On-Line Supplement to: Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG

Lukas Hübner, Alexey M. Kozlov, Demian Hespe, Peter Sanders, Alexandros Stamatakis
May 10, 2021

This is the on-line supplement to “Exploring Parallel MPI Fault Tolerance Mechanisms for Phylogenetic Inference with RAxML-NG”. In Section 1 we describe the characteristics of the empirical datasets we use in our experiments. In Section 2 we describe the hardware, software, and tree search settings we use. In Section 3 we describe how we simulate rank failures when executing FT-RAxML-NG under ULFM and OpenMPI. In Section 4 we discuss the impact of mini-checkpointing (without failures) as well as the combination of mini-checkpointing + recovery + additional work in case of (simulated) failures on the overall runtime. In Section 8 we describe our modifications to, and implementation of, the tree-based phylogeny-aware Multiple Sequence Alignment (MSA) compression scheme described by Ané and Sanderson [1].

1 Datasets

Table 1 lists the characteristics of the empirical datasets used in our experiments.

Table 1: Characteristics of the datasets used for evaluating RAxML-ng and FT-RAxML-NG

| Designator | Data type | # taxa | # alignm. sites | # unique patterns | # partitions | file size [MiB] | Reference |
|------------|-----------|--------|-----------------|------------------|--------------|----------------|-----------|
| SongD1     | DNA       | 37     | 1,338,678       | 746,408          | 1            | 48             | Song et al. [19] |
| MisoD2a    | DNA       | 144    | 1,240,377       | 1,142,662        | 100          | 171            | Misof et al. [12] |
| MisoD2b    | DNA       | 144    | 413,459         | 371,434          | 50           | 57             | Misof et al. [12] |
| WickD3a    | DNA       | 103    | 436,077         | 422,676          | 14           | 43             | Wicket et al. [25] |
| WickD3b    | DNA       | 103    | 290,718         | 277,375          | 8            | 29             | Wicket et al. [25] |
| XiD4       | DNA       | 46     | 239,763         | 165,781          | 1            | 11             | Xi et al. [26]   |
| PrumD6     | DNA       | 200    | 394,684         | 236,674          | 75           | 76             | Prum et al. [15] |
| TarvD7     | DNA       | 36     | 21,410,970      | 8,320,738        | 1            | 736            | Tarver et al. [23] |
| PeteD8     | DNA       | 174    | 3,011,099       | 2,248,590        | 4,116        | 500            | Peters et al. [14] |
| ShiD9      | DNA       | 815    | 20,364          | 13,311           | 29           | 16             | Shi and Rabosky [17] |
| StamD10    | DNA       | 436    | 1,371           | 1,011            | 1            | 0.6            | Stamatakis et al. [20] |
| NagyA1     | AA        | 60     | 172,073         | 156,312          | 594          | 10             | Nagy et al. [13] |
| ChenA4     | AA        | 58     | 1,806,035       | 1,547,914        | 1            | 100            | Chen et al. [2] |
| YangA8     | AA        | 95     | 504,850         | 476,259          | 1,122        | 46             | Yang et al. [27] |
2 Experimental Setup

2.1 Hardware and Software

2.1.1 ForHLR II

The first system we use for our experiments is the ForHLR II supercomputer located at the Steinbruch Center for Computing (SCC) in Karlsruhe. It comprises a total of 1,178 worker nodes. Each node is equipped with two sockets of Intel Xeon E5-2660 v3 (Haswell) Deca-Core CPUs with a clock rate of 2.1 GHz (max. 3.3 GHz) which results in a theoretical maximum peak performance 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 via an InfiniBand 4X EDR interconnection [22]. Figure 1 shows the architecture of ForHLR II.

![ForHLR II architecture](https://www.scc.kit.edu/dienste/forhlr2.php)

**Figure 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; simplified to only show the part of the infrastructure we used.

ForHLR II has 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 maximum I/O performance of 2 GiB s\(^{-1}\) on a single node and 10 GiB s\(^{-1}\) across all nodes. Two file server nodes provide file accesses [21]. Each of them has identical hardware as the compute nodes. The above is the default configuration on the ForHLR II. It is however 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 intend to measure the performance for a typical use case, as typical RAxML-ng users will not manually tune their file-system configuration.

All nodes are running Red Hat Enterprise Linux (RHEL) 7.x and Slurm 20.02.3.

