.2).

6.2.4. Runtime Overhead Without Failures

We measure the runtime overhead caused by mini-checkpointing when no failures occur. We expect FT-RAxML-ng (Fault- Tolerant RAxML-ng) to be slower, because it creates mini-checkpoints in addition to the regular checkpoints. This is the penalty we have to pay for fault-tolerance even in the case that we do not need it. We want to separate the run-

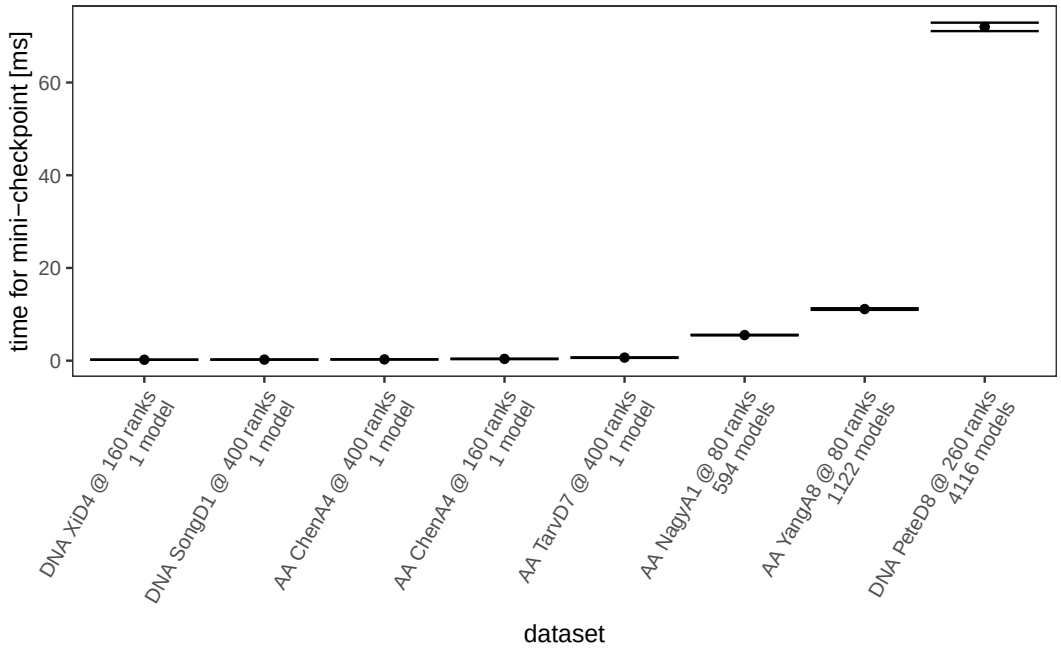


Figure 6.3.: Time required for model parameter broadcasting. The mean (dots) and standard deviation (error bars) of the time required to create each mini-checkpoint is shown. We measure all mini-checkpoints in one tree search (47 to 582). Each node has 20 CPU cores and executes 20 MPI ranks. The empirical datasets are described in Table 4.1. Only if the number of models *and* the number of ranks increases, the time we require to redistribute the model parameters also increases.

time overhead caused by ULFM from the runtime overhead caused by our modifications to RAxML-ng. We therefore measure the runtime of FT-RAxML-ng with OpenMPI v4.0 and ULFM v4.0.2u1 as MPI implementations (see Table 6.1). In our measurements, the slowdown of FT-RAxML-ng running with OpenMPI v4.0 compared to the unmodified RAxML-ng running under OpenMPI v4.0 is 1.02 ± 0.02. The slowdown of FT-RAxML-ng running under ULFM v4.0.2u1 compared to the unmodified RAxML-ng running under OpenMPI v4.0 is 1.08 ± 0.07.

Table 6.1.: Overall runtimes of unmodified RAxML-ng vs FT-RAxML-ng (see Chapter 6) when no failure occurs. That is, we perform mini-checkpoints (model updates) and tree updates but do not simulate failures. "s.dwn": slowdown

type	dataset	ranks	OpenMPI RAxML [s]	OpenMPI FT-RAxML [s]	s.dwn	ULFM FT-RAxML [s]	s.dwn
AA	NagyA1	80	2,985	3,014	1.01	3,025	1.02
AA	ChenA4	160	685	720	1.05	686	1.02
AA	YangA8	80	1,182	1,230	1.04	1,210	1.02
DNA	SongD1	400	1,365	1,383	1.01	1,541	1.13
DNA	XiD4	160	3,760	3,858	1.03	4,466	1.19
DNA	TarvD7	400	700	709	1.01	739	1.06
DNA	PeteD8	260	5,393	5,492	1.02	6,197	1.15

6.3. Recovery after Failure

Checkpointing is only half the battle. The other half is restoring the program state after a failure. In the following, we describe how RAxML-ng resumes tree search after a failure.

6.3.1. Problem Statement

After one or more ranks fails, the remaining ranks have to detect this failure. Next, the surviving ranks have to agree on which ranks are still alive and restore the search to a valid state. As little work as possible should be lost because of a failure. Furthermore, the program has to restart the tree search without user intervention.

6.3.2. Algorithm

The responsibilities arising from the problem described (see Section 6.3.1) are divided between ULFM and RAxML-ng. After ULFM detects and reports a failure, RAxML-ng mitigates the failure, restores the search state, and restarts the tree search.

Detecting and Mitigating a Failure

ULFM provides mechanisms to detect if a rank failure occurred [76]. Rank failures are not detected and reported when they occur but only the next time each rank calls an MPI operation. If a rank failure occurs during an MPI call, the call's return value might indicate an error on some ranks but not on others. Reporting failures on all ranks simultaneously would require all ranks to agree if a failure occurred after each communication. Such an algorithm has been shown to take at least $\mathcal{O}(p^2)$ time, where p is the number of ranks [11].

If ULFM detects a failure during an MPI call, it indicates this in the return value. For details see Section 5.2. In RAxML-ng, the `ParallelContext` class abstracts away all calls to MPI functions. This is true for the `C++` part of the code as well as the `C` part. The latter is passed the wrapper for Allreduce as a callback function. As always, code reuse simplifies code extension. We introduce a new method `faul_tolerant_mpi_call` into the `ParallelContext` class. This method takes a (temporary) function as a parameter. This temporary function calls the MPI function to be executed in a failure tolerant way. The temporary function also passes on the MPI function's return code to the calling `ParallelContext::faul_tolerant_mpi_call`. If ULFM reports a failure during the MPI call, `faul_tolerant_mpi_call` creates a new MPI communicator (see Section 5.2). This new communicator includes only the surviving ranks. Next, `faul_tolerant_mpi_call` updates the world parameters (e.g. rank count, node count) stored in the `ParallelContext` class and throws a `RankFailureException`. The code which issued the fault tolerant MPI call has to catch and handle this exception.

We differentiate between recoverable and unrecoverable failures. In both cases, the result of the respective MPI call is undefined. ULFM does not guarantee, that each rank is notified of a rank failure during the same MPI collective call (see Section 5.2). Thus, different invocations of `faul_tolerant_mpi_call` might throw the `RankFailureException` on different ranks. In the case `faul_tolerant_mpi_call` throws a (recoverable) `RankFailureException`, it has shrunk the communicator successfully to only include surviving ranks. Executing further MPI calls will then be possible. In the case `faul_tolerant_mpi_call` throws a `UnrecoverableRankFailureException`, it failed to create a new valid communicator. The calling code can no longer assume that any MPI call will behave in any particular way or even return at all. In this case, we can do nothing more than exit the program.

To illustrate this, let us consider an example. We change the method `mpi_broadcast(...)` in the class `ParallelContext::mpi_broadcast(...)` to handle rank failures. For this, we replace the MPI call with a call to `fault_tolerant_mpi_call`. We pass a lambda function executing the MPI call. This lambda function passes on the return value of the MPI function. The `fault_tolerant_mpi_call` function executes the lambda function once and checks its return value for a rank failure. If it detects a rank failure, it throws a `RankFailureException`. The `ParallelContext::mpi_broadcast` function lets this exception propagate upwards to its caller which then has to handle it.

Listing 6.1: Simplified code *without* failure mitigation

```
1 void ParallelContext::mpi_broadcast(void * data, size_t size, int root) {
2     if (this->_num_ranks > 1) {
3         return MPI_Bcast(data, size, MPI_BYTE, root, this->_comm);
4     }
5 }
```

Listing 6.2: Simplified code *with* failure mitigation

```
1 void ParallelContext::mpi_broadcast(void * data, size_t size, int root) {
2     if (this->_num_ranks > 1) {
```

```
3           // Using lambdas enables easy captchure of all the needed variables.
4           // If fault_tolerant_mpi_call detects an error, it will throw an exception
5           // which we will pass on to the code calling us.
6           fault_tolerant_mpi_call([&] () {
7               return MPI_Bcast(data, size, MPI_BYTE, root, this->_comm);
8           }};
9       }
10  }
```

We adjust all functions in `ParallelContext` similarly, abstracting away the MPI details. The responsibility of the caller is to catch the `RankFailureException` and restore the search state after a rank failure. We discuss this in the following section.

Restoring the Search State after Failure

After ULFM establishes a new communicator containing only the surviving ranks, the program is still in an invalid state. No data has been redistributed or reloaded. Also, no rank is responsible for calculating the likelihood scores of the sites the failed ranks were responsible for (see Section 3.2). To simplify the process of restoring the search state, we design it as a local operation, that is, no communication between ranks is required until the search state is restored. If another rank fails in the meantime, all surviving ranks will first complete the restoration process and can afterwards handle the additional failure.

To restart the tree search, the algorithm re-executes the load balancer (see Section 3.3), reloads the MSA data, and restores the search state. The load balancer redistributes the MSA sites to the, now, reduced set of ranks. Each rank then loads the respective MSA data using partial loading. In partial loading, each rank selectively reads only those parts of the MSA file from disk it requires for its likelihood computations. To restore the search state, the algorithm copies over the model parameters from the last mini-checkpoint and the currently best tree from the backup copy. Additionally, we need to invalidate and recompute internal caches, for example, the CLVs (see Section 2.2.1). Algorithm 3 shows the fundamental ideas behind the implementation of this procedure in the `TreeInfo` class. `RedoPartitionAssignment` re-runs the load balancer with the now reduced set of ranks. `UpdateInternalDataStructures` is for example responsible for clearing and recomputing the CLVs and the local table of the sites the rank has to handle. Because of the partial loading feature of RAxML-ng, each rank needs to load only those parts of the MSA that it requires for the likelihood computations. Restoring the models and the tree are local copy operations.

6.3.3. Evaluation

We implemented the algorithms described above in RAxML-ng and describe respective experiments in the this section.

Algorithm 3 Restore Search State after Rank Failure

```
1: procedure RestoreSearchState
2:     assignment ← RedoAssignment(new rank count)
3:     UpdateInternalDataStructures(assignment)
4:     Restore tree from local backup copy
5:     for all models do
6:         Restore model from local backup copy
7:     end for
8: end procedure
```

Time Required for Restoring the Search State

We profile the different sections of the recovery procedure (described in Section 6.3.2). For this, we simulate failures (see Section 5.3) and measure the time required for different parts of the recovery procedure. For each run with 20 ranks, we simulate 19 rank failures; for each run with at least 40 ranks, we simulate 39 rank failures.

We expect the time required for reloading the MSA data to increase with the total number of sites times the number of taxa times the number of possible states. Amino Acids have 20 possible states, DNA has four. On the ForHLR II, two file servers (see Section 4.2) handle disk access. This is true independent of the number of ranks we use for the phylogenetic tree inference. Thus, we expect the time for loading the MSA data to not depend on the number of ranks in the computation. We expect the time required for invalidating and recomputing the caches (e.g. CLVs) to increase with the number of sites on each rank. This is, because these operations are embarrassingly parallel.

The time required for restoring the search state is below 100 ms for six out of eight runs, including three with 400 ranks. The remaining two runs are working on datasets with more than 500,000 sites, at least 95 taxa, and required up to 535 ms (see Figure 6.4).

Time Required for Reloading the MSA Data

We further investigat the time required to load MSA data from disk. This is interesting, because the MSA is loaded from a central disk array. Therefore, reloading this data could be a bottleneck when restoring. We measure the MSA loading time required for different datasets on the ForHLR II (see Section 4.2). We differentiate between the initial load operation and all subsequent load operations. “Initial” here means the first time any rank loads this data. If rank 2 loads the data rank 1 has previously loaded, we do not count this as an initial loading operation. In fact, all subsequent loading operations labelled as “further load op.” are loading data which this rank has not loaded before during this run (but other ranks have).

In each experiment, the load balancer decides which rank loads which part of the MSA. The ranks then load their respective part of the data. This is the initial load operation. Next, the load balancer shifts the responsibilities by one, that is, rank 2 now loads the data rank

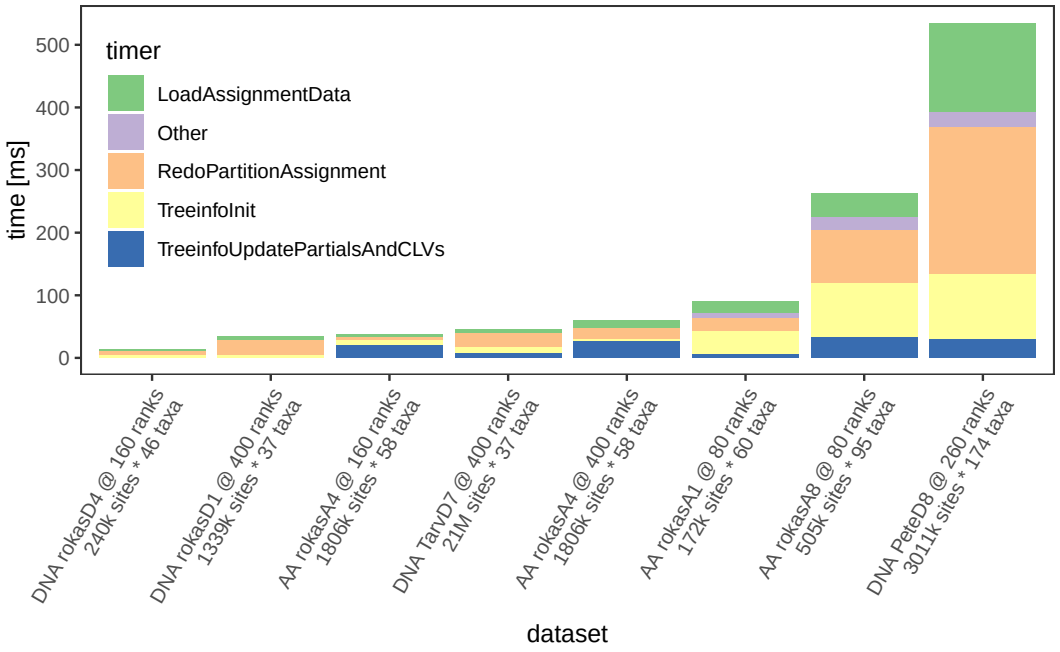


Figure 6.4.: Time required for recovery form checkpoint. `TreeinfoInit`: "LoadAssignmentData" is the time required to load the MSA data a rank is responsible for from disk. "RedoPartitionAssignment" is the time required for re-running the load balancer. "TreeinfoInit" is the time required for updating the `TreeInfo` data structure. This includes hundreds of memory allocations. "TreeinfoUpdatePartialsAndCLVs" is the time required for updating cached likelihoods. "Other" includes for example the time required for updating data structures needed for the mini-checkpointing.

1 previously loaded and so on. We repeat this process until each rank has loaded each part of the MSA exactly once. We call all operations after the first one "further load operations". Figure 6.5 shows the average time required to load a rank's part of the MSA, as well as the standard deviation. For the measurement of the initial loading operation we compute the average and standard deviation across all ranks. For the rebalancing measurements, we first compute the average time required on each rank. Next, we compute the average and standard deviation of the ranks' averages. We perform the measurements on the ForHLR II which has two separate file server nodes connected to the compute nodes via an EDR InfiniBand network. The initial load operation represents an upper-bound on what we expect the loading operation to require when the cache of the file system does not contain a copy yet. We expect the repeated loading of the MSA, albeit on different ranks, to be a reasonable guess on the read performance if the filesystem's cache already contains a copy of the data.

We show the file sizes of the corresponding MSA files in Table A.4 in the appendix. For example the DNA dataset `PeteD8` has 3,011,000 sites and 174 ranks. Its encoding has a size of 500 MiB. This corresponds to the expected 8 bit per site per taxon. In case of frequent failures,

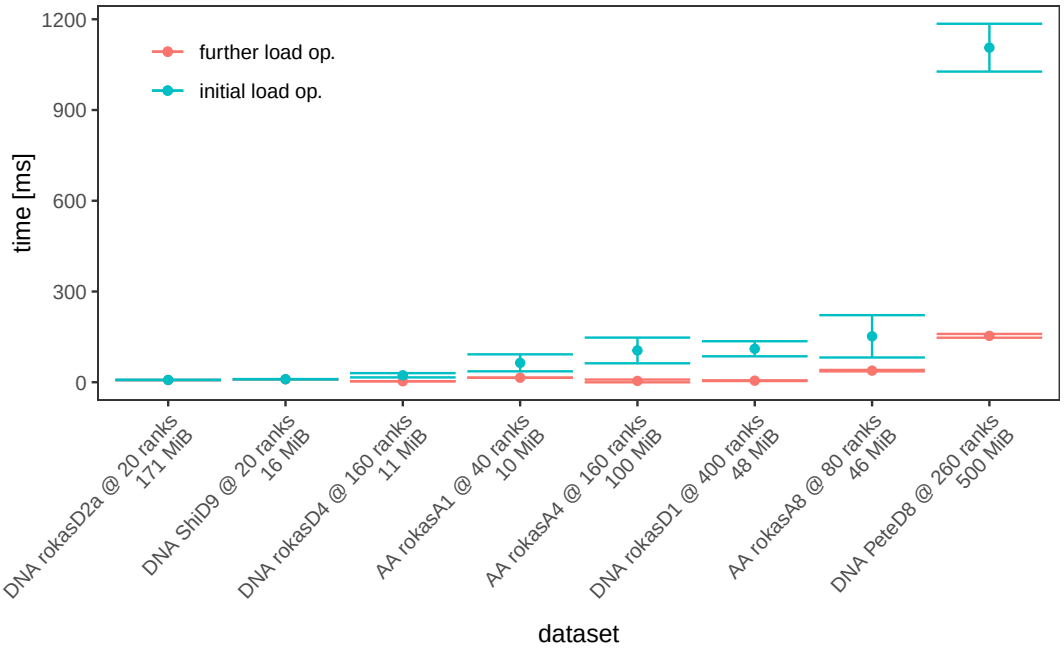


Figure 6.5.: Time required for reloading the MSA from disk on the ForHLR II (see Section 4.2). Red dots and error bars ("initial load op.") indicate the time required to load the rank's part of the MSA the first time this data is accessed by any rank, that is, on program start-up. The blue dots and error bars ("further load op.") indicates the time it took a rank to load a part of the MSA it had never loaded before (but other ranks already have). These data can be assumed to be cached by the file servers.

the cluster's file system will cache the MSA data. This reduces the possible gain from storing the data in the main memory of the compute nodes. If few failures occur, the MSA data will be accessed infrequently. In this case, the 1.2 s overhead for reloading the data could amortize. We performe these measurements on one cluster system only, namely ForHLR II. They can therefore not be generalized.

6.3.4. Runtime Overhead With Failures

We measure the runtime overhead caused when failures occur (see Table 6.2). We expect FT-RAxML-ng (Fault- Tolerant RAxML-ng) to be slower, because it creates mini-checkpoints in addition to the regular checkpoints, has to restore from failures and perform some computations twice. We use OpenMPI v4.0 for all measurements and simulate failures as described in Section 5.3. In our measurements, the slowdown of FT-RAxML-ng running with OpenMPI v4.0 compared to the unmodified RAxML-ng running under OpenMPI v4.0 is 1.3 ± 0.2. In all experiments, the final likelihood-score (see **??** deviates less than 3×10^{-8} from that of the reference run. A difference log-likelihood might for example occur if there is a failure

during an SPR round. In this case, we restart the SPR round from the currently best known tree topology and therefore evaluate different SPR moves (see Section 2.2.2.1).

Table 6.2.: Overall runtimes of unmodified RAxML-ng ("reference") vs FT-RAxML-ng ("runtime") (see Chapter 6) when failures occur. That is, we perform mini-checkpoints (models and tree updates) and simulate failures. We use OpenMPI v4.0 for all measurements and simulate failures as described in Section 5.3 "s.dwn": slowdown

type	dataset	ranks	failures	reference	runtime	s.dwn
AA	NagyA1	80	10	2,985	3,077	1.03
AA	ChenA4	160	10	685	1,093	1.60
AA	YangA8	80	7	1,182	1,832	1.55
DNA	SongD1	400	10	1,365	1,413	1.04
DNA	XiD4	160	10	3,760	4,246	1.13
DNA	TarvD7	400	10	700	998	1.43
DNA	PeteD8	26	10	5,393	7,037	1.30

7. Eliminating Disk Access

Having to reload the MSA from disk after each failure is not optimal. The rollback data structures we describe in Section 6.2 already contains the data that change over the runtime of RAxML-ng. In this Chapter, we discuss two possibilities to store the static MSA data in memory such that it will survive PE failures. Note that storing a full, uncompressed MSA in the memory of each PE would constitute a waste of memory.

7.1. Tree Based Compression of Multiple Sequence Alignments

We will first look into how we can compress the MSA to reduce its size. The shrunken MSA could then fit into the main memory of each PE (see Section 7.1.1). Anè and Sanderson [5] describe a compression scheme for MSAs based on parsimony trees. A parsimony tree is a phylogenetic tree (see Section 2.1) which minimizes the number of mutations along its edges [26]. In this Section, we provide an expansion of the compression scheme to include nucleotide ambiguities (see Section 7.1.1) and an algorithm to encode and decode the given MSA.

As there are four nucleotide states, we need two bits to encode a single nucleotide. When using ambiguities (combination of states, see below) we need 4 bits. By leveraging that the sequences in an MSA are related, we can decrease the number of bits we need to encode a nucleotide on average. Ané and Sanderson predict a compression of up to 0.02 bits per nucleotide and taxon on a dataset with 100 taxa using their compression scheme. The compression efficiency increases on datasets with more taxa [5]. We investigate if this compression is sufficient to allow us to reduce the MSA's size to fit into the main memory of each PE in the next Section.

7.1.1. Description of the Encoding

An MSA consists of a set of aligned sequences. This implies, that every sequence has the same length, possibly including gaps. Sequences can be, for example DNA or AA. We will focus on DNA in this Section, but we can extend the encoding to the 20-state AA alphabet. DNA has four states (`A`, `C`, `T`, `G`). In real world datasets, however, we sometimes want to encode that we are unsure of the exact nucleotide at a certain position. We will call this an ambiguity. An ambiguity can also mean that we observed multiple nucleotides at this position in different sequencing runs. The International Union of Pure and Applied Chemistry (IUPAC) defines 4

nucleotide states, 11 ambiguities and a gap [53]. For example a `K` represents either a `G` or a `T` at a position in the sequence.

The sequences of an MSA are located at the tips of a corresponding phylogenetic tree (see Section 2.2). The main idea of the tree based compression scheme is to encode only one sequence in full, the one marked in **bold** in Figure 7.1. We store all other sequences as a set of changes along the edges of the tree. We annotate these changes next to the edges, for example $5 \rightarrow C$ means that the nucleotide state at the fifth site of the sequence changes to a `C` along the edge. This means that, ancestral states (inner nodes) have a sequence associated with them. These ancestral sequences are not shown in Figure 7.1 but we nonetheless have to compute them. For the reconstruction to work, we have to store the tree topology for the encoding as well. The most parsimonious tree guarantees the shortest encoding [5].

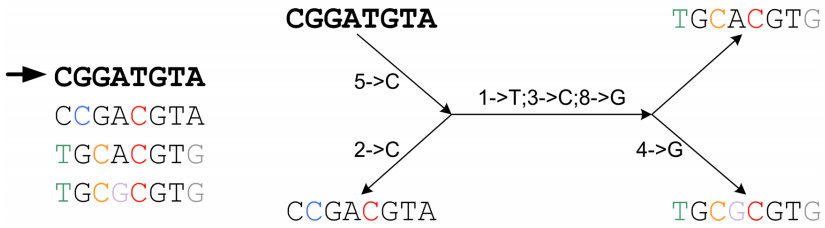
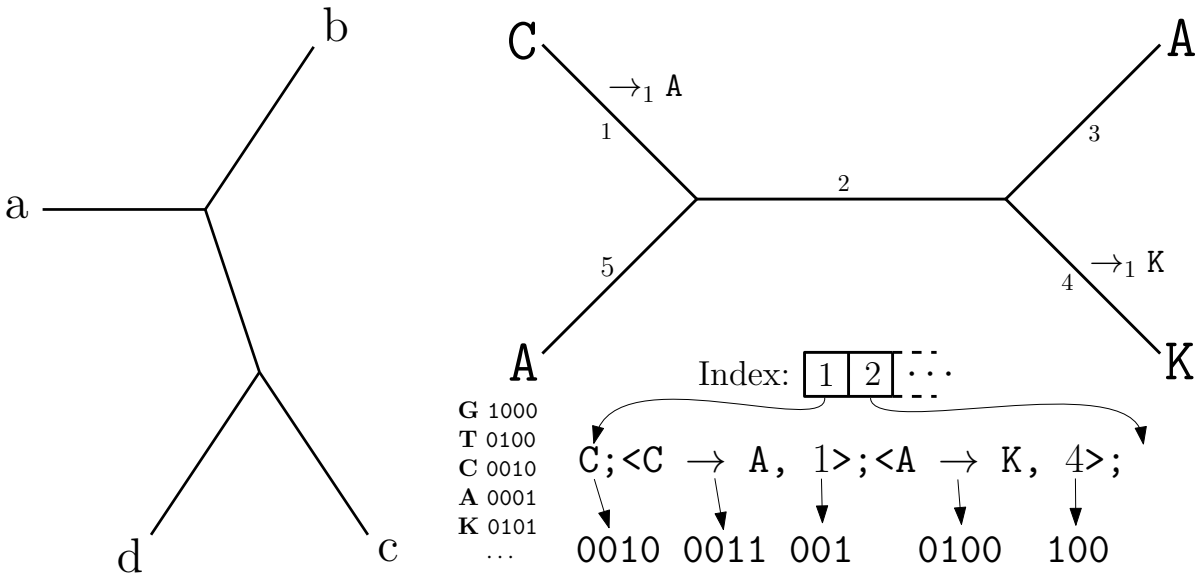


Figure 7.1.: Tree based encoding of sequences in an MSA as described by Ané and Sanderson [4]. We have to encode only the tree topology, the sequence in **bold**, and the changes annotated along the edges. We can then reconstruct the full MSA from this encoding.

In the following, we give a quick summary of the encoding scheme presented by Anè and Sanderson [5] and how we adapt it to 16 nucleotide states. We encode the tree topology in the binary Newick format. Take for example the tree in Figure 7.2a. We can encode it as `(a,b,(c,d));` in the ASCII Newick format. Each pair of parentheses represents an inner node, each letter represents a tip. Commas separate the nodes from each other and a semicolon ends the encoding. By omitting the tip names and exploiting that the tree is bifurcating and unrooted, we can encode the same tree topology as `(())`. We can further use a `1` to represent an opening parenthesis and `0` to represent a closing parenthesis. This way, we can encode the tree topology by bit as `1100`.

We store the ancestral states of the root with four bits using a one-hot encoding. In a one-hot encoding, a single bit encodes for one of the four basic nucleotide states, that is, A = 0001, C = 0010, T = 0100, or G = 1000. A single nucleotide state is therefore 4 bits long. We encode ambiguities by setting multiple bits at once. We encode a gap by clearing all bits. We store the changes to a site directly after the nucleotide state at this site at the root sequences. By doing so, we ensure that all the data we need to decode the nucleotide states at one site is stored continuously, enabling cache-efficient decoding. Additionally, we do not need to store the change's position in the sequence, which could become large, as it is stored following the site it modifies. We also store an index data structure I mapping the site identifier to the start of the encoding for this site. We implement this using an array. We thus gain random read access to the sites of the encoding.



(a) Bifurcating Tree

(b) Sequence encoding

Figure 7.2.: Encoding of the Tree and MSA. (a) A bifurcating tree which we can encode as `(a,b,(c,d));` in the ASCII Newick format. By leaving out the tip names and exploiting that the tree is bifurcating, we can also encode it as `(())` which we can represent as the `1100` bit vector. (b) Encoding of a single site of the MSA. We encode the nucleotide's root state (upper left). Then, we encode the changes to the nucleotide state along the edges of the tree. We encode these changes in pre-order. We encode each change using the change mask and edge number (annotated next to the edges). We obtain the change mask by `XOR`ing the nucleotide states before and after the change. The index data structure points to the beginning of each site's encoding.

We encode the changes of a nucleotide along the edges of the tree as the change's substitution mask and the edge this changes occurs on. We obtain the substitution mask by `XOR`ing the nucleotide state before and after the change. For example, if a `T` (0100) is replaced by a `C` (0010), the substitution mask will be $mask = C \oplus T = 0100 \oplus 0010 = 0110$. In contrast to the substitution mask used by Anè and Sanderson [5], this allows us to handle ambiguities. The mask 0000 encodes for no change and is not needed to encode a change. We can therefore assign the meaning "end-of-changelist" to it. Using the index data structure described above, however, this is not necessary. As some states (for example gaps) are more likely to occur than others (for example `K`) we could use Huffman coding [52] to further compress the states. For now, we do not do this because it will complicate the encoding and decoding procedures.

We identify the edges by a unique identifier. We obtain this identifier by numbering the edges in pre-order (parent, left child, right child). We chose pre-order instead of post-order as used by Anè and Sanderson [5] because this way we store the changes "root to tip", yielding the decoding more straightforward. We can encode the end of the encoding using no-change

and invalid-edge-number. If we are using the index data structure, we can store a dummy entry pointing just past the last valid change to mark the end of the encoding.

Size of the Compressed MSA

We base the following calculations on Anè's and Sanderson's [5] calculations. As we want to allow for ambiguities, we need to allow 16 instead of 4 states. Let us assume, that the encoding size doubles because of this. Let n be the number of sequences, m the sequence length, and L^P the number of substitutions in the parsimony topology. If we store each sequence in full, representing each site with 4 bits, we obtain an encoding of length $4nm$ bits. Using a parsimony-based encoding, we need to describe the shape of the tree using $2n - 4$ bits. Next, we can optionally write the taxon labels, denoted by T. We chose a root of the tree arbitrarily and use $4m$ bits to encode the sequence at this root. We encode the changes to the sites' nucleotide states along the tree. Each of these changes consists of a substitution mask and the edge number this change occurs on. For this, we need $L^P(4 + \log_2(2n - 3))$ bits. We also have to save the index data structure which stores the location of the start of the encoding of each site. This requires mp bits, where p is the size of a pointer. In total, the encoding requires $2n - 4 + T + 4m + L^P(4 + \lg(2n - 3)) + mp$ bits. The encoding length heavily depends on the number of changes in the parsimony tree L^P. As finding the most-parsimonious tree is $\mathcal{NP}$-hard, we must rely on heuristics.

So how much memory do we need for a typical dataset? This depends on the number of changes across the parsimony tree L^P. Let us assume that, using four bits per site instead of two bits per site will double the space needed for the compressed MSA. We can then use the compression efficiency reported by Ané and Sanderson [5] and calculate the additional memory usage for the encoding on each PE.

Table 7.1.: Empiric encoding efficiency for real-world datasets as reported by Ané and Sanderson [5]. L^P is influenced by the diversity between the sequences in these datasets. The column giving the size of the compressed MSA assumes that the encoding efficiency decreases by 50 % when we encode 16 instead of 4 nucleotide states. That is, we need twice the amount of bits per site.

number of taxa	encoding efficiency	size of compressed MSA
100	0.70 bit taxon^{-1} site^{-1}	140 bit site^{-1}
500	0.25 bit taxon^{-1} site^{-1}	250 bit site^{-1}
1,000	0.20 bit taxon^{-1} site^{-1}	400 bit site^{-1}

The largest dataset we considered in Chapter 4 is `TarvD7` . It consists of around 21 million sites. Let us assume it would also have 500 taxa. An analysis of this theoretical dataset would require 625 MiB of additional storage for the MSA on each PE. As all cores in a single node share the same memory. If we assume that all ranks on a node fail if a hardware failure occurs on this node, we need to store the compressed MSA only once per node, not per

rank. We never update the compressed MSA once it has been loaded. We therefore expect concurrent access not to be an issue. Especially as the different cores on the same shared memory machine read different parts of the alignment.

This leaves the question if we can spare 625 MiB on each node? Let us assume an analysis with tip-inner and pattern compression turned OFF (see the RAxML-ng manual), using 1,000 site PE^{-1}, $S_{DNA} = 4$ nucleotide states per site, $R = 20$ rate-categories, and $|MSA| = 500$ taxa. Let us denote the number of CLVs with $|CLVs|$. We can then compute the memory requirements for the CLVs per PE, denoted as M_{CLV}, using the following formula:

$$\begin{aligned} M_{CLV} &= |CLVs| \cdot \mathrm{sizeof}(CLV) \cdot \mathrm{sizeof}(double) \\ &= (2|MSA| - 2) \cdot S_{DNA} \cdot R \cdot 8\,\mathrm{B} \\ &= 0.609\,\mathrm{MiB\ site}^{-1} \end{aligned} \tag{7.1}$$

Assuming 20 cores per node, as on ForHLR II, the CLVs would take up 12.78 GiB on each node. This would allow us to use the remainder of the memory, 55.95 GiB on ForHLR II, to store the compressed MSA. We could therefore store up to 2,100,000,000 sites, around two thirds of the human genome or 1/75th of the largest known plant genome.

7.1.2. Description of the Algorithm

In this Section, we provide a description of the algorithm for compression and decompression of the MSA. Compressing the tree consists of finding the ancestral states of the parsimony tree as well as encoding the tree topology and the sequences. For decompression, we need to first decode the tree and then the sequences (see Section 7.1.1).

Finding the Ancestral States of the Parsimony Tree

Given the sequences at the tips as well as the parsimony tree, Fitch [35] and Hartigan [48] each propose a method for reconstructing the ancestral states of this tree. Both published their algorithms before the introduction of Sanger Sequencing in 1977. They focus on building small trees from morphological traits instead of today's large trees based on molecular data. We can use Hartigan's algorithm to calculate a single assignment of sequences to inner nodes. This assignment has the property, that the number of mutations across the tree is minimal. It is not necessarily the only such assignment. Hartigan also provides a proof of correctness [48].

Hartigan's algorithm [48] takes a phylogenetic tree with fixed topology and fixed states at its tips as input. The algorithm consists of two phases (see Algorithm 4). The first phase assigns a set of possible ancestral states to each inner node of the phylogenetic tree. The second phase then selects one ancestral state per inner node. It does this in a way that minimizes the number of mutations across the tree.

To compute the possible ancestral states, we look at each inner node in post-order (left child, right child, parent). This means that once we get to a node, we already processed both its children. For each tip, the set of possible states consists of the single fixed state we received

Algorithm 4 Hartigan's [48] algorithm: Overview

Given: A Tree T with i tips and the corresponding MSA S with i sequences $S_0 \dots S_i$

1: **for each** $s_i \in S$ **do**
2: PHASE1(s_i, T)
3: PHASE2(s_i, T)
4: **end for**

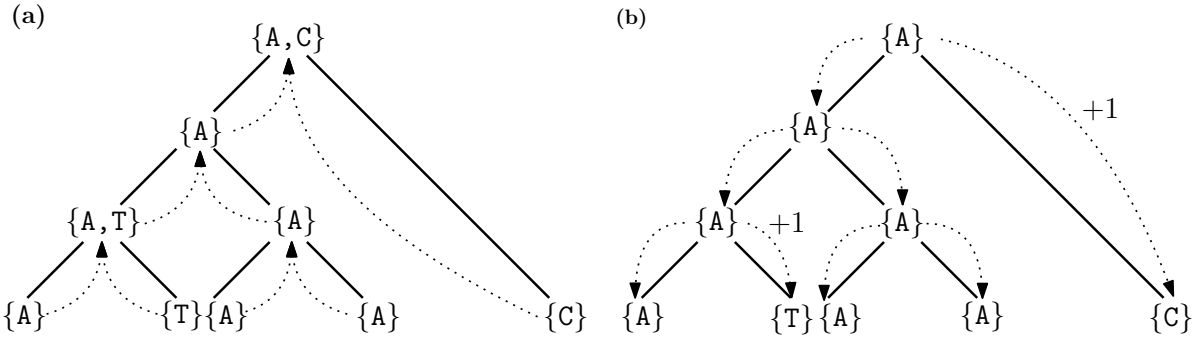


Figure 7.3.: Reconstruction of the ancestral states as described by Hartigan [48]. A bifurcating phylogenetic tree with fixed topology and sequences at the tips is given. (a) In the first phase, we build possible ancestral states. If both children of the same parent have common states, we set these as possible ancestral states. If the two children do not have common states, we set the union of the children's states as possible ancestral states. (b) In the second phase, we chose ancestral states. For the root, we chose a state randomly from its possible ancestral states. For each node, we check if its parent's ancestral state is a possible ancestral state of the node. If it is, we chose it as the node's ancestral state. If it is not, we choose a random state from the child's possible ancestral states. In this case, a mutation occurred (+1).

as input. For all inner nodes, we look if the current node's children have common possible states. If they do, the current node's possible states is the intersection of the children's states. In case they do not, we assign the union of the children's possible states to the current node (see Figure 7.3.a and Algorithm 5).

To select an ancestral state for each node, we start at the root and traverse the tree in pre-order. This means, that we once we get to a node, we have already processed its parent. For the root, we choose one of the possible states at random. For each inner node, we choose its parents state, if this state is a possible state of the current node. If it is not, we choose a random state of the current node's possible states. In this case, one mutation occurred. For the tips, the states are already set, and we do not alter them. If the tip's state differs from its parent's state, a mutation occurred (see Figure 7.3.b and Algorithm 6).