2.1.2 SuperMUC-NG

The second system used in our benchmarks is the SuperMUC-NG supercomputer at the Leibniz-Rechenzentrum (LRZ) in Munich. It consists of 6,336 thin nodes (plus additional nodes that are not used in our experiments). Each node is equipped with 96 GiB RAM and two Intel Xeon Platinum 8174 (Skylake) CPUs with 33 MiB cache and 24 physical cores plus hyperthreading each. The cores run at a clock rate of 3.1 GHz (max. 3.9 GHz) resulting in a total peak performance of 26.3 PFLOPS. All nodes within an island (a collection of 792 nodes) are connected via a fast OmniPath network with 100 Gbit s\(^{-1}\) using a fat tree topology (all our experiments run within one island). We also disable the Energy Aware Runtime which dynamically

\(^1\)https://www.scc.kit.edu/dienste/forhlr2.php

\(^2\)https://doku.lrz.de/display/PUBLIC/SuperMUC-NG
decreases the CPU frequency. The file system is a IBM Spectrum Scale (GPFS) with a theoretical maximum I/O performance of 6 to 25 GiB/s for the area we use. All nodes are running SuSE Linux Enterprise (HPC Module) Version 12 and Slurm 18.08.4–2.

2.1.3 Build Details

We use OpenMPI 4.0.2 and ULFM v4.0.2u1 and GCC 8.2.0 for our experiments where not mentioned otherwise. FT-RAxML-NG is based upon e0e4b45 on branch coarse released on March 5th 2020. As FT-RAxML-NG is compiled with NDEBUG defined, we do the same for Reference-RAxML-NG.

2.2 Tree Search Settings

We conduct the profiling experiments with the following RAxML-ng options when not stated otherwise: To use partial input file loading, we have to use random starting trees for initiating the tree search. The respective random seeds used are listed in Table 2. The following performance-relevant and model-specific RAxML-ng settings are used in each run: Tip-inner is disabled, pattern compression is enabled, per-rate scalers are disabled, site-repeats are enabled, the fast Subtree Pruning and Regrafting (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 Message Passing Interface (MPI) rank is one (see RAxML-ng\(^3\) for details). The number of partitions (models) we use for each dataset is specified in each experiment.

| dataset    | random seed ForHLR II | random seed SuperMUC |
|------------|------------------------|----------------------|
| NagyA1     | 1574547114             | 0                    |
| ChenA4     | 1574484011             | 0                    |
| YangA8     | 1574484011             | 0                    |
| PeteD8     | 1574549152             | 0                    |
| SongD1     | 1574549152             | 0                    |
| MisoD2a    | 1574443931             | 0                    |
| XiD4       | 1574528895             | 0                    |
| ShiD9      | 1574549152             | 0                    |

Table 2: Random Seeds used in the profiling experiments.

3 Failure Simulation

We can simulate core failures in numerous ways without root access to the High Performance Computing (HPC) machines. When using User Level Failure Mitigation (ULFM) with no heartbeat thread, it suffices to put the application program into a long sleep to simulate a failure. When using a heartbeat thread, sending the SIGKILL signal to the process ID of a rank will simulate a failure. The program cannot catch, block or ignore SIGKILL. It can therefore not perform any cleanup operation [10]. Other possible signals we can send include SIGSEGV, SIGILL, SIGFPE, SIGBUS, SIGXFSZ, SIGPWR, and SIGXCPU. None of these signals allow the receiving process to perform a cleanup operation. Our tests show that ULFM detects all of these simulated failures as rank failures with no noticeable difference, that is, the next MPI operation will fail with MPI_ERROR_PROC_FAILED. We are able to revoke the communicator using MPI_Comm_revoke() and the new communicator we subsequently build using ULFM’s MPI_Comm_shrink() does not contain this “failed” node. We choose to simulate failures in experiments with ULFM by killing a process via signalling SIGKILL either by invoking kill -SIGKILL [8] or via self-signalling using raise(SIGKILL) [9].

\(^3\)https://github.com/amkozlov/raxml-ng/wiki
When using OpenMPI we simulate failures either by calling `MPI_Comm_split()` such, that all ranks remain in the same communicator but the rank ids are shifted by one. This causes every rank to obtain a new site (data) assignment. We then proceed by executing a full recovery procedure. This allows us, for example, to measure the overhead induced by recovery but without having to account for the overhead caused by continuing with fewer ranks, after a rank failure.