In a bifurcating tree, $|E| = |V| - 1$ holds. Therefore, we can perform a Depth First Search (DFS) in $O(|V|)$ time. As the sequences of the MSA are located at the tips, the tree has $|V| = 2n - 1$ nodes, where n is the number of sequences in the MSA. In phase 1, we compute a set intersection and possibly a set union for each node. There are only 16 possible values in

Algorithm 5 Hartigan's [48] algorithm: Build possible ancestral states

```
1:  function Phase1(Site s_i, Tree T)
2:      traverse T in post-order
3:          if current node N is a tip then
4:              V(N) ← {nucleotide(N)}
5:          else
6:              let A and B be the children of N.
7:              if V(A) ⋂ V(B) ≠ ∅ then
8:                  V(N) ← V(A) ⋂ V(B)
9:              else
10:                 V(N) ← V(A) ⋃ V(B)
11:             end if
12:         end if
13:     end traversal
14: end function
```

Algorithm 6 Hartigan's [48] algorithm: Select ancestral states

```
1:  function Phase2(Site s_i, Tree T)
2:      For the root R of T, choose an element S from V(R) at random.
3:      traverse T in pre-order                          ▷ Skipping the root
4:          let the current node be A and its parent be P.
5:          if V(P) ⊆ V(A) then          ▷ We already set V(P) to a single element.
6:              V(A) ← V(P)
7:          else
8:              V(A) ← {RandomChoice(V(A))}
9:          end if
10:     end traversal
11: end function
```

the sets. We can therefore compute unions and intersections in $O(1)$ time. We can use, for example, a binary set representation and bitwise `OR` and `AND` functions for this. In phase 2, we have to compute an element-of and random choice. We can compute element-of in $O(1)$ time using a bitmask. We replace the random choice with always choosing the Most Significant Bit (MSB) in $O(\log(16)) = O(1)$ time. The overall runtime for computing the ancestral states is therefore $O(n)$.

Encoding the Tree

We can encode a bifurcating tree T in a binary Newick format (see Figure 7.2a). In the ASCII Newick format, each set of opening and closing parenthesis represents an inner node of the tree. Inner nodes can be named and branches can have lengths, but we do not need to use

this information. In our case, each string substitutes the name of a sequence at a tip of the tree. We provide an example in Figure 7.1, whose Newick encoding is `(a,b,(c,d));`. As the tree topology is fixed, we can store the sequence names separately. As long as they are in a defined order, the assignment of sequence names to nodes is unambiguous. We use the sequence names to associate sequences, stored in the MSA to the tips. We can also omit the semicolon and the commas. As our tree is bifurcating, the encoding will still be unambiguous. We therefore encode our example tree as `(())`. We can use a 1 to represent a `(` and a 0 to represent a `)`. This yields the binary encoding `1100`.

Algorithm 7 Tree encoding

Given: A bifurcating tree T.

1: **function** EncodeTree(Tree T) ▷ Runtime: $O(|V|) = O(2n-1)$
2: *newickString* ← ToNewickStringRooted(T) ▷ Runtime: $O(|V|)$
3: **for each** $c \in$ *newickString* **do**
4: Write 1 for '(' and 0 for ')'
5: **end for**
6: Write name of sequences in the order they appear in the Newick string
7: **end function**

We can compute a tree's encoding by iterating over the ASCII Newick string (see Algorithm 7). To encode the tree topology, we skip all non parenthesis, write out a 1 for each opening parenthesis, and a 0 for each closing parenthesis we encounter. We encode the names of the sequences in the same order as they are in the Newick string. To avoid ambiguity, we list tips before inner nodes in the list of children of each node. That is, we encode the example tree (see Figure 7.2a) as `(a,b,(c,d));` and not as `((c,d),a,b);`. This algorithm uses a procedure to compute the Newick string of a tree already implemented in RAxML-ng. If such a procedure is not available, we can use an algorithm analogous to the one we describe for decoding a tree (see Algorithm 8).

To decode the tree from the binary stream, we read it bit by bit (see Algorithm 8). The encoding always starts with a 1. Each time we read a 1, we create a new node in our tree data structure. We also initialize a counter $C[N]$ storing how many children this node still needs. The first node ("root") needs three children, all other inner nodes need two children. A tip has no children and is not explicitly encoded. We therefore need to fill in the tips when we encounter the end of the list of children of an inner node (marked by a 0). If a inner node has one tip and one inner node as children, the tip will be the left child. If the root has tips as children, they are also left of the inner node(s). This ensures that the mapping of sequences to tips is unambiguous.

We insert this new node as the right child of the current node and decrement $C[\mathit{currentNode}]$. If the current node already has a right child, we will insert the new node as the left child. If the current node already has two children, it has to be the root node, and we add a third child to the left of the two existing children. Next, we set the newly inserted node as the new current node. Each time we encounter a 0, we add all missing tips under the current node. If

Algorithm 8 Tree decoding

```
1:  function DecodeTree(File F)                          ▷ Runtime: O(|V| * T_insertPE)
2:      let T be the resulting tree, and N be the current node
3:      let C[] be an array storing the number of expected children per node
4:      assert F.ReadBit() = 1
5:      N ← T.root
6:      C[N] ← 3                                         ▷ The root will have three children
7:      open ← 1
8:      repeat
9:          c ← F.ReadBit()
10:         if c = 1 then
11:             assert C[N] > 0
12:             C[N] ← C[N] − 1
13:             open ← open + 1
14:             if N has no right child then
15:                 Insert a new node as the right child of N.
16:                 N ← N.leftChild
17:             else
18:                 Insert a new node as the left child of N.
19:                 N ← N.rightChild
20:                 open ← open − 1
21:             end if
22:             C[N] ← 2                                 ▷ The new node will have two children
23:         else
24:             Insert C[N] children under N
25:             C[N] ← 0
26:             N ← N.parent
27:         end if
28:     until open = 0
29:     assert ∀N : C[N] = 0
30:     Read the names of the sequences
31:     Get associated sequences from the MSA
32: end function
```

only one tip is missing, we add it to the left of the existing inner node. The node will then have exactly three children if it is the root, or two children, otherwise. We use the counter $C[N]$ to detect how many children this node is still missing. Next, we move up to the parent of the current node. If we have read the same amount of 1s and 0s, the encoding is finished. Next, we read the names of the sequences and assign them to the tips in the same order they were written, for example in pre-order. We then get the associated sequences from the MSA.

Encoding of the Sequences

Given the tree T with n sequences $S^* = \{S^1, S^2, \ldots, S^n\}$ at the tips and $n-1$ ancestral sequences $A^* = \{A^1, A^2, \ldots, A^n\}$ at the inner nodes, we can now describe the compression of a MSA (see Figure 7.2b and Algorithm 9). We will denote the s-th site of the j-th sequence as S_s^j.

To facilitate read access to random sites, we store the start of the encoding of each site in an index data structure I. We know the number of sites in advance. The site identifiers range from 0 to the number of sites minus one. We can thus use a simple vector for I. Also, we can skip the number of bytes required by I when writing the encoding. We can then later come back and write I here. As we need I before the encoding of the sequences, this makes decompressing more straightforward.

For each site i, we write the nucleotide state s_i^{root} at the root sequence, followed by the changes to this site along the tree. To encode the changes, we traverse the tree in pre-order. We number the tree edges in pre-order, too. If the current node's nucleotide state for this site differs from that of its parent, we have to encode a change. We do this using the edge number leading to the current site as well as the nucleotide change mask (see Section 7.1.1).

Algorithm 9 MSA compression

```
1:  function Encode(Tree T, Sequences S*)                ▷ T_EncodeTree + m * T_dfs
2:      let I be a vector mapping each site to its start location in the encoding
3:      EncodeTree(T)
4:      Skip space for |S^root| + 1 pointers in the output stream to later store I in
5:      for each s_i^root ∈ S^root do
6:          I.PushBack(<i, current position>)
7:          Write s_i to output stream                   ▷ 4 bit, optionally use Huffman coding
8:          traverse T in pre-order                      ▷ Skipping the root
9:              let A be the current node
10:             let e_j be the edge from A's parent to A; number edges by pre-order
11:             if s_i^A ≠ s_i^parent then
12:                 Write <s_i^A, j>.                    ▷ See coding explanation in Section 7.1.1.
13:             end if
14:         end traversal
15:     end for
16:     M.PushBack(<EOF, current position + 1>)
17:     Go back and write I
18: end function
```

To decode an MSA (see Figure 7.2b and Algorithm 10) we first read the tree topology as described in Algorithm 8. Next, we read the index data structure, mapping the site identifiers to the start of their encoding in the bitstream. For each site s we want to decode, we go to the specified location and start reading. The first four bits we read are the site's nucleotide state at the root sequence s^{root}. We then traverse the tree T, applying the changes along the edges.

We read the changes in the same order as we wrote them, that is, pre-order. Thus, we will never have to go back in the tree traversal to apply a change.

Algorithm 10 MSA decompression

```
1:  function DECODE(File F, Range of Sites R ⊆ [1, |S^1|])   ▷ Runtime: |R| * T_dfs ∈ O(m * n)
2:      T ← DECODETREE(F)
3:      I ← READI(F)
4:      for each s ∈ R do
5:          Go to start of the site's encoding in the file                ▷ As indicated by I
6:          Read S_s^root from the input stream                           ▷ One hot-encoded
7:          Read < substitutionMask, edgeID > from input stream
8:          traverse T in pre-order
9:              Set the node's nucleotide state to its parent nucleotide state
10:             if next change is on the edge leading to the current node then
11:                 Apply change-mask to the current node' state
12:                 Read < substitutionMask, edgeID > from input stream
13:             end if
14:         end traversal
15:     end for
16: end function
```

Note that we describe how to write the encoding to a file. If we want to keep the compressed MSA only in memory, we can simplify the algorithm. In this case, we do not need to encode and decode the tree. We also do not need to store the index data structure I in the same bitstream as the compressed sites.

We encode sites independently of each other. We can therefore compute and write the encoding for each site sequentially on a single PE. Alternatively, we can distribute the sites across multiple PEs and collect the encoding afterwards. At no point in time do we have to keep all sites in memory on the same PE. We therefore do not introduce a memory bottleneck.

7.2. General Redundant In-Memory Static Storage

The MSA compression we describe above (see Section 7.1) is specific to our application domain. In this section, we present a general approach to storing invariant data redundantly in memory across multiple PEs. In this case, the domain specific part of our algorithm consist only of the redistribution of likelihood computations after a failure.

7.2.1. Problem Statement and Previous Work

The load-balancer assigns each PE a set of sites for which it has to perform the calculations (see Section 2.2.1). For this, it needs to hold the alignment data for these sites in memory.

After a PE failure, we have to recalculate the assignment of sites to PEs. The PEs then have to load the subset of the alignment data they need for calculating the likelihood score on the sites assigned to them. As reloading from disk can be too slow, the assignment data of all sites should be kept in memory, distributed across all PEs. As we need to access this data after one or more PEs failed, it is crucial that we store this data redundantly.

7.2.2. Preliminaries and Related Work

Different fields of computer science are in need of data duplication for recovery purposes. One example is Redundant Array of Inexpensive Disks (RAID) storage. To increase reliability, a RAID system either mirrors the data to additional disks or uses a parity code. Parity codes represent a way to reduce the number of copies we need to restore the data. They work by storing one copy of the data as well as the sum of the data instead of multiple copies of the data [81]. For example in a three disk setup, disks A and B store the data and disk C stores the bitwise XOR $C = A \oplus B$. This is called a Reed-Solomon code. Reed-Solomon codes can be extended to an arbitrary number of data storing instances (disks in RAID, compute nodes in HPC). They can also be extended to handle an arbitrary number of failures. In this case, the computational effort will increase [86, 93].

Plank used RAID-like `XOR`-sums to improve disk-based checkpointing in HPC applications [84]. Bosilca *et al.* [14] applied Reed-Solomon codes to in-memory checkpointing in a matrix-matrix multiplication algorithm. All of these techniques assume that we can replace failed disk or PEs and therefore do not need to redistribute the data.

A performance evaluation of mirroring-based vs parity-based checkpointing on SIMD machines, found parity-based methods to be an order of magnitude slower than mirroring-based methods [21]. This is, because if we want to restore a block of data in a mirroring-based duplication system, we need to transfer only one copy of exactly this block over the network. If we, however, want to do the same in a parity-based duplication system, we need to transfer and `XOR` multiple blocks. Dimakis *et al.* [24] reduced the amount of data transfer required compared to basic Reed-Solomon codes. Chen and Dongarra [20] present a strategy to make parity-checkpointing scale independently of the number of PEs by using a pipelined calculation of the parity-checksum and subgroups. That is, we divide the data into blocks, on which we compute the checksum in parallel across different PEs; not unlike pipelining in modern CPUs. This improved parity-based scheme still needs to transfer more data than a mirroring-based approach [24].

Peer-to-Peer (P2P) networks and cloud file systems also have to deal with failing storage. They, however, are facing different challenges than we are. In both settings we can assume that, while storage space should not be wasted, we will always have enough space available to create another replica of a file. Additionally, in P2P networks, decreasing peer-to-peer bandwidth usage is often substantially more important than decreasing disk usage [49].

In our case, the amount of memory available to store additional copies of the MSA is limited. Data loss, however, is less severe than for example in a file system as the MSA data is still

kept on disk. We are also trying to reduce the time taken for restoration, that is, we want to minimize the latency and not the bandwidth used.

7.2.2.1. h-Relations

In parallel computing with distributed memory and message passing, the h-relation problem arises. It occurs if each PE has at most h messages to send and at most h messages to receive. The source and destination of each message is not constrained. Communication is carried out in rounds. Each PE is able to send and receive one message per round (full-duplex). The task is to find an order in which to send these messages, such that we require as few rounds as possible [1].

7.2.3. Redistribution of Calculations

On program start-up, the load balancer assigns each PE a set of sites. This PE is responsible for computing the likelihood scores of these sites. After a PE fails, we have to redistribute the sites it was responsible for. We cannot know in advance if and when a PE is going to fail. It is also possible that more than one PE fails at the same time or before we completed recovery. We might therefore need to redistribute more than one PE's share of sites.

How much work does each PE obtain?

We decided not to replace failed nodes but to rebalance the load onto the surviving PEs (see Section 5.1). As the number of PEs is reduced through failure, at least one processor has to receive more work. We can set a limit on how much new work each of the p PE gets. For example $a \cdot \text{workToRedistribute}$, where a is a factor greater than $1/p$ but less than one. We want to choose a such, that the work gets distributed among as many PEs as possible without introducing new partitions to the PEs. To simplify things further, we can ignore site-repeats (Section 3.3) when looking at the work each PE has to perform. By doing this, the work of a PE scales linearly with the number of sites we assign to it. We can therefore use the number of sites instead of work in the above term.

Which PE obtains which work?

Currently, we rerun the initial load balancer for the reduced set of PEs. This yields an assignment of sites to PEs which is uncorrelated with the old one. Therefore, all PEs might need to load new sequence data. Our goal should be to avoid this.

To reduce the number of PEs which need to load data, we can use a greedy approach to assigning work to sites. We assign each site we need to redistribute to a random PE which already computes the likelihood score of another site in the same partition. If there is no such PE with spare capacity left, we redistribute the remaining sites randomly across PEs with spare capacity. In this case, we lift the restriction that these PEs must already have another site of the same partition assigned to them.

A more elaborate approach would be to build a bipartite graph with the sites we want to redistribute on the left-hand side and the PEs on the right-hand side (see Figure 7.4). We connect each site to all PEs which already have other sites of the same partition assigned to them. Next, we search for a maximal b-constrained matching with $b(site) = 1$ and $b(PE) = capacity(PE)$. A b-matching is an expansion of the normal matching problem in which the maximum number of edges in the matching incident to each vertex v is bound by a function $b(v)$. See Section 7.2.6 on how we can solve this problem algorithmically. If there are unmatched sites, we randomly distribute them among the PEs.

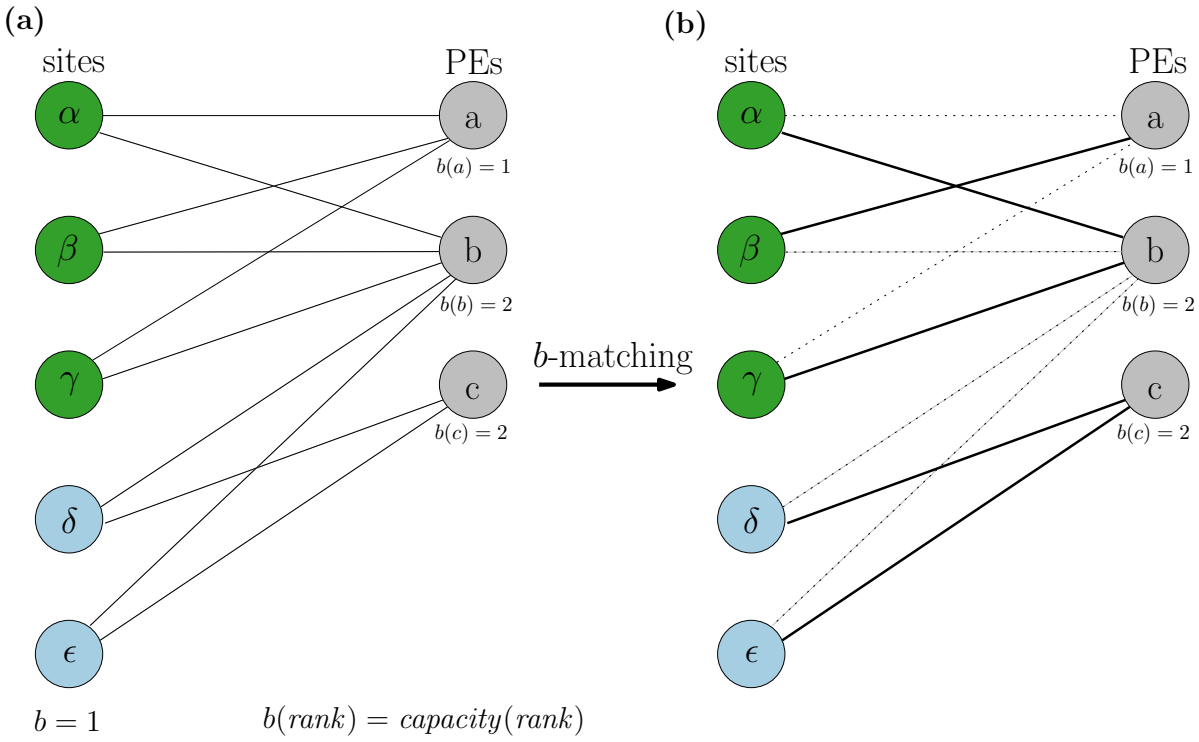
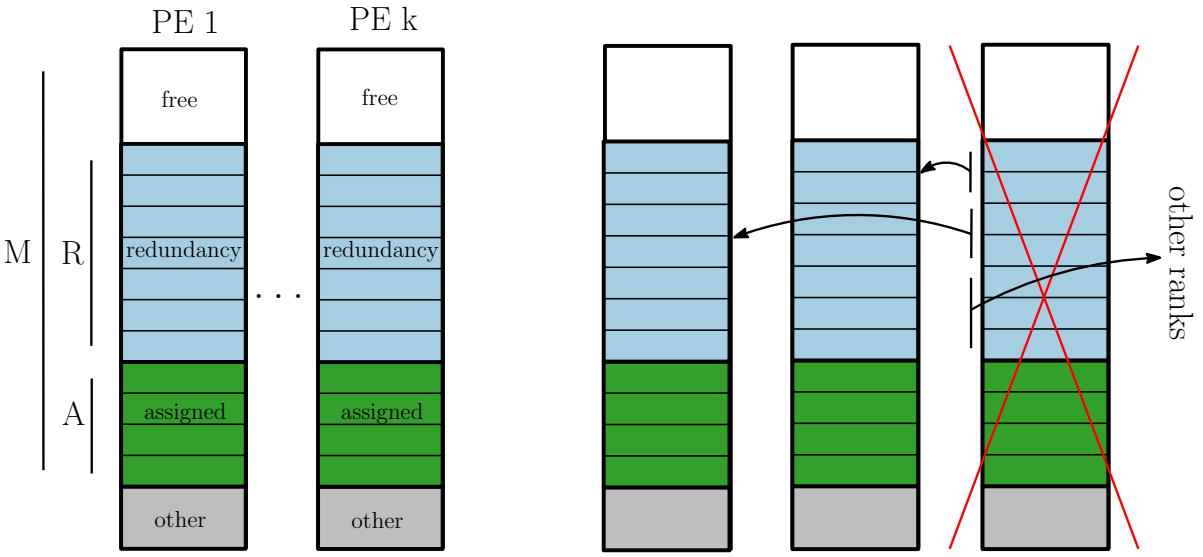


Figure 7.4.: Redistribution of Calculations. (a) Sites we want to redistribute are shown on the left. PEs that could get new sites are shown on the right. The sites belong to one of two partitions: blue and green. We connect each site to all PEs which already have other sites of the same partition assigned to them. For example PE 1 already has sites belonging to partition green, PE 2 has sites belonging to both partitions, and PE 3 has sites belonging to partition blue. (b) A b-matching induces an assignment of sites to PEs which already have a site of this partition. We never exceed a PE's capacity. If sites remain unmatched, we distribute them randomly.

7.2.4. Restoring Redundancy After Failure

If one or more PEs fail, we will lose copies of at least one block of MSA data. The redundancy therefore decreases. To increase the resilience of the system against multiple failures occurring over time, that is, not at once, we can restore this redundancy.

Each PE has a finite amount of memory M which we can use for storing alignment data for likelihood computations A and redundant copies of other PE's alignment data R (see Figure 7.5a). If one PE fails, we loose at most one copy of each site's alignment data. We decided which PE has to replicate and store another copy of A while redistributing the likelihood computations. We also have to redistribute the redundant copies R of the failed PE among the remaining PEs (see Figure 7.5b). For now, let us assume that there is sufficient memory left to do this. We can assign, (not transfer yet) each block of sites to the remaining PEs using a pseudo-random permutation. The number of blocks on the failed PE can be higher than the number of remaining PEs. In this case, we need to assign multiple blocks to some PEs. The number of blocks we assign to each PE will differ by at most 1.



(a) memory layout **(b)** redistribution of blocks

Figure 7.5.: M denotes the entire memory available for storing the assignment data. "Other" includes the CLVs (see Section 2.2.1) and therefore encompasses the majority of the total memory requirement of RAxML-ng. (a) The memory layout. We group sites into blocks. Each PE stores copies of the blocks it needs for its likelihood calculations (green; A). Additionally, each PE stores copies of other blocks (blue, R) to provide them to other PEs in case of failure. (b) If a PE fails, we have to redistribute its redundancy copies (blue, R) among the remaining PEs. We reassigned its blocks belonging to A already in the previous step (see Section 7.2.3).

It can happen that we assign to a PE the copy of a block it already stores. Another copy of this block can be part of the PE's alignment data for likelihood calculations A or redundancy

copies R. In both cases, the PE has to exchange replication responsibilities with another PE (see Figure 7.6). Let r be the number of (additional) copies of the alignment data. For each PE that stores at least two copies of the same data after the initial reassignment of the alignment data, we have to evaluate at most r other PEs as possible exchange partners. This is, because the exchange partner is not allowed to have a copy of the block itself as this would reduce redundancy. The exchange partner must also have a block the "source" PE of the exchange does not have: If the exchange partner ("destination") has fewer blocks than the source PE, it gets assigned the block and does not give up another block. The maximum number of blocks on any PE will not increase. If the destination has more blocks than the source PE, at least one of these blocks must be a block the source PE does not have. We can then exchange this block. If the destination has the same amount of blocks an exchange is possible, too. If the destination already has the block we are trying to exchange, it is one of the at most r invalid destinations. We have already filtered these out it the previous step. The destination therefore cannot have a copy of this block already. This means, that at least one of its blocks cannot be present at the source PE and the PEs can therefore swap them. We can calculate which blocks need to be swapped before we transfer them to the respective nodes. We do not need to actually transfer blocks between the source and destination during this step. Instead, we input the computed responsibilities in the h-relations algorithm we describe in Section 7.2.5. For all of this, no communication between the PEs takes place as we conduct all these computations offline. We transfer only the data in the next step. The result will be a list of PEs to which we have assigned a new block to store in their R space.

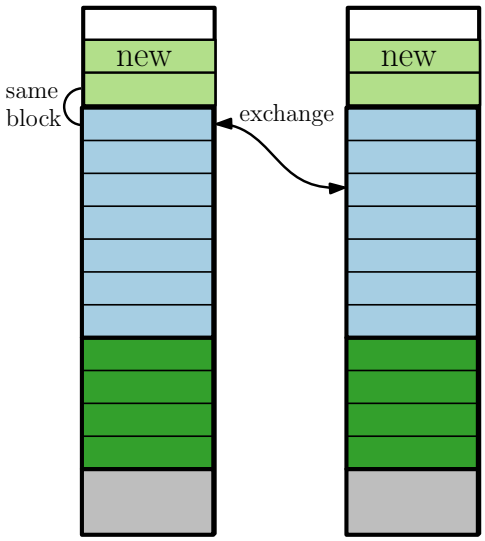


Figure 7.6.: If a PE stores a block twice, redundancy suffers as multiple copies of a block would be lost in a single PE failure. The PE therefore has to exchange a block with another PE.

Reducing redundancy when running out of memory

It is possible, that the amount of memory across all PEs is not sufficient to store the current amount of redundant copies after a PE failure. In this case, we have to reduce the degree of redundancy. For each block that did not lose one copy during the PE failure, we need to mark exactly one copy for deletion. We do not need to mark copies evenly across the PEs. If some PEs have more free space than others, we can fill this free space in the subsequent redundancy copy redistribution step. We can therefore mark random copies of blocks with extra redundancy for deletion.

7.2.5. Redistribution of Data

Both, restoring redundancy and redistribution of likelihood-computation responsibilities requires transmission of data blocks among the PEs. Up until now, we did not transfer any blocks. We only computed which PE needs which block. We will now consider how to efficiently transfer these blocks over the network. Each PE may need to receive multiple blocks of data, each of which might be present at multiple other PEs. We need to find an assignment of source to sink PEs describing which PE will send which data block to which other PE. This is an extension of the h-relation in which multiple PEs can send the same data (see Section 7.2.2.1). We need to minimize the maximum number of blocks a single PE has to send. We have already set the number of blocks each PE has to receive in the previous steps.

We can express the problem as a graph. We write PEs which need to receive blocks on the left side, blocks in the middle, and PEs which can send blocks on the right side (see Figure 7.7). We connect each PE on the left to all the blocks it needs. If two PEs need the same block, we will duplicate the node representing the block (middle column). We connect each PE on the right to each block it can send. Next, we compute a block-saturating minimum b-matching on the bipartite subgraph of nodes representing blocks and nodes representing source PEs as well as the respective edges (middle and right column). In such a matching, each block is incident to exactly one matching edge. Each source PE can be incident to a maximum of b matching edges. We need to find the minimal b_{min} which fulfils the block-saturating property. Such a matching will turn the multi-source h-relation into an ordinary h-relation. We can then compute the order in which to transmit blocks using for example the algorithm presented by König [66]. We describe how to compute a unilaterally-saturating minimum-b matching in Section 7.2.6.

7.2.6. Unilaterally-Saturating b-Matchings in Bipartite Graphs

The b-matching problem is a generalization of the matching problem in graphs, where the objective is to choose a subset of M edges in the graph such that at most a specified number b of edges in M are incident to each vertex v. We call a vertex saturated, if it is incident to an edge in the matching. A perfect matching is a matching in which all vertices are saturated [89]. For bipartite graphs we define an unilaterally-saturating matching as a matching which saturates

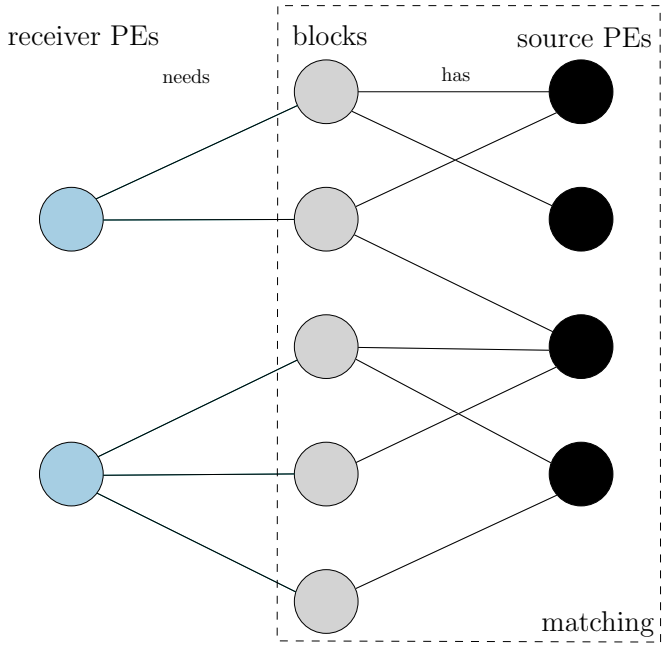
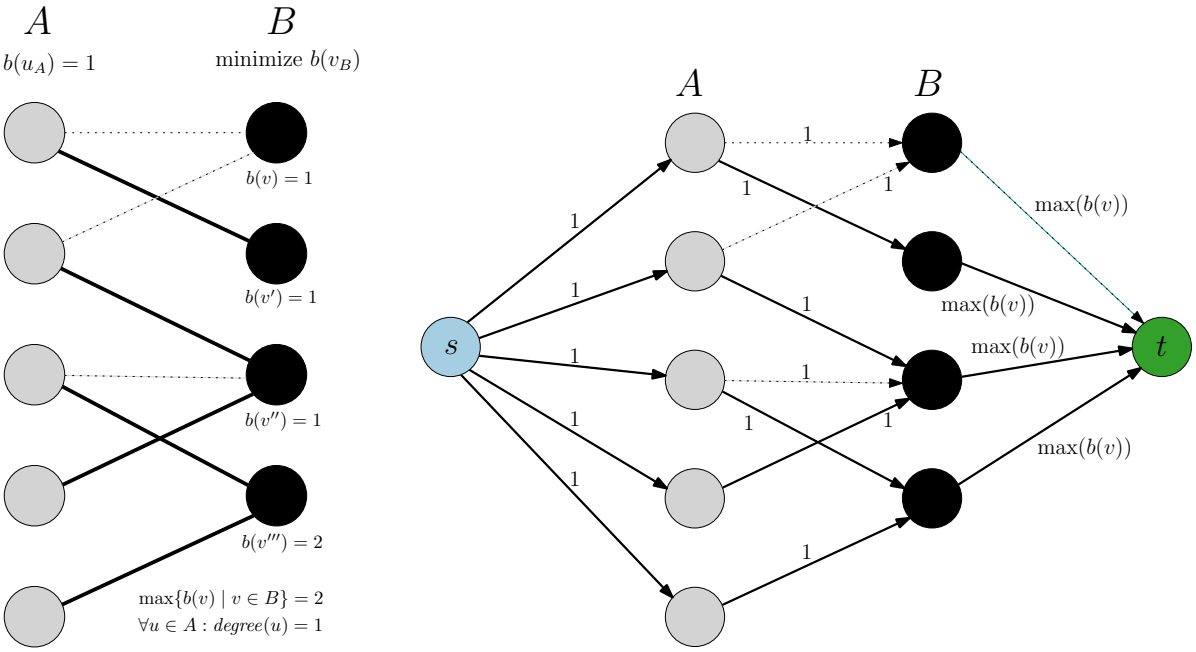


Figure 7.7.: Redistribution of data block. We model the redistribution as the multi-sender h-relations problem which we solve using a left-saturating minimum-b matching. We connect receiver PEs (left) to the blocks they need (middle). We connect each source PE (right) to the blocks it can send. For a matching, we need to consider only the middle and right column. Each block has to be incident to exactly one edge in the matching. We are minimizing the maximum number of edges any source PE is incident to.

all edges of one (given) of the two sets of vertices, that is, "left" or "right". All vertices v are incident to less or equal than b vertices of the matching. The vertices in the non-saturated group do not need to be incident to an edge in the matching.

For fixed b, a number of algorithms have been proposed which maximize the sum of edge weights of edges in the b-matching [6, 50, 51, 62]. Also, flow-networks have been used to find maximum matchings in bipartite graphs before [30, 57]. For our case, we need to minimize $b = \max(b_v)$ while ensuring that each vertex of the (w.l.o.g.) left side of the bipartite graph is matched. For this, we can use flow-networks. Ford and Fulkerson [37] describe an algorithm to compute the maximum flow in a network. The Ford-Fulkerson method works as follows: While there is an augmenting path from source to sink in the residual graph, add this path to the flow. They do not specify in which order to apply the augmenting paths.

The flow network we use to model a left-saturated minimal b-matching (see Figure 7.8b) has the following properties: All edges between the source and vertices of group A have a weight of exactly 1. All edges between a vertex of group A and a vertex of group B have a weight of exactly 1. There is exactly one edge between the source and each vertex in group A. This edge is the only incoming edge of these vertices. If we would set b to infinity, the incoming and outgoing flow through this vertex in a maximum flow is therefore exactly 1.



(a) as bipartite graph

(b) as flow problem

Figure 7.8.: (a) A-saturated minimal b-matching. All vertices in A are incident to exactly one edge in the matching. All vertices in B are incident to at most 2 vertices in the matching. There is no solution for $\max(b(V_B)) = 1$. (b) The same bipartite graph matching problem modelled as a network flow problem with source s and sink t. The edge capacities are annotated next to the edges. Solid edges have flow greater than zero. Dotted edges have a flow flow of zero.

This means, that exactly one outgoing edge of each vertex in A will be in the matching M if we set b large enough. Our task is of course to set b as small as possible while still having a flow of 1 through each vertex $v_A \in A$.

First, we maximize the flow with $b_{max} = 1$ (see Algorithm 11). If there exists a bipartite matching saturating all nodes in A, there must also be a flow in the network using all edges from s to nodes in A. As each of these edges has a weight of 1, the flow will have a capacity of exactly $|A|$. If a flow with a capacity of at least A exists in the network, the Ford-Fulkerson will find it [37]. Therefore, if the Ford-Fulkerson method does not find such a flow, there exists no bipartite matching saturating all nodes in A with the current b_{max}. We thus have to increase b_{max} by one and try to find a matching again. By using this iterative approach, we guarantee that no $b'_{max} < b_{max}$ exists for which a A-saturating matching is possible. We ruled out each b'_{max}. We do not need to reset the current flow and residual graph when increasing b_{max}. This is, because the Ford-Fulkerson method does not specify the order in which we have to apply the existing augmenting paths.

As there is exactly one edge from the source to each vertex in A with capacity of 1, a flow of 1 has to go over each of these edges. An augmenting path from source s to sink t will never

Algorithm 11 Ford-Fulkerson Method on a Bipartite Graph

let G be an adjacency array storing all edges between nodes in A and B as well as the flow in edge direction (0 or 1). We can use this data structure for the normal and residual Graph.
let $D[j]$ be an array storing the flow from each vertex $v_j \in B$ to the sink t.
let $F[i]$ be an array which stores the predecessor in of each node i in the current search tree
Ensure: $\forall i, j : G[(v_i, v_j)].\text{flow} + G[(v_j, v_i)].\text{flow} = 1$
$b \leftarrow 1$
for all $v \in A$ **do**
    let Q be an empty queue.
    Q.ENQUEUE(v)
    $F[v] \leftarrow NULL$
    repeat
        $v \leftarrow Q.dequeue()$
        Mark v as visited
        if $v \in B$ and $D[v] < b$ **then** ▷ Augmenting path found
            Increase the flow (stored in G) along all edges in the path (stored in F)
            Clear F
            $D[v] \leftarrow D[v] + 1$
            break
        end if
        for all neighbours w of v **do**
            if w not marked as visited **and** edge (v, w) has spare capacity **then**
                Q.ENQUEUE(w)
                $F[w] \leftarrow v$
            end if
        end for
    until Q.EMPTY()
    if no augmenting path found **then**
        $b \leftarrow b + 1$
        Restart loop iteration for the same start vertex v
    end if
end for

go over an edge (v_A, s) in the residual graph for all $v_A \in A$. We can therefore iterate over the vertices in A and initiate breadth first searches from there. An augmenting path from s to t will also never go over an edge (t, v_B) in the residual graph for all $v_B \in B$. We therefore can trim our Breadth First Search (BFS) search at t. $N_G(v)$ is the set of vertices adjacent to v in G, $N_R(v)$ the set of vertices adjacent to v in the residual graph.

The runtime of this algorithm is $O(|A| \cdot (|A| + |B) \cdot |E|)$ where $|A|$ and $|B|$ are the number of elements in the sets A and B respectively and $|E|$ is the number of edges. We can apply this algorithm to find solve the multi-sender h-relation when redistributing blocks (see Section 7.2.5). In this case, $|A|$ is the number of blocks that we need to transfer, $|B|$ is bound by the number of PEs, and $|E|$ is bound by the number of blocks we need to transfer times the number of copies per block.

7.3. A Probabilistic Approach

The algorithms described above are not trivial to implement. In this Section, we present a probabilistic approach to keeping read-only data redundantly in-memory across PEs. This approach is easier to implement. We have an object O (the MSA) of size L which we want to distribute over p PEs. We divide O into k blocks $O_0, \ldots, O_{k-1}$ of size L/k with $k \ll p$. Each PE i stores the block $O_{i \bmod k}$. If a PE wants to load block O_a, it will search for the next PE which stores O_a and fetch O_a from this PE (see Algorithm 12 and Figure 7.9). We could implement this using Remote Direct Memory Access (RDMA).