4 Runtime overhead with and without failures

We measure the runtime overhead caused by mini-checkpointing without failures occurring as well as the runtime overhead by-checkpointing, recovery, and recomputing the lost work in case of failures (see Figure 2). We perform these experiments on the SuperMUC-NG supercomputer.

![Figure 2: Runtime of reference vs fault-tolerant RAxML with different MPI implementations.](image)

Each symbol represents exactly one measurement (1–10 repeats, see Section 4.1). All measurements on one dataset using OpenMPI v4.0.2 are run on the same set of nodes. The runtimes are normalized to the mean reference runtimes for the respective dataset. The text below the measurements displays the respective mean reference runtime. The measurements on PeteD8 with (simulated) failures converged to a different tree and should therefore be interpreted with extreme caution (see Section 4.3).
4.1 Runtime overhead without failures

Even with no failures, we expect FT-RAxML-NG (Fault-Tolerant RAxML-ng) to be slower than Reference-RAxML-NG. This is because FT-RAxML-NG 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 determine the runtime overhead caused by ULFM separately from the runtime overhead induced by our modifications to RAxML-ng. We therefore measure the runtime of FT-RAxML-NG with OpenMPI v4.0.2 and ULFM v4.0.2u1 as MPI implementations (see Figure 2). ULFM has been merged into the OpenMPI code base, but is not included in any release yet. As this represents the bleeding-edge in fault-tolerant MPI development, we also perform measurements using the OpenMPI version (git commit 6d58d843b6).

Using OpenMPI v4.0.2, and ULFM v4.0.2u1 we perform 10 repeats on the datasets NagyA1, YangA8, SongD1, XiD4, and ShiD9. We perform only 3 repeats for the datasets ChenA4, PeteD8, and TarvD7. We calculate, that conducting only 3 repeats conserves approximately $\frac{2}{3}$ of the otherwise needed energy-, CO$_2$-, and computing-resources. For the same reason, we perform fewer repeats using the fault-tolerant OpenMPI (FT-OpenMPI) version on these datasets. In our measurements, the slowdown of FT-RAxML-NG, without failures, running with OpenMPI v4.0.2 compared to the unmodified RAxML-ng running under OpenMPI v4.0.2 is 1.00 ± 0.04 (Figure 2).

ULFM v4.0.2u1 did not run correctly on the SuperMUC-NG when using more than 13 nodes. We therefore could not perform any measurements using ULFM v4.0.2u1 for the TarvD7 and ChenA4 datasets. The slowdown of FT-RAxML-NG running under ULFM v4.0.2u1 compared to the unmodified RAxML-ng running under OpenMPI v4.0.2 is 1.7 ± 0.6 (Figure 2). The slowdown of FT-RAxML-NG running under FT-OpenMPI compared to Reference-RAxML-NG running under OpenMPI v4.0.2 is 6 ± 5.

Given a sufficiently high frequency of job failures, we believe that a slowdown of 1.00 ± 0.04 is alleviated by the corresponding savings in man hours and a shorter walltime-to-completion as manual job re-submissions are not required any more. Nonetheless, the ULFM overhead of 70 ± 60% and the FT-OpenMPI overhead of 500 ± 500% are unacceptably high. As ULFM and the OpenMPI version we used are currently still in their beta version, we are confident that this overhead will be reduced in future versions as it is critical for designing fault-tolerant parallel production-level applications.

4.2 Runtime overhead with failures

In case of (simulated) failures, we expect FT-RAxML-NG to be slower than FT-RAxML-NG and Reference-RAxML-NG without failures, as we have to recover from, and recompute the work lost during these failures. For the measurements with simulated failures we use OpenMPI v4.0.2 as the MPI implementation. We simulate ten failures as described in Section 3. The time points at which we simulate the failures are evenly distributed over the overall runtime of the runs. In our measurements, the slowdown of FT-RAxML-NG with 10 failures compared to Reference-RAxML-NG is 1.4 ± 0.3; that is approximately 4% additional runtime per failure.