Algorithm 12 Get data block O_a on PE j

1: **for all** blocks O_a required on PE j **do**
2: $\quad i \leftarrow \text{argmin}_{i'}\{b = j + i' \mid b \bmod k = a \text{ and PE } b \text{ is alive}\}$
3: $\quad$ Get O_a from PE i
4: **end for**

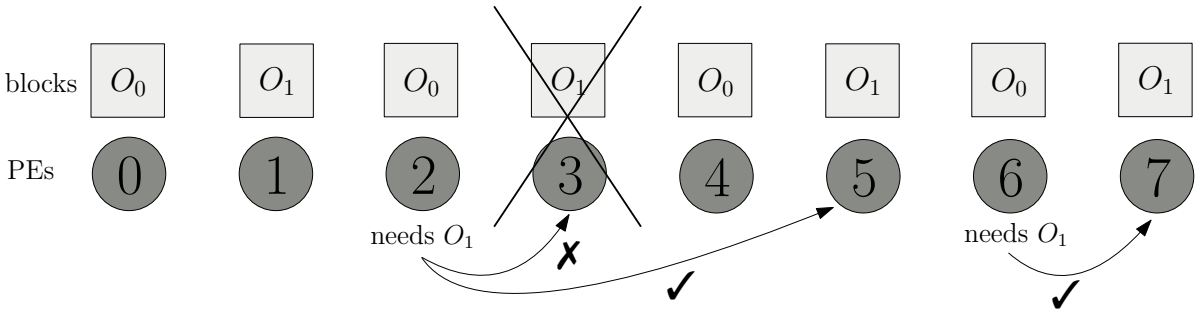


Figure 7.9.: Probabilistic redundant in-memory read-only storage. We divide the object (MSA) O into blocks. In this example we use two blocks. Each PE stores one block. If a PE wants to access a remote block, it requests it from the next alive PE which stores this block. PE 2 cannot request block O_1 from PE 3 (✗), because the latter is not alive. It can, however, request block O_1 from PE 5 (✓).

This approach always works if the number of failed nodes is less than p/k. For random failures the number of failures that we can tolerate without loosing data can be even higher.

We can use a random permutation to distribute the blocks onto the PEs. This will cause the worst-case to occur for a random input instead of occurring systematically.

We can adjust the formula given by Casanova *et al.* [17] for replicated computations to our situation. The *MNFTI* denotes the mean number of failures to interruption. This is the expected number of PEs that must fail such that for at least one block O_j there is no more copy available. A PE that failed once, cannot fail again. Let n_f be the number of failures. The formula for the case $g = 2$, that is, there are two replicas per block, is then:

$$\mathbb{E}(NFTI|n_f) = \begin{cases} 1, & \text{if } n_f = k \\ 1 + \frac{2k-2n_f}{2k-n_f}\mathbb{E}(NFTI|n_f+1), & \text{otherwise} \end{cases} \tag{7.2}$$

No closed formula is known. For a general formula with more than two replicas ($g > 2$), see Casanova *et al.* [17]. Let us give an example: Using 512 nodes and $g = 3$ copies per block, we have to set $k = 170$. Casanova *et al.* [17] calculated $MNFTI(k = 128, g = 3) = 75.9$. This means that, we can expect nearly 76 nodes to fail before we loose any blocks. Also, loosing blocks of the MSA is not catastrophic for RAxML-ng as we can always reload them from disk. Let us reconsider our example from Section 7.1.1. We calculated a memory requirement for the CLVs of 0.609 MiB site^{-1} per node. As we have a replication level of $g = 3$, each node has to additionally store 3 times as much of the MSA as it already stores. Assuming 4 bits per nucleotide, that is an additional 1.5 B per site. This is negligible compared to the memory used for CLVs and we could afford an even higher level of redundancy.

Part IV.

Summary

8. Discussion

We designed and implemented a fault tolerance scheme for RAxML-ng. It will automatically detect rank failures using ULFM, redistribute the computations to the surviving ranks and restart the tree search from the last checkpoint (see Section 6.3). To reduce the amount of work we loose in case of a rank failure, we increased the checkpointing-frequency. We also made checkpointing more fine-grained by separate checkpointing of the tree topology and evolutionary model parameters (see Section 6.2).

RAxML-ng now supports fault tolerance in the tree search mode, using multiple starting trees, and multiple partitions. RAxML-ng can handle multiple failures at once and multiple successive failures automatically. There is no limit on the number of failures that can occur simultaneously or sequentially. We also support mitigating failures which occur during the recovery of a previous failure. As recovery is a local operation, the subsequent collective operation will fail and restore the search state to the same mini-checkpoint. Further, we can tolerate failures during checkpointing and so-called mini-checkpointing. In contrast to the existing recovery scheme, a recovery is initiated automatically after a failure, that is, the user does not have to take any action.

We benchmark our algorithms for checkpointing and recovery (see Section 6.3). In our experiments, creating a checkpoint of the model parameters requires at most 72.0 ± 0.9 ms (400 ranks, 4,116 partitions). Creating a checkpoint of the tree topology requires at most 0.575 ± 0.006 ms (1,879 taxa). The overall runtime of RAxML-ng increases by a factor of 1.02 ± 0.02 when using the new checkpointing scheme and by a factor of 1.08 ± 0.07 when using the new checkpointing scheme and ULFM v4.0.2u1 as the MPI implementation. Restoring the search state after a failure requires at most 535 ± 19 ms. We simulated up to ten failures, which caused the overall runtime to increase by a factor of 1.3 ± 0.2.

To the best of our knowledge, this is the first implementation of automatic recovery after a rank failure in a phylogenetic tree search tool. We are now one step closer to preparing RAxML-ng for the upcoming challenges of exascale systems (see Chapter 5).

We also analysed the distribution of computations across ranks in RAxML-ng. We showed, that there is an imbalance of work of up to 30 % in our measurements (see Section 4.5.3). We also showed, that for some runs, a single rank requires the most time to process the current work package for 30 % of all work packages (see Section 4.5.4). We analysed the impact of site-repeats on the distribution of work. We found, that when disabling the site-repeats feature, the work is significantly more balanced compared to site-repeats enabled. Disabling site-repeats is not a solution. The omission of redundant computation we archive by using this feature induces a speedup of up to 417 % in our measurements (see Section 4.5.5). By using a load-balancer which takes into account the computational work saved using site-

repeats, we could therefore further reduce the overall runtime of RAxML-ng. We propose and implement a site-repeats aware load balancer by reducing the problem to a judicious hypergraph partitioning problem in another publication [8].

After a rank failure, we have to redistribute the work to the surviving ranks. Those ranks which we assign new sites to, have to load the part of the MSA they need for these new computations from disk. We described three approaches on how to eliminate this disk accesses by storing the data redundantly in the memory of the compute nodes (see Chapter 7). We presented algorithms to solve the multi-sender h-relation problem and the unilaterally-saturating b-matching problem. To the best of our knowledge, no research into low-latency access, redundant storage without replacement of failed ranks, multi-source h-relations, or unilaterally-saturating b-matchings has been published yet (see Section 7.2.2).

Making HPC Applications Fault-Tolerant

A complex program might invoke hundreds of MPI calls at different parts in the code. We have to check the return value of each one of them for a possible rank failure. If we detect a rank failure, we also have to handle it correctly. If these MPI calls are not abstracted away in a wrapper class (as for example `ParallelContext` in RAxML-ng), this is impractical [72]. Although the PMPI interface provides wrappers to all MPI functions [113] using them for fault tolerance would prevent us from using profiling tools. That is, because profiling tools also rely on the PMPI interface. This is therefore a stopgap solution at best. RAxML-ng encapsulates all its MPI calls in `ParallelContext` and we thus did not encounter this problem. This, again highlights the importance of good software engineering practices in scientific software.

We faced three main software engineering challenges while implementing fault-tolerance mechanisms. First, when a failure occurs, we have to jump to the recovery routine. This recovery routine will restore a consistent state. In RAxML-ng, we added the recovery routine to the `TreeInfo` class. This class wraps high-level routines for optimizing the evolutionary model, the branch lengths, and conducting SPR rounds. When recovering from a rank failure, the recovery routine will need access to some data we passed to it in its constructor. Some of these data was not intended to be valid for longer than the constructor call when we initially designed these constructors. We therefore either have to copy this data or change the constructor's interface, that is, require the parameters with which we call it to be valid over the entire runtime of the program. Secondly, in case of failure, there is a long jump in our code. We might detect a failure at every MPI call. We then have to first jump to the recovery routine and then back to the point in code we restart our algorithm from. We need to take care to not leak any memory or other resources here. We implemented these mechanisms using `C++` exceptions. In a `C` codebase, this will complicate the program design considerably [72]. Third, ULFM does not guarantee to report rank failures at the same MPI call on all ranks. This means, that different ranks might be in different lines of code when they get notified of the failure. This increases the logic needed to recover a consistent state – both in code and in the mind of the programmer. ULFM offers the operation `MPI_Comm_agree`, which enables us to synchronize the current knowledge about failures. `MPI_Comm_agree` conducts multiple

collective operations. ULFM refrains from reporting failures it noticed in the last collective operation until we call another MPI operation. We are therefore guaranteed to obtain the failure report at the same line of code on each rank; either during `MPI_Comm_agree` or at the following MPI call. `MPI_Comm_agree`, however, is slow and should be used sparingly.

9. Outlook

In Chapter 4 we showed the need for a load-balancer for phylogenetic inference algorithms which is aware of site-repeats. We propose and implement such an algorithm in another publication [8]. We still need to integrate this new load-balancer into RAxML-ng and evaluate the speedup we can obtain by using it.

In Chapter 8 we describe three algorithms for eliminating the disc-access during a recovery from a rank failure. Implementing and evaluating these algorithms constitutes future work. We expect these algorithms to speed-up the recovery from a rank failure even further. Once it is implemented, we can use the tree-based compression of an MSA (see Section 7.1) to save space when storing MSAs on disk and in databases, too.

Improving the Performance of Mini-Checkpointing

The mini-checkpointing algorithm we describe in Section 6.2.2 has a runtime of $\min(p, m) \cdot T_{bcast}(p)$. Here, p denotes the number of PEs and $T_{bcast}(p)$ denotes the time required for a single broadcast. We can speed up mini-checkpointing by limiting the number of replicas of each model to $f + 1$. We can then tolerate up to f simultaneous PE failures. By choosing f large enough, we can use statistics to show that our algorithm will still only fail with negligible probability. When using this approach, we need to broadcast each model to f other PEs. All PEs which need this model for their likelihood computations already have a consistent copy and we can thus save some messages. This mini-checkpointing scheme will scale with $\min(f, m) \cdot T_{bcast}(f)$. The expected number of simultaneous failures scales linearly with the number of PEs p (see Chapter 5). To keep the probability of successful program completion constant, we would therefore have to scale f linearly with p. Elnozahy and Plank [27] predict that we will need checkpoint algorithms whose runtime *decreases* as the number of PEs increases. They argue, that the expected time between two failures decreases with a growing number of PEs. Therefore, less time is available to complete the recovery, conduct useful computations, and then checkpoint the current state before the next PE failure occurs. The time taken for checkpointing and recovery consequently has to decrease as the number of nodes increases. This is not possible with (current) checkpoint/restart mechanisms, but we will still need them as backup for the more efficient recovery mechanisms [27].

Steps to a Production Ready RAxML-ng Extension

Some RAxML-ng features are not failure-tolerant yet. For example, currently only `-search` mode without bootstrap replicas is supported. Also, only fine-grained parallelization is

supported. We currently checkpoint the tree topology by copying it. This might be too slow for large trees, containing tens of thousands of nodes. An alternative would be to perform a full copy of the tree topology only at certain points in time and store all intermediate changes applied to the topology as rollback moves.

Improving the Frequency of Checkpointing

Although we improved the frequency of checkpointing considerably (see Figure 6.1), there is still room for improvement. We currently create mini-checkpoints after each call of an optimization routine for the tree topology, model parameters, or branch lengths. To increase the checkpoint frequency further, we need to implement fault-tolerant versions of the respective optimization algorithms, that is Newton-Raphson, Brent [15] and BFGS [36].

Numerical Instability of Allreduce Operations

Allreduce operations on floating-point values are numerically unstable. If the number of PEs which take part in the allreduce operation changes, the result might change as well. This is, because floating-point operations are only approximately associative and commutative. The changed order of operations will cause the small inaccuracies to pile up differently.

This has impacts on the reproducibility of tree searches. When no failure occurred, we can always reproduce a result by using the same number of PEs and the same random seed. If a failure occurred, RAxML-ng will conduct different allreduce operations with a different number of PEs. To reproduce this result, we have to simulate a failure at the exact same moment in the tree search. Implementing either a numerically stable allreduce operation or a failure-log to enhance reproducibility is subject of future work.

A. Appendix

A.1. Profiling RAxML-ng

A.2. Random Seeds for Profiling Runs

We list the random seeds we set via `-seed` in the profiling experiments (see Section 4.5.1ff) in Table A.1. We use `0` as the random seed in all other experiments.

Table A.1.: Random Seeds in the Profiling Experiments

dataset	ranks	nodes	random seed
PrumD6	200	10	1574530043
MisoD2a	20	1	1574443931
XiD4	160	8	1574528895
SongD1	80	4	1574463367
SongD1	400	20	1574549152
SongD1	360	18	1574547114
ShiD9	20	1	1574445908
ChenA4	160	8	1574484011

A.2.1. Absolute Difference of Time Required for Work and Communication

For each work and communication package, we measure how long each rank requires to process it. On each rank, we then compute the difference in time required on the fastest rank and this rank. We store these differences for all work packages in a histogram with exponentially growing bins (similar to the measurements done in Section 4.5.1).

To ascertain the time required on the fastest rank, we perform one additional `MPI_Allreduce` call after each work package and its associated `MPI_Allreduce` operation. We do not measure the time required for this operation. The bars depict the range between the 0.05 and 0.95 quantile of the time these code segments requires on a single rank (see Figure 4.2). Black dots indicate the median. The colours group together ranks on the same physical node.

There are no obvious differences between the distributions of different ranks. The variance is greater for work packages than it is for `MPI_Allreduce` calls. The fastest rank has a time difference to the fastest rank (itself) of 0 ns. To be able to use a logarithmic scale, we show this

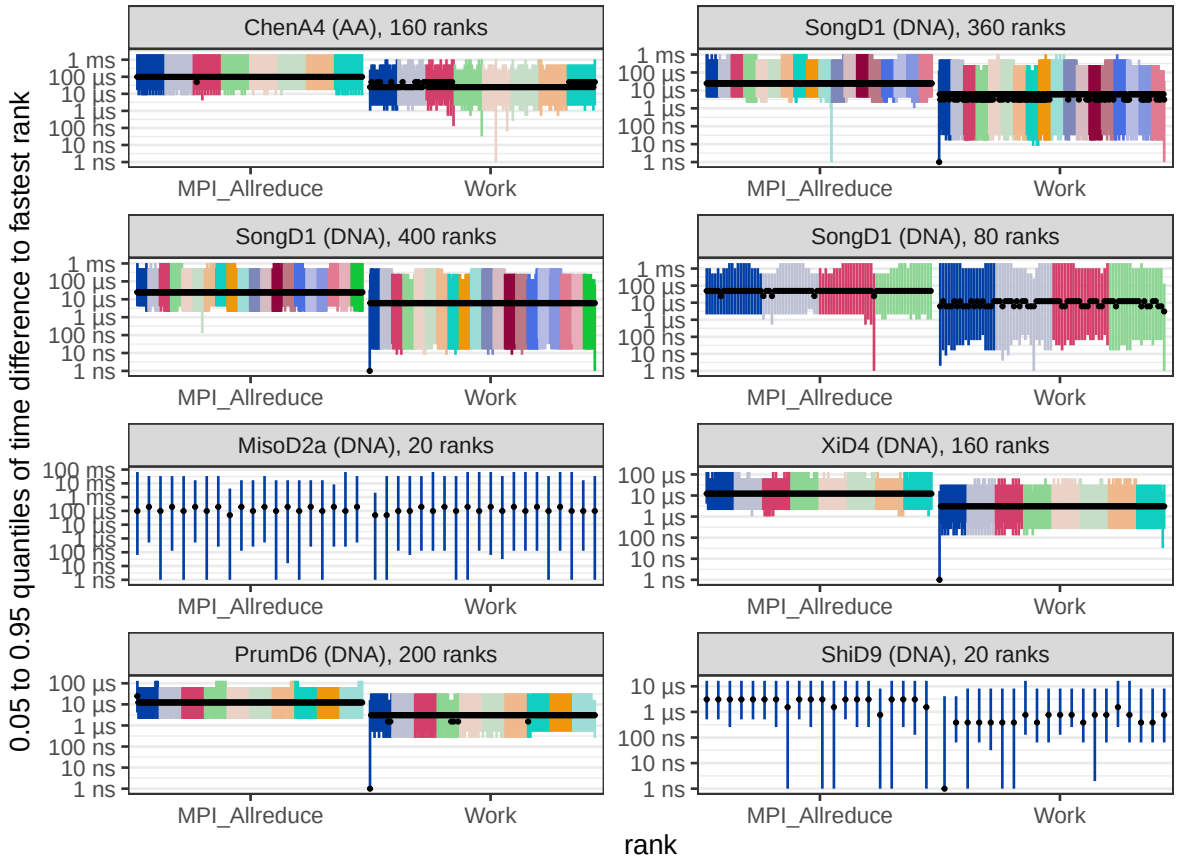


Figure A.1.: Relative time required for work and communication. Each bar depicts one rank. The colours group ranks by physical nodes. Each bar depicts the range between the 0.05 and 0.95 quantile of the difference between the time required on the fastest rank and this rank per work or communication package. Black dots indicate the median. We bin the values into exponentially growing bins ([1 to 2) ns, [2 to 4) ns, [4 to 8) ns, . . .). There are 20 ranks running on each node (one per physical CPU core). The fastest rank has a time difference to the fastest rank (itself) of 0 ns. To be able to use a logarithmic scale, we show this as 1 ns.

as 1 ns. Rank 0 is often the first one to finish. This is expected, as the load balancer assigns it the least work if the amount of work is not evenly dividable (see Section 4.5.1).

A.2.2. Relative Difference of Time Required for Work and Communication

.

We measure how much the time required to complete the same work or communication package differs between ranks. We measure the relative differences of the time required to process work packages and communication packages between ranks. To ascertain the time required by the fastest rank and the time required on average, we conduct one additional

`MPI_Allreduce` call after each work package and its associated `MPI_Allreduce` operation. We do not measure the time required for this operation. For simplicity, we call the relative differences Package-Specific Slowdon (PSS).

We chose to compare against the average instead of against the fastest rank, as there are outliers when looking at the minimum time (see Appendix A.2.1). Each rank computes $t_{rank}/t_{average}$ for each work or communication package. We choose to show only the range between the 0.05 and the 0.95 quantile in Section 4.5.2. We show the same data from the 0.01 to the 0.99 quantile in Figure A.2 and a summary in Table A.2. In Figure A.3, we show the data without removing any outliers.

Table A.2.: Summary on the relative differences of time required for work and communication. We show the overall minimum (min) and maximum (max) value across all ranks. We show the smallest value among the 0.01-quantiles of each rank in the column min(q01). Analogously, we show the maximum value among the 0.99-quantiles of each rank in the column max(q99).

type	dataset	ranks	min	max	min(q01)	max(q99)
AA	XiD4	160	0.14	11.0	0.87	1.15
DNA	SongD1	360	0.09	11.0	0.61	1.65
DNA	SongD1	400	0.09	11.0	0.57	1.75
DNA	SongD1	80	0.09	11.0	0.51	1.55
DNA	MisoD2a	20	0.09	11.0	0.69	1.15
DNA	XiD4	160	0.09	11.0	0.80	1.35
DNA	PrumD6	200	0.09	11.0	0.87	1.15
DNA	ShiD9	20	0.09	11.0	0.87	1.25

A.2.3. Imbalance of Work and Communication

We want to spend as much time working and as few time communicating as possible, as this increases the parallelization efficiency and decreased the overall runtime. If a rank finishes with its work package, it waits at the barrier of the following `MPI_Allreduce` for all the other ranks to finish their work packages. It therefore spends a higher portion of time inside `MPI_Allreduce` and less time outside of `MPI_Allreduce` calls than the other ranks. If different ranks spent different amounts of time working and communicating, this points to an imbalance in the distribution of work.

We measure the time inside `MPI_Allreduce` calls (communication) as well as the time outside them (work). If we write a checkpoint, we discard the current work package. We show the fraction of runtime spend doing work in Figure A.4. The work packages which are discarded do also not count towards the total runtime. All time that is not spent processing work packages is therefore spent waiting in `MPI_Allreduce` calls. We also compare the fraction of total runtime spend working with site-repeats turned on and off (see Figure A.5)

Table A.3.: Distribution of work: Maximum number of sites assigned to a single rank.

type	dataset	ranks	max sites per rank
AA	NagyA1	160	666
DNA	SongD1	80	9,331
DNA	SongD1	360	2,074
DNA	SongD1	400	1,867
DNA	MisoD2a	20	57,134
DNA	XiD4	160	1,037
DNA	PrumD6	200	1,133
DNA	ShiD9	20	666

The run on MisoD2a on 20 ranks has the most sites per rank (57,134; see Table A.3) and the highest work to communication ratio. The run ShiD9 on 20 ranks has the least sites per rank (666) but has a higher work to communication ratio than for example SongD1 on 400 ranks (1,867). As the 20 ranks of the ShiD9 all run on a single physical node, MPI can use shared memory and local sockets for communication. On the SongD1 run, MPI has to conduct communication between 20 nodes and 400 ranks. Some runs, for example on XiD4 with 1,037 sites per rank on 160 ranks or on PrumD6 with 1,133 sites per rank on 200 ranks have a work to runtime ratio of around 0.5. This indicates that we use too few sites per rank for RAxML-ng to efficiently parallelize the three search. This does affect the overall runtime, but not the load balance, which we want to investigate.

A.2.4. Number of MPI calls per Second

We measure the number of `MPI_Allreduce` calls per second (see Figure A.6). This gives us an indication on how many work packages RAxML-ng processes every second. We measure the most `MPI_Allreduce` calls per second, around 20,000, on the run on the ShiD9 dataset with 20 ranks (666 sites per rank, see Table A.3). When we, for example, evaluate the log-likelihood of a tree, we conduct one allreduce operation after we finished the local likelihood computation. If there are fewer sites per rank, we have to perform fewer local likelihood operations and therefore have to perform more allreduce operation per second.

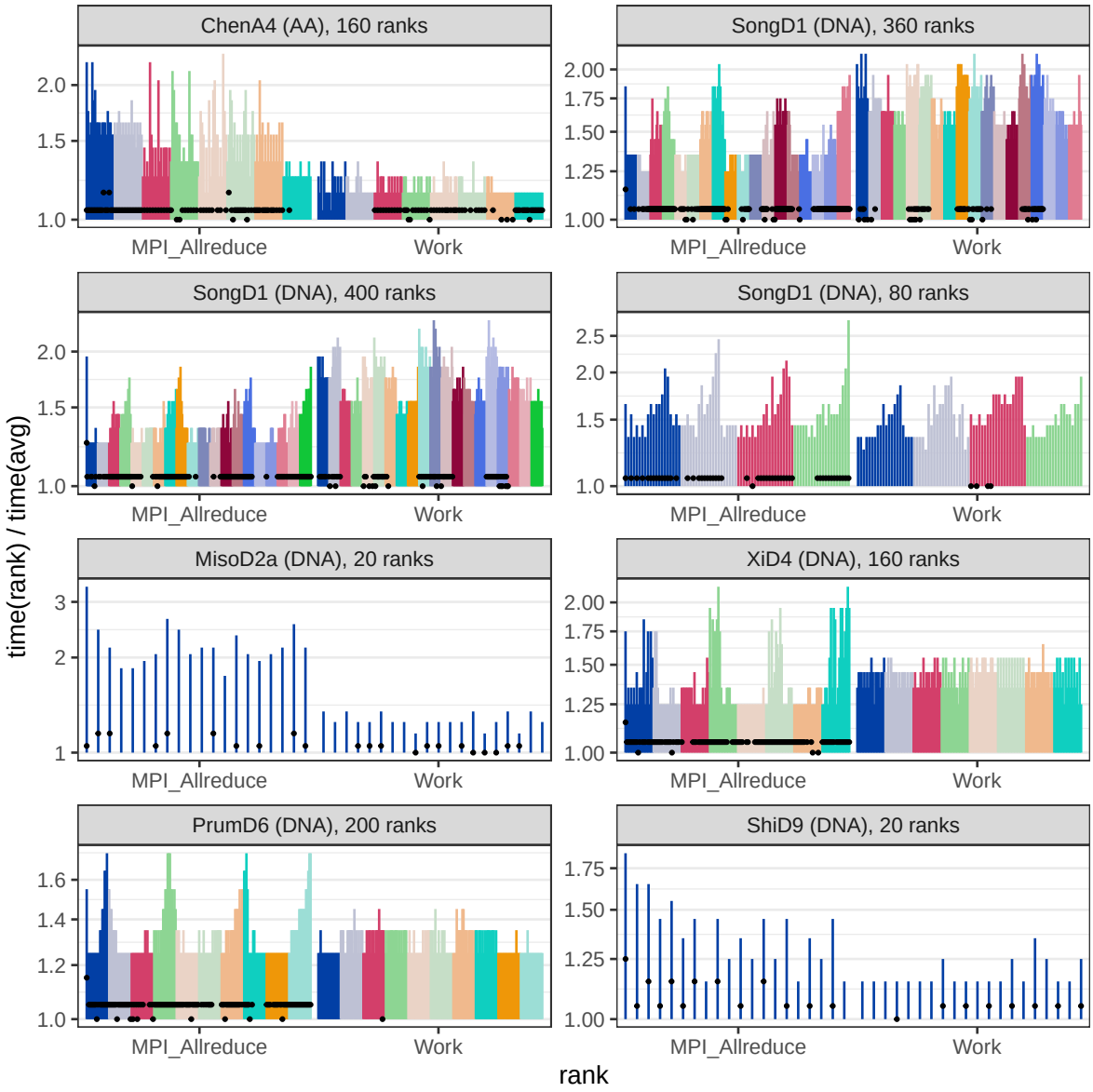


Figure A.2.: Relative differences of the time required for work and communication packages. Each rank computes $t_{rank}/t_{average}$ for each work and communication package. We call this the Package-Specific Slowdon (PSS). Each bar depicts the distribution of the PSSs of one rank. The colours group together ranks on the same node. The bar ranges from the 0.01 to the 0.99 quantile of the PSS. Black dots indicate the median of the PSS. For example: A bar ranging up to 1.6 means, that this rank required 60 % more time than the average rank for at least 1 % of the work packages. We truncate the y-axis below 1.

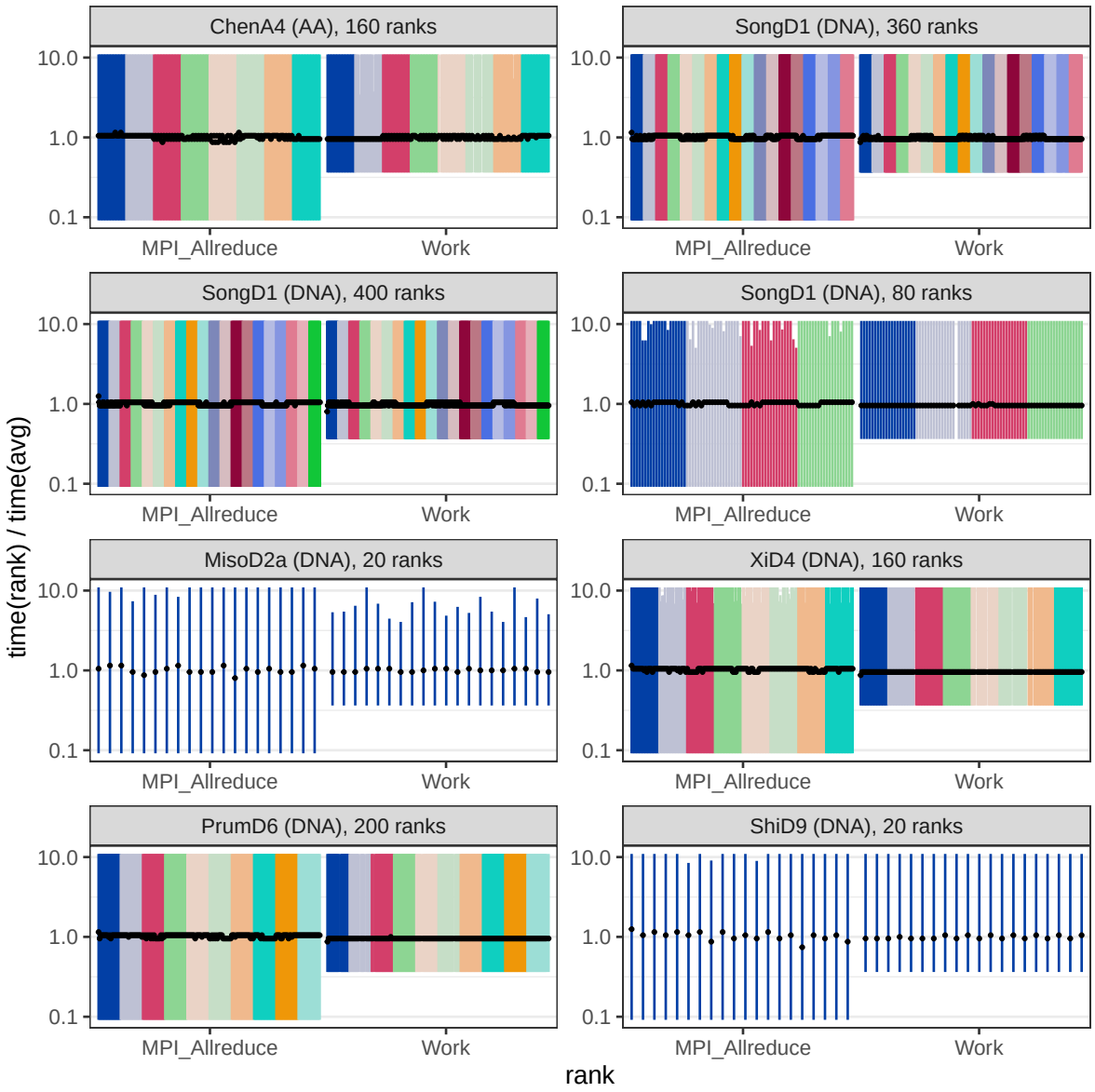


Figure A.3.: Relative differences of the time required for work and communication packages. Each rank computes $t_{rank}/t_{average}$ for each work and communication package. We call this the Package-Specific Slowdon (PSS). Each bar depicts the distribution of the PSSs on one rank. The colours group together ranks on the same node. The bar ranges from the the smallest to the largest measurement of the PSS. Black dots indicate the median of the PSS. For example: A bar ranging up to 1.6 means, that this rank required 60 % more time than the average rank for at least one of its work packages. The histogram implementation we use saves all values above 11 as 11 and all values below $\frac{1}{11}$ as $\frac{1}{11}$.

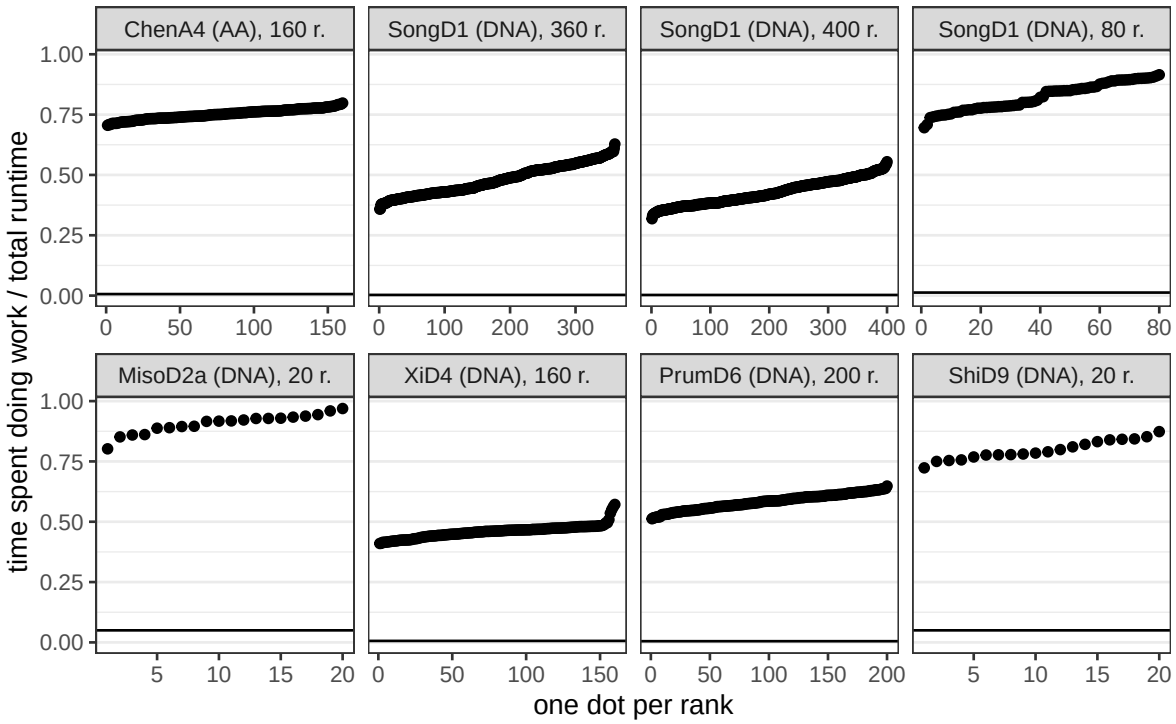


Figure A.4.: Fraction of total runtime spend outside of `MPI_Allreduce` calls ("working", bigger is better). A rank which finishes its current work package waits at the barrier of the following `MPI_Allreduce` for all the other ranks. It therefore spends a higher portion of time inside `MPI_Allreduce` and less time outside than the other ranks. If different ranks spent different amounts of time working and communicating, this points to an imbalance in the distribution of work.

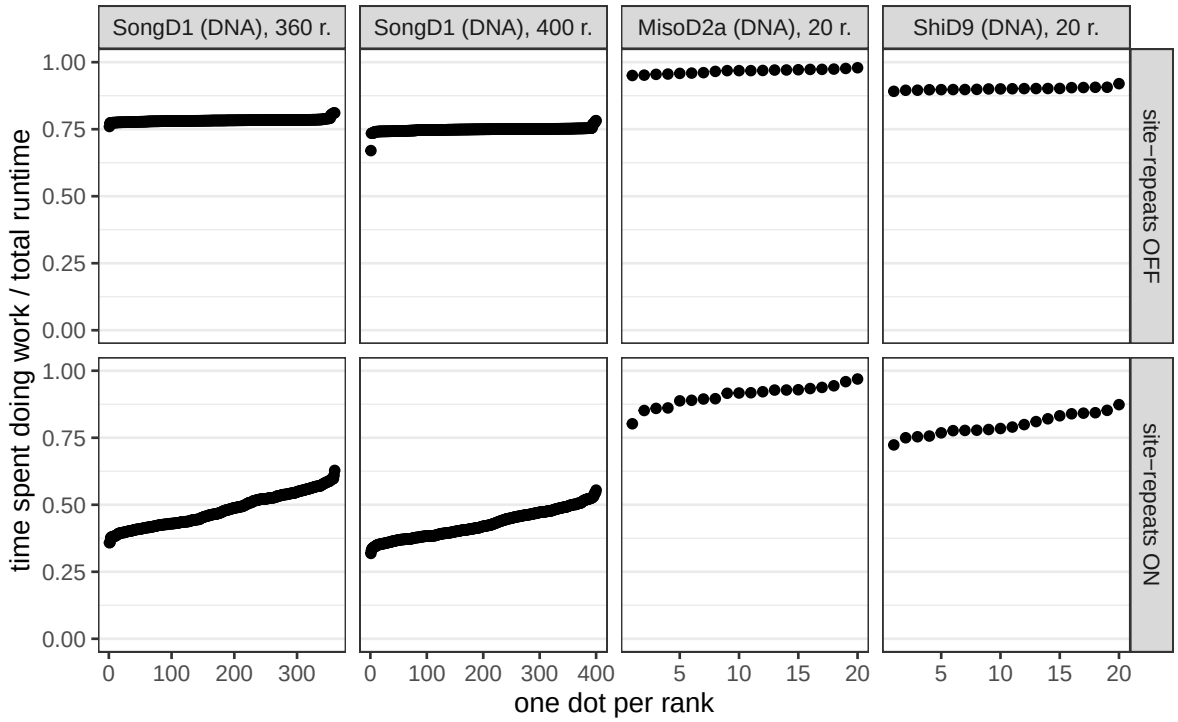


Figure A.5.: Influence of site-repeats on the time spend working vs communicating. Each rank computes how much time it spends outside of `MPI_Allreduce` calls ("working", more is better) vs inside `MPI_Allreduce` calls ("communicating", less is better). A rank which finishes its current work package waits at the barrier of the following `MPI_Allreduce` for all the other ranks. It therefore spends a higher portion of time inside `MPI_Allreduce` and less time outside than the other ranks. If different ranks spent different amounts of time working and communicating, this points to an imbalance in the distribution of work.

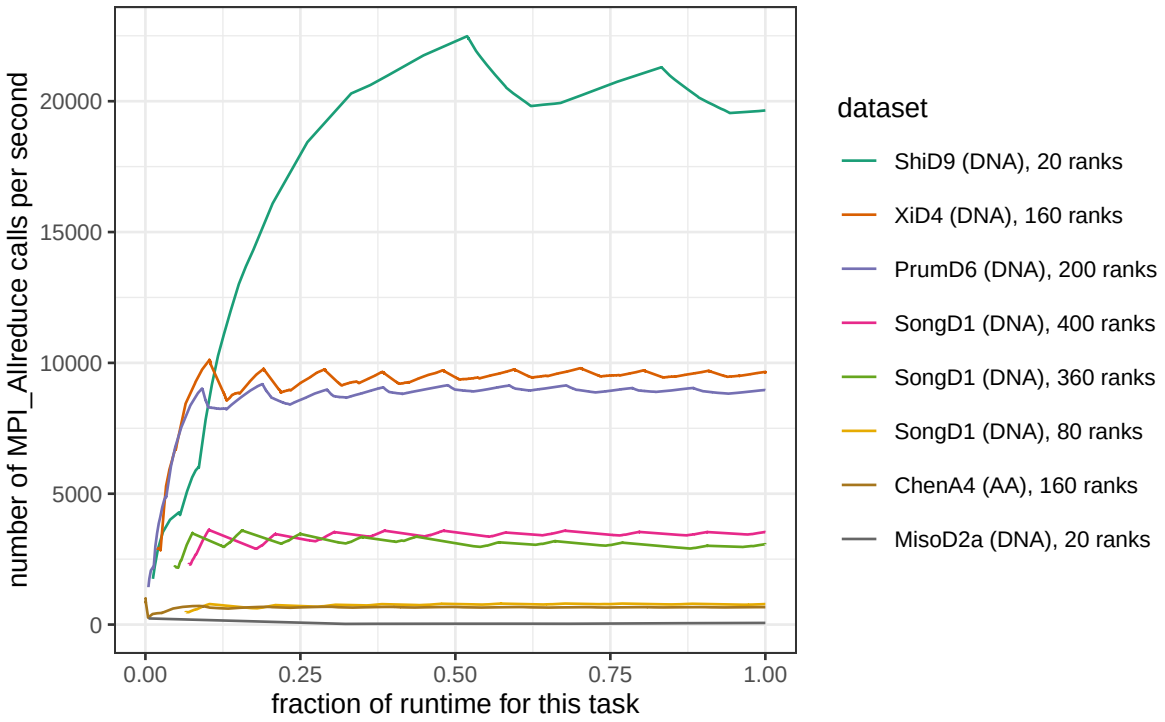


Figure A.6.: Number of `MPI_Allreduce` calls per second by different datasets. Measured over the complete runtime of one RAxML-ng run. See Section 4.4 for a description of the datasets. There are 20 ranks running on each node (one rank per physical CPU core).

A.3. File Sizes of MSA data

We list the sizes of the MSA datasets described in Section 4.4 in Table A.4. The biggest AA dataset is ChenA4 (100 MiB); the biggest DNA dataset is PeteD8 (500 MiB).

Table A.4.: File size of the MSA datasets. See Table 4.1 for a description of the datasets.

datatype	dataset	file size [MiB]
AA	NagyA1	10
AA	YangA8	46
AA	ChenA4	100
DNA	XiD4	11
DNA	ShiD9	16
DNA	SongD1	48
DNA	MisoD2a	171
DNA	PeteD8	500

A.4. Fault Tolerant RAxML-ng

A.4.1. Checkpointing the Tree

Mini-checkpointing consists of updating the model parameters and the tree topology. We evaluate the model parameter updates in Section 6.2.3. Updating the backup copy of the tree topology is a local operation. We therefore expect it to be faster than updating the model parameters, which requires possibly multiple broadcast operations. We measure all tree update operations in one tree search (190 to 14,203 depending on the dataset; see Figure A.7). We benchmark on eight different datasets with 37 to 174 taxa. The time required to update the backup copy of the tree correlates with the number of taxa in the tree (Pearson-correlation: $corr = 0.99$, $p < 2 \cdot 10^{-06}$).

A.4.2. Overhead of Restoration and Mini-Checkpointing

In Table A.5 we list the benchmark results on the recovery and mini-checkpoint algorithms discussed in Section 6.2.3 and Section 6.3.3.

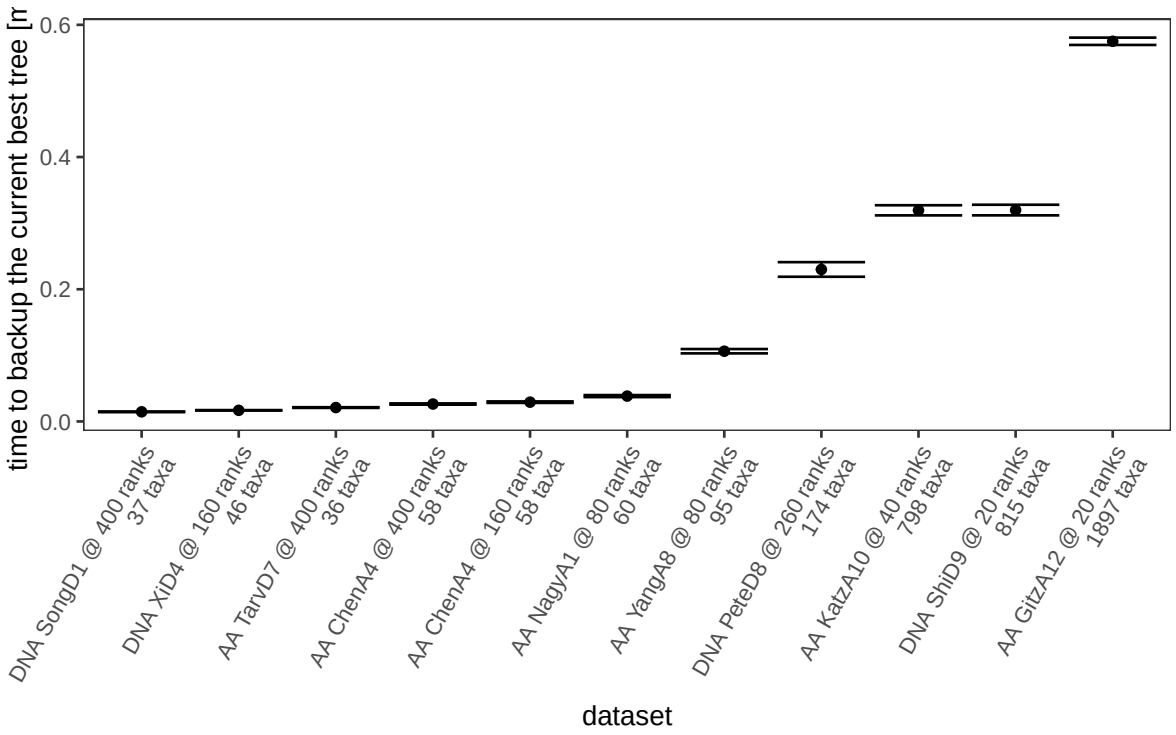


Figure A.7.: Time required for updating the backup copy of tree we need to restore in case of failure. We update the backup copy of the tree each time we improve the current tree topology. We measure all tree update operations in one tree search (190 to 14,203, depending on the dataset).

Table A.5.: Time required for mini-checkpoints and recovery after a failure. For YangA8, we measure seven recoverys, for all other datasets ten. We measure all model updates and tree recoveries in one run (at least 29).

dataset	ranks	taxa	sites	models	recovery [ms]	update models [ms]	update tree [ms]
NagyA1	80	60	172,073	594	90 ± 4	5.5 ± 0.01	0.039 ± 0.001
ChenA4	160	58	1,806,035	1	27 ± 3	0.3 ± 0.02	0.032 ± 0.001
YangA8	80	95	504,850	1,122	263 ± 11	11.1 ± 0.2	0.099 ± 0.002
SongD1	400	37	1,338,678	1	34 ± 5	0.24 ± 0.01	0.015 ± 0.001
XiD4	160	46	239,763	1	13 ± 1	0.21 ± 0.01	0.017 ± 0.001
TarvD7	400	36	21,410,970	1	46 ± 4	0.66 ± 0.06	0.025 ± 0.001
PeteD8	260	174	3,011,099	4116	535 ± 19	72.0 ± 0.9	0.21 ± 0.01
KatzA10	40	798	34,991	1		0.8 ± 0.1	0.310 ± 0.008
GitzA12	20	1,897	18,328	1		1.8 ± 0.8	0.575 ± 0.006
ShiD9	20	815	20,364	29		0.71 ± 0.02	0.320 ± 0.008

A.5. Additional image sources

The tree, list, and disk icon were made by Freepik from `www.flaticon.com`.
The model icon was made by Eucalyp from `www.flaticon.com`.

Acronyms

AA Amino Acid. 7, 19, 21, 49, 52, 94, 97, 98

ABFT Algorithm Based Fault Tolerance. 32

BFS Breadth First Search. 71

CLV Conditional Likelihood Vector. 32, 48–50, 56, 66, 74

DFS Depth First Search. 57

DNA Deoxyribonucleic Acid. 5–7, 19, 21, 49, 52, 94, 97, 98

HPC High Performance Computing. 2, 3, 14, 31, 32, 35, 36, 63, 77

IUPAC International Union of Pure and Applied Chemistry. 52

LLH Log-Likelihood. 16

MPI Message Passing Interface. v, 4, 14, 16, 17, 19, 21, 31–37, 40, 45–47, 77, 94, 97

MPICH MPI Chameleon. 33

MSA Multiple Sequence Alignment. iii, v, vi, viii, 7, 14, 15, 38, 40, 48–57, 59–63, 66, 73, 74, 76, 78, 97, 98

MSB Most Significant Bit. 58

MTBF Mean Time Between Failure. 31

MTTF Mean Time To Failure. viii, 31, 38

P2P Peer-to-Peer. 63

PE Processing Element. iii, 3, 4, 15, 16, 31, 33, 35, 52, 55, 56, 60, 62–69, 71, 73, 78, 79

PMPI Profile Layer of MPI. 17, 77

PSS Package-Specific Slowdon. 21, 23–25, 92, 93

RAID Redundant Array of Inexpensive Disks. 63

RDMA Remote Direct Memory Access. 73

SCC Steinbruch Center for Computing. 18

SIMD Single Instruction Multiple Data stream. 19, 63

SPR Subtree Pruning and Regrafting. vi, 9–11, 19, 22, 38, 39, 41, 43, 44, 77

ULFM User Level Failure Mitigation. vi, viii, 33–37, 39, 40, 46, 47, 76, 77

Glossary

Multiple Sequence Alignment A set of amino acid or DNA sequences which are aligned to each other. Sequence alignment has the goal to insert gaps of varying lengths into the sequences such that those regions which share a common evolutionary history are aligned to each other. On possible heuristic for computing an MSA is to minimize the number of differences between the aligned sites of the MSA [18].

User Level Failure Mitigation A MPI implementation which supports detecting and mitigating rank failures. See Section 5.2.

Conditional Likelihood Vector A cache for partial likelihood computations. The majority of the memory used by RAxML-ng stores CLVs. See Section 2.2.1.

Subtree Pruning and Regrafting A method for optimizing the topology of a phylogenetic tree. It consists of removing (pruning) a subtree from the currently best scoring tree and reinserting (regrafting) it into a neighbouring branch. See Section 2.2.2.1.

CPUs, Ranks, Nodes, and PEs See Section 1.4

(Log-) Likelihood score of a tree The probability of seeing the sequence data given the tree topology, branch lengths, and evolutionary model. *Not* the probability that this is the correct tree. Section 2.2.1.

Bibliography

[1] Micah Adler, John W. Byers, and Richard M. Karp. "Scheduling parallel communication: The *h*-relation problem". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995, pp. 1–20. DOI: 10.1007/3-540-60246-1_109.

[2] Nikolaos Alachiotis and Alexandros Stamatakis. "A Generic and Versatile Architecture for Inference of Evolutionary Trees under Maximum Likelihood". In: *Conference Record of the 44th IEEE Asilomar Conference on Signals, Systems and Computers (ASILOMAR) Studies*. Nov. 2010.

[3] Md Mohsin Ali et al. "Complex scientific applications made fault-tolerant with the sparse grid combination technique". In: *The International Journal of High Performance Computing Applications* 30.3 (July 2016), pp. 335–359. DOI: 10.1177/1094342015628056.

[4] C. Ané, O. Eulenstein, and R. Piaggio-Talice. *Phylogenetic compression and model selection: an improved encoding scheme*. Tech. rep. 2005.

[5] Cécile Ané and Michael J. Sanderson. "Missing the Forest for the Trees: Phylogenetic Compression and Its Implications for Inferring Complex Evolutionary Histories". In: *Systematic Biology* 54.1 (Feb. 2005). Ed. by Mike Steel, pp. 146–157. DOI: 10.1080/10635150590905984.

[6] Richard P. Anstee. "A polynomial algorithm for *b*-matchings: An alternative approach". In: *Information Processing Letters* 24.3 (Feb. 1987), pp. 153–157. DOI: 10.1016/0020-0190(87)90178-5.

[7] Rizwan A. Ashraf, Saurabh Hukerikar, and Christian Engelmann. "Shrink or Substitute: Handling Process Failures in HPC Systems using In-situ Recovery". In: (Jan. 14, 2018). arXiv: 1801.04523v1 [cs.DC].

[8] Ivo Baar et al. "Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning". In: (Mar. 2019). DOI: 10.1101/579318.

[9] David Bader, Moret Bernard, and Lisa Vawter. "Industrial applications of high-performance computing for phylogeny reconstruction". In: *Proc. SPIE 4528, Commercial Applications for High-Performance Computing*. 27. 2001. DOI: 10.1117/12.434868.

[10] Charles F. Baer, Michael M. Miyamoto, and Dee R. Denver. "Mutation rate variation in multicellular eukaryotes: causes and consequences". In: *Nature Reviews Genetics* 8.8 (Aug. 2007), pp. 619–631. DOI: 10.1038/nrg2158.

[11] Wesley Bland et al. "Post-failure recovery of MPI communication capability". In: *The International Journal of High Performance Computing Applications* 27.3 (June 2013), pp. 244–254. DOI: 10.1177/1094342013488238.

[12] David Boehme et al. "The Case for a Common Instrumentation Interface for HPC Codes". In: *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*. IEEE, Nov. 2019. DOI: 10.1109/protools49597.2019.00010.

[13] George Bosilca. *Post pbSToy94RhI/xUrFBx_1DAAJ on the ULFM mailing list*. Jan. 2020.

[14] George Bosilca et al. "Algorithmic Based Fault Tolerance Applied to High Performance Computing". In: (June 19, 2008). arXiv: 0806.3121v1 [cs.DC].

[15] R. P. Brent. *Algorithms for minimization without derivatives*. Mineola, N.Y: Dover Publications, 2002. ISBN: 9780486419985.

[16] Franck Cappello et al. "toward exascale resilience: 2014 update". In: *Supercomputing Frontiers and Innovations* 1.1 (Sept. 2014). DOI: 10.14529/jsfi140101.

[17] Henri Casanova, Frédéric Vivien, and Dounia Zaidouni. "Using Replication for Resilience on Exascale Systems". In: *Computer Communications and Networks*. Springer International Publishing, 2015, pp. 229–278. DOI: 10.1007/978-3-319-20943-2_4.

[18] Maria Chatzou et al. "Multiple sequence alignment modeling: methods and applications". In: *Briefings in Bioinformatics* 17.6 (Nov. 2015), pp. 1009–1023. DOI: 10.1093/bib/bbv099.

[19] Meng-Yun Chen, Dan Liang, and Peng Zhang. "Selecting Question-Specific Genes to Reduce Incongruence in Phylogenomics: A Case Study of Jawed Vertebrate Backbone Phylogeny". In: *Systematic Biology* 64.6 (Aug. 2015), pp. 1104–1120. DOI: 10.1093/sysbio/syv059.

[20] Zizhong Chen and Jack Dongarra. "A Scalable Checkpoint Encoding Algorithm for Diskless Checkpointing". In: *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, Dec. 2008. DOI: 10.1109/hase.2008.13.

[21] Tzi-Cker Chiueh and Peitao Deng. "Evaluation of checkpoint mechanisms for massively parallel machines". In: *Proceedings of Annual Symposium on Fault Tolerant Computing*. IEEE Comput. Soc. Press, 1996. DOI: 10.1109/ftcs.1996.534622.

[22] B. Chor and T. Tuller. "Maximum likelihood of evolutionary trees: hardness and approximation". In: *Bioinformatics* 21.Suppl 1 (June 2005), pp. i97–i106. DOI: 10.1093/bioinformatics/bti1027.

[23] Charles Darwin. *On the origin of species by means of natural selection*. John Murray, Nov. 1859.

[24] Alexandros G. Dimakis et al. "Network Coding for Distributed Storage Systems". In: *IEEE Transactions on Information Theory* 56.9 (Sept. 2010), pp. 4539–4551. DOI: 10.1109/tit.2010.2054295.

[25] Jack Dongarra, Thomas Herault, and Yves Robert. *Fault tolerance techniques for high-performance computing*. https://www.netlib.org/lapack/lawnspdf/lawn289.pdf. 2015.

[26] Richard Durbin, Sean R. Eddy, and Anders Krogh. *Biological Sequence Analysis*. Cambridge University Press, 1998. 370 pp. ISBN: 0521629713. URL: https://www.ebook.de/de/product/3242471/richard_durbin_sean_r_eddy_anders_krogh_biological_sequence_analysis.html.

[27] E. N. Elnozahy and J. S. Plank. "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery". In: *IEEE Transactions on Dependable and Secure Computing* 1.2 (Apr. 2004), pp. 97–108. ISSN: 1941-0018. DOI: 10.1109/TDSC.2004.15.

[28] Encycloaedia Britannica. *Phylogeny*. Ed. by John L. Gittleman. Sept. 13, 2016. URL: https://www.britannica.com/science/phylogeny.

[29] Christian Engelmann and Al Geist. "A Diskless Checkpointing Algorithm for Super-Scale Architectures Applied to the Fast Fourier Transform". In: *Proceedings of the 1st International Workshop on Challenges of Large Applications in Distributed Environments*. CLADE '03. USA: IEEE Computer Society, 2003, p. 47. ISBN: 0769519849.

[30] Shimon Even and R. Endre Tarjan. "Network Flow and Testing Graph Connectivity". In: *SIAM Journal on Computing* 4.4 (Dec. 1975), pp. 507–518. DOI: 10.1137/0204043.

[31] J. Felsenstein. "Maximum Likelihood and Minimum-Steps Methods for Estimating Evolutionary Trees from Data on Discrete Characters". In: *Systematic Biology* 22.3 (Sept. 1973), pp. 240–249. DOI: 10.1093/sysbio/22.3.240.

[32] Joseph Felsenstein. "Evolutionary trees from DNA sequences: A maximum likelihood approach". In: *Journal of Molecular Evolution* 17.6 (Nov. 1981), pp. 368–376. DOI: 10.1007/bf01734359.

[33] Joseph Felsenstein. "The Number of Evolutionary Trees". In: *Systematic Zoology* 27.1 (Mar. 1978), p. 27. DOI: 10.2307/2412810.

[34] José Luis Fernández-García. "Phylogenetics for Wildlife Conservation". In: *Phylogenetics*. InTech, Sept. 2017. DOI: 10.5772/intechopen.69240.

[35] Walter M. Fitch. "Toward Defining the Course of Evolution: Minimum Change for a Specific Tree Topology". In: *Systematic Zoology* 20.4 (Dec. 1971), pp. 406–416. DOI: 10.2307/2412116.

[36] R. Fletcher. *Practical methods of optimization*. Chichester New York: Wiley, 1987. ISBN: 9780471915478.

[37] L. R. Ford. *Flows in networks*. Princeton, N.J. Woodstock: Princeton University Press, 2010. ISBN: 9780691146676.

[38] G. Fox et al. "The phylogeny of prokaryotes". In: *Science* 209.4455 (July 1980), pp. 457–463. DOI: 10.1126/science.6771870.

[39] Vincent W. Freeh et al. "Just-in-time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs". In: *Journal of Parallel and Distributed Computing* 68.9 (Sept. 2008), pp. 1175–1185. DOI: 10.1016/j.jpdc.2008.04.007.

[40] Sunil P. Gavaskar and Ch D. V. Subbarao. "a survey of distributed fault tolerance strategies". In: *International Journal of Advanced Research in Computer and Communication Engineering* 2.11 (Nov. 2013). ISSN: 2278-1021.

[41] GeneBank. *GenBank and WGS Statistics*. Apr. 1, 2020. URL: https://www.ncbi.nlm.nih.gov/genbank/statistics/.

[42] Matthew A. Gitzendanner et al. "Plastid phylogenomic analysis of green plants: A billion years of evolutionary history". In: *American Journal of Botany* 105.3 (Mar. 2018), pp. 291–301. DOI: 10.1002/ajb2.1048.

[43] Toni I. Gossmann et al. "Ice-Age Climate Adaptations Trap the Alpine Marmot in a State of Low Genetic Diversity". In: *Current Biology* 29.10 (May 2019), pp. 1712–1720. DOI: 10.1016/j.cub.2019.04.020.

[44] William Gropp. "MPICH2: A New Start for MPI Implementations". In: *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg, 7: Springer-Verlag, 2002.

[45] Stéphane Guindon and Olivier Gascuel. "A Simple, Fast, and Accurate Algorithm to Estimate Large Phylogenies by Maximum Likelihood". In: *Systematic Biology* 52.5 (Oct. 2003). Ed. by Bruce Rannala, pp. 696–704. DOI: 10.1080/10635150390235520.

[46] Saurabh Gupta et al. "Failures in large scale systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2017. DOI: 10.1145/3126908.3126937.

[47] Paul H. Hargrove and Jason C. Duell. "Berkeley lab checkpoint/restart (BLCR) for Linux clusters". In: *Journal of Physics: Conference Series* 46 (Sept. 2006), pp. 494–499. DOI: 10.1088/1742-6596/46/1/067.

[48] J. A. Hartigan. "Minimum Mutation Fits to a Given Tree". In: *Biometrics* 29.1 (Mar. 1973), p. 53. DOI: 10.2307/2529676.

[49] Octavio Herrera-Ruiz and Taieb Znati. "Performance of redundancy methods in P2P networks under churn". In: *2012 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, Jan. 2012. DOI: 10.1109/iccnc.2012.6167437.

[50] Bert Huang and Tony Jebara. "Fast *b*-matching via Sufficient Selection Belief Propagation". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, 2011, pp. 361–369. URL: http://proceedings.mlr.press/v15/huang11a.html.

[51] Bert Huang and Tony Jebara. “Loopy Belief Propagation for Bipartite Maximum Weight *b*-Matching”. In: *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics*. Ed. by Marina Meila and Xiaotong Shen. Vol. 2. Proceedings of Machine Learning Research. San Juan, Puerto Rico: PMLR, 2007, pp. 195–202. URL: http://proceedings.mlr.press/v2/huang07a.html.

[52] David A. Huffman. “A method for the construction of minimum-redundancy codes”. In: *Resonance* 11.2 (Feb. 2006), pp. 91–99. DOI: 10.1007/bf02837279.

[53] IUPAC. *Nucleotide Codes*. URL: https://www.bioinformatics.org/sms/iupac.html.

[54] Fernando Izquierdo-Carrasco, Stephen A. Smith, and Alexandros Stamatakis. “Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees”. In: *BMC Bioinformatics* 12.1 (Dec. 2011). DOI: 10.1186/1471-2105-12-470.

[55] E. D. Jarvis et al. “Whole-genome analyses resolve early branches in the tree of life of modern birds”. In: *Science* 346.6215 (Dec. 2014), pp. 1320–1331. DOI: 10.1126/science.1253451.

[56] Thomas H. Jukes and Charles R. Cantor. “Evolution of Protein Molecules”. In: *Mammalian Protein Metabolism*. Elsevier, 1969, pp. 21–132. DOI: 10.1016/b978-1-4832-3211-9.50009-7.

[57] T. Kameda and I. Munro. “A O(|V|·|E|) algorithm for maximum matching of graphs”. In: *Computing* 12.1 (Mar. 1974), pp. 91–98. DOI: 10.1007/bf02239502.

[58] Laura A. Katz and Jessica R. Grant. “Taxon-Rich Phylogenomic Analyses Resolve the Eukaryotic Tree of Life and Reveal the Power of Subsampling by Sites”. In: *Systematic Biology* 64.3 (Dec. 2014), pp. 406–415. DOI: 10.1093/sysbio/syu126.

[59] Michael Kerrisk. *Manual Page of Linux’s `kill`*. http://man7.org/linux/man-pages/man1/kill.1.html.

[60] Michael Kerrisk. *Manual Page of Linux’s `raise`*. http://man7.org/linux/man-pages/man3/raise.3.html.

[61] Michael Kerrisk. *Manual Pages of Linux’s Signals*. http://man7.org/linux/man-pages/man7/signal.7.html.

[62] Arif Khan et al. “Efficient Approximation Algorithms for Weighted *b*-Matching”. In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S593–S619. DOI: 10.1137/15m1026304.

[63] Andreas Knüpfer et al. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir”. In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31476-6.

[64] Y. Kodama, M. Shumway, and R. Leinonen. “The sequence read archive: explosive growth of sequencing data”. In: *Nucleic Acids Research* 40.D1 (Oct. 2011), pp. D54–D56. DOI: 10.1093/nar/gkr854.

[65] Nils Kohl et al. “A Scalable and Extensible Checkpointing Scheme for Massively Parallel Simulations”. In: (Aug. 28, 2017). arXiv: 1708.08286v2 [cs.DC].

[66] Dénes König. “Gráfok és alkalmazásuk a determinánsok és a halmazok elméletére”. In: *Matematikai és Természettudományi Értesít.* 1916. Chap. 34, pp. 104–119.

[67] B. Korber. “Timing the Ancestor of the HIV-1 Pandemic Strains”. In: *Science* 288.5472 (June 2000), pp. 1789–1796. DOI: 10.1126/science.288.5472.1789.

[68] Alexey Kozlov. “Models, Optimizations, and Tools forLarge-Scale Phylogenetic Inference,Handling Sequence Uncertainty,and Taxonomic Validation”. PhD thesis. Karlsruher Institut für Technologie (KIT), Jan. 17, 2018.

[69] Alexey M. Kozlov, Andre J. Aberer, and Alexandros Stamatakis. “ExaML version 3 a tool for phylogenomic analyses on supercomputers”. In: *Bioinformatics* 31.15 (Mar. 2015), pp. 2577–2579. DOI: 10.1093/bioinformatics/btv184.

[70] Alexey M. Kozlov et al. “RAxML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference”. In: *Bioinformatics* 35.21 (May 2019). Ed. by Jonathan Wren, pp. 4453–4455. DOI: 10.1093/bioinformatics/btz305.

[71] M. K. Kuhner and Joe Felsenstein. “A simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates.” In: *Molecular Biology and Evolution* (May 1994). DOI: 10.1093/oxfordjournals.molbev.a040126.

[72] Ignacio Laguna et al. “Evaluating and extending user-level fault tolerance in MPI applications”. In: *The International Journal of High Performance Computing Applications* 30.3 (July 2016), pp. 305–319. DOI: 10.1177/1094342015623623.

[73] Charles H. Langley and Walter M. Fitch. “An examination of the constancy of the rate of molecular evolution”. In: *Journal of Molecular Evolution* 3.3 (Sept. 1974), pp. 161–177. DOI: 10.1007/bf01797451.

[74] Charng-Da Lu. “Failure Data Analysis of HPC Systems”. In: (Feb. 20, 2013). arXiv: 1302.4779v1 [cs.DC].

[75] Bunjamin Memishi et al. “Fault Tolerance in MapReduce: A Survey”. In: *Computer Communications and Networks*. Springer International Publishing, 2016, pp. 205–240. DOI: 10.1007/978-3-319-44881-7_11.

[76] Message Passing Interface Forum. *ULFM Specification*. http://fault-tolerance.org/wp-content/uploads/2012/10/20170221-ft.pdf. Feb. 2017.

[77] B. Misof et al. “Phylogenomics resolves the timing and pattern of insect evolution”. In: *Science* 346.6210 (Nov. 2014), pp. 763–767. DOI: 10.1126/science.1257570.

[78] László G. Nagy et al. “Latent homology and convergent regulatory evolution underlies the repeated emergence of yeasts”. In: *Nature Communications* 5.1 (July 2014). DOI: 10.1038/ncomms5471.

[79] Lam-Tung Nguyen et al. “IQ-TREE: A Fast and Effective Stochastic Algorithm for Estimating Maximum-Likelihood Phylogenies”. In: *Molecular Biology and Evolution* 32.1 (Nov. 2015), pp. 268–274. DOI: 10.1093/molbev/msu300.

[80] Michael Obersteiner et al. “A highly scalable, algorithm-based fault-tolerant solver for gyrokinetic plasma simulations”. In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems - ScalA ’17*. ACM Press, 2017. DOI: 10.1145/3148226.3148229.

[81] David A. Patterson et al. “Introduction to redundant arrays of inexpensive disks (RAID)”. In: *Digest of Papers. COMPCON Spring 89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage*. IEEE Comput. Soc. Press, Mar. 3, 1989. DOI: 10.1109/cmpcon.1989.301912.

[82] Ralph S. Peters et al. “Evolutionary History of the Hymenoptera”. In: *Current Biology* 27.7 (Apr. 2017), pp. 1013–1018. DOI: 10.1016/j.cub.2017.01.027.

[83] Wayne Pfeiffer and Alexandros Stamatakis. “Hybrid MPI/Pthreads parallelization of the RAxML phylogenetics code”. In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, Apr. 2010. DOI: 10.1109/ipdpsw.2010.5470900.

[84] J. S. Plank. “Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques”. In: *Proceedings 15th Symposium on Reliable Distributed Systems*. IEEE Comput. Soc. Press, 1996. DOI: 10.1109/reldis.1996.559700.

[85] J. S. Plank, Kai Li, and M. A. Puening. “Diskless checkpointing”. In: *IEEE Transactions on Parallel and Distributed Systems* 9.10 (1998), pp. 972–986. DOI: 10.1109/71.730527.

[86] James S. Plank. *A Tutorial on Reed-Solomon Coding for Fault-Tolerance inRAID-like Systems*. Tech. rep. University of Tennessee, Department of Computer Science, 107 Ayres Hall, Knoxville, TN 37996, U.S.A.(email: plank@cs.utk.edu): University of Tennessee, Nov. 1997.

[87] Morgan N. Price, Paramvir S. Dehal, and Adam P. Arkin. “FastTree 2 – Approximately Maximum-Likelihood Trees for Large Alignments”. In: *PLoS ONE* 5.3 (Mar. 2010). Ed. by Art F. Y. Poon, e9490. DOI: 10.1371/journal.pone.0009490.

[88] Richard O. Prum et al. “A comprehensive phylogeny of birds (Aves) using targeted next-generation DNA sequencing”. In: *Nature* 526.7574 (Oct. 2015), pp. 569–573. DOI: 10.1038/nature15697.

[89] Fatemeh Rajabi-Alni, Alireza Bagheri, and Behrouz Minaei-Bidgoli. “An $O(n^3)$ time algorithm for the maximum weight b-matching problem on bipartite graphs”. In: (Oct. 13, 2014). arXiv: 1410.3408v2 [cs.DS].

[90] Eric Roman. *A Survey of Checkpoint/Restart Implementations*. Tech. rep. Lawrence Berkeley National Laboratory, 2002.

[91] Barry Rountree et al. "Adagio: making DVS practical for complex HPC applications". In: *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*. ACM Press, 2009. DOI: 10.1145/1542275.1542340.

[92] Peter Sanders et al. *Sequential and ParallelData Structures and Algorithms The Basic Toolbox*. 2019, pp. 402–404.

[93] T. Santos and J. Barbosa. "Examining Checkpoint and Storage Schemes for Fault Tolerance in Computing Clusters". In: *Doctoral Symposium in Informatics Engineering* (2012), pp. 103–114.

[94] Constantin Scholl et al. "The divisible load balance problem with shared cost and its application to phylogenetic inference". In: (Jan. 2016). DOI: 10.1101/035840.

[95] Florian Schornbaum and Ulrich Rüde. "Extreme-Scale Block-Structured Adaptive Mesh Refinement". In: *SIAM Journal on Scientific Computing (SISC) 40-3 (2018), pp. C358-C387* (Apr. 22, 2017). DOI: 10.1137/17M1128411. arXiv: 1704.06829v3 [cs.DC].

[96] Russell Schwartz and Alejandro A. Schäffer. "The evolution of tumour phylogenetics: principles and practice". In: *Nature Reviews Genetics* 18.4 (Feb. 2017), pp. 213–229. DOI: 10.1038/nrg.2016.170.

[97] John Shalf, Sudip Dosanjh, and John Morrison. "Exascale Computing Technology Challenges". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 1–25. DOI: 10.1007/978-3-642-19328-6_1.

[98] Jeff J. Shi and Daniel L. Rabosky. "Speciation dynamics during the global radiation of extant bats". In: *Evolution* 69.6 (June 2015), pp. 1528–1545. DOI: 10.1111/evo.12681.

[99] Marc Snir et al. "Addressing failures in exascale computing". In: *The International Journal of High Performance Computing Applications* 28.2 (Mar. 2014), pp. 129–173. DOI: 10.1177/1094342014522573.

[100] S. Song et al. "Resolving conflict in eutherian mammal phylogeny using phylogenomics and the multispecies coalescent model". In: *Proceedings of the National Academy of Sciences* 109.37 (Aug. 2012), pp. 14942–14947. DOI: 10.1073/pnas.1211733109.

[101] Alexandros Stamatakis. "distributed and parallel algorithms and systems for inference of huge phylogenetic trees based on the maximum likelihood method". PhD thesis. Technische Universität München, June 2004.

[102] Alexandros Stamatakis. "RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies". In: *Bioinformatics* 30.9 (Jan. 2014), pp. 1312–1313. DOI: 10.1093/bioinformatics/btu033.

[103] Alexandros Stamatakis. "RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models". In: *Bioinformatics* 22.21 (Aug. 2006), pp. 2688–2690. DOI: 10.1093/bioinformatics/btl446.

[104] Alexandros Stamatakis, T. Ludwig, and H. Meier. “Computing Large Phylogenies with Statistical Methods: Problems & Solutions”. In: *Proceedings of 4th International Conference on Bioinformatics and Genome Regulation and Structure (BGRS2004)*. Novosibirsk, Russia, 2014.

[105] Alexandros Stamatakis, T. Ludwig, and H. Meier. “RAxML-III: a fast program for maximum likelihood-based inference of large phylogenetic trees”. In: *Bioinformatics* 21.4 (Dec. 2004), pp. 456–463. DOI: 10.1093/bioinformatics/bti191.

[106] Paola Stefanelli et al. “Whole genome and phylogenetic analysis of two SARS-CoV-2 strains isolated in Italy in January and February 2020: additional clues on multiple introductions and further circulation in Europe”. In: *Eurosurveillance* 25.13 (Apr. 2020). DOI: 10.2807/1560-7917.es.2020.25.13.2000305.

[107] Steinbruch Center for Computing (SCC). *ForHLR - Hardware and Architecture*. https://wiki.scc.kit.edu/hpc/index.php/ForHLR_-_Hardware_and_Architecture. 2020.

[108] Steinbruch Center for Computing (SCC). *Konfiguration des ForHLR II*. https://www.scc.kit.edu/dienste/forhlr2.php. 2020.

[109] James E. Tarver et al. “The Interrelationships of Placental Mammals and the Limits of Phylogenetic Inference”. In: *Genome Biology and Evolution* 8.2 (Jan. 2016), pp. 330–344. DOI: 10.1093/gbe/evv261.

[110] S. Tavare. “Some probabilistic and statistical problems in the analysis of DNA sequences”. In: *Lectures on Mathematics in the Life Sciences. Providence: Amer. Math. Soc* (1986). Ed. by R. M. Miura, pp. 57–58.

[111] Keita Teranishi and Michael A. Heroux. “Toward Local Failure Local Recovery Resilience Model using MPI-ULFM”. In: *Proceedings of the 21st European MPI Users’ Group Meeting on - EuroMPI/ASIA ’14*. ACM Press, 2014. DOI: 10.1145/2642769.2642774.

[112] The Open MPI Project. *MPI_Allreduce man page*. Mar. 20, 2020.

[113] The OpenMPI Project. *OpenMPI FAQ*. https://www.open-mpi.org/faq/?category=perftools. Accessed 11th May 2020. May 2019.

[114] Thomas N. Theis and H.-S. Philip Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science & Engineering* 19.2 (Mar. 2017), pp. 41–50. DOI: 10.1109/mcse.2017.29.

[115] M. Vijay and R. Mittal. “Algorithm-based fault tolerance: a review”. In: *Microprocessors and Microsystems* 21.3 (Dec. 1997), pp. 151–161. DOI: 10.1016/s0141-9331(97)00029-x.

[116] Tiffany Williams and Bernard Moret. “An Investigation of Phylogenetic Likelihood Methods”. In: *Proceedings of 3rd IEEE Symposium on Bioinformatics and Bioengineering (BIBE’03)*. 2003, pp. 79–86.

[117] C. R. Woese, O. Kandler, and M. L. Wheelis. "Towards a natural system of organisms: proposal for the domains Archaea, Bacteria, and Eucarya." In: *Proceedings of the National Academy of Sciences* 87.12 (June 1990), pp. 4576–4579. DOI: 10.1073/pnas.87.12.4576.

[118] G. A. Wray, J. S. Levinton, and L. H. Shapiro. "Molecular Evidence for Deep Precambrian Divergences Among Metazoan Phyla". In: *Science* 274.5287 (Oct. 1996), pp. 568–573. DOI: 10.1126/science.274.5287.568.

[119] Zhenxiang Xi et al. "Coalescent versus Concatenation Methods and the Placement of Amborella as Sister to Water Lilies". In: *Systematic Biology* 63.6 (July 2014), pp. 919–932. DOI: 10.1093/sysbio/syu055.

[120] Ya Yang et al. "Dissecting Molecular Evolution in the Highly Diverse Plant Clade Caryophyllales Using Transcriptome Sequencing". In: *Molecular Biology and Evolution* 32.8 (Apr. 2015), pp. 2001–2014. DOI: 10.1093/molbev/msv081.

[121] Ziheng Yang. "Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods". In: *Journal of Molecular Evolution* 39.3 (Sept. 1994), pp. 306–314. DOI: 10.1007/bf00160154.

[122] Xiaofan Zhou et al. "Evaluating Fast Maximum Likelihood-Based Phylogenetic Programs Using Empirical Phylogenomic Data Sets". In: *Molecular Biology and Evolution* 35.2 (Nov. 2017), pp. 486–503. DOI: 10.1093/molbev/msx302.

[123] Ilya Zhukov et al. "Scalasca v2: Back to the Future". In: *Tools for High Performance Computing 2014*. Springer International Publishing, 2015, pp. 1–24. DOI: 10.1007/978-3-319-16012-2_1.