4.3 Reproducible Results and Numerically Unstable AllReduce Operations

We need to distinguish between two types of “differences in the result” in scientific software: bitwise reproducibility and accuracy. The observed deviation in the final likelihood-scores stems solely from the numerical instability of the MPI AllReduce algorithms used. Take, for example, the number of ranks that are used to execute RAxML-ng. Depending on how many cores participate at the parallel reduction operation that we use to compute the overall sum over the per-site log-likelihood scores, the intermediate results of this reduction operation per core will be rounded differently. This means that the results are not bitwise identical, but still accurate. This is a known problem not only in (FT/standard-)RAxML-NG but of parallel scientific computing in general. In addition to this, different compiler versions and flags as well as different hardware instruction sets will also hinder bitwise reproducibility[16]. Because different core counts might yield different likelihoods for intermediate results, the hill-climbing heuristic during topology optimization may take a different path and therefore arrive at a different result.
For the data shown in Figure 2, the Log-Likelihood (LLH) is exactly the same across all repeats using the same settings and MPI runtime. Also, as long as there are no failures involved, the LLH is exactly the same across all MPI runtimes except FT-OpenMPI. The maximum relative difference without failures to the reference is $4.9 \times 10^{-6}$. The maximum relative difference between the reference and a run with 10 simulated failures is $2.05 \times 10^{-6}$.

There are minor differences in the LLH of the runs on ChenA4 (0.000018 LLH-units), PeteD8 (360), and TarvD7 (0.00084) when comparing FT-RAxML-NG without failures run under OpenMPI v4.0.2 and FT-OpenMPI. In all three cases, the resulting final tree topologies are structurally exactly identical though. When comparing FT-RAxML-NG with 10 failures to Reference-RAxML-NG, there are also some minor differences in the LLH ($< 0.01$ LLH-units), but the tree inferences arrive at exactly the same tree topologies.

The exception is PeteD8: For this dataset, the final LLH differs by 150 likelihood units and the resulting tree topologies are structurally different. The absolute and relative Robinson-Foulds distances\[4\] between these two topologies are $RF_{\text{abs}}(A, B) = 4$ and $RF_{\text{rel}}(A, B) = 0.011696\%$. We perform all statistical tests provided by IQtree [11] to check whether these two trees differ significantly. The fact that the tree searches arrive at different trees also yields the respective runtime measurements incomparable. IQtree issues multiple errors hinting to a rough LLH-surface (\texttt{ABNORMAL\_TERMINATION\_IN\_LNSRCH}) and warns that the estimated model parameters are at a boundary that can cause numerical instability, but in the end deems the two trees significantly different ($p < 0.05$ for all tests). This divergence of tree searches is caused by numerically unstable MPI\_AllReduce operations as described above and constitutes a well-known phenomenon [16, 3]. Keeping results bitwise identical and reproducible thus represents an important future challenge of fault-tolerant codes and supercomputing in general.

### 4.4 Faster than the Reference Allows

We observe, that for some datasets (e.g. TarvD7 and ChenA4), the fault-tolerant code actually runs faster than Reference-RAxML-NG. The inferred tree topologies as well as the LLHs are identical. In order to analyse if this is caused by compiler and linker optimizations, we inferred trees on the TarvD7 and ChenA4 datasets with Reference-RAxML-NG and FT-RAxML-NG without failures using three different compilers (see Table 3). All tree inferences on a single dataset arrive at the same tree. The result’s LLHs of the inference on the ChenA4 datasets differ by 0.000018 between the Intel compiler and the two GCC compilers. This is another artefact of non-reproducible collective operations (see Section 4.3). To be able to compile RAxML-ng with the Intel compiler we had to increase the C\++ version from 11 to 14 and disable the Terraphast module.

| dataset | #ranks | compiler | reference | with checkpoints |
|---------|--------|----------|-----------|-----------------|
| A4      | 4000   | gcc 8.2.0| 655.6 to 6,619 s | 623.4 to 649.1 s |
| A4      | 4000   | gcc 9.2.0| 568.6 s | 535.3 s |
| A4      | 4000   | icc 19.1 | 610.5 s | 635.2 s |
| D7      | 4000   | gcc 8.2.0| 373.4 to 378.7 s | 346.5 to 348.3 s |
| D7      | 4000   | gcc 9.2.0| 338.3 s | 298.6 s |
| D7      | 4000   | icc 19.1 | 329.1 s | 317.8 s |

**Table 3:** Comparison of runtimes of reference-RAxML-ng and FT-RAxML-NG using different compilers. Always using OpenMPI v4.0.2.

The inference on ChenA4 using icc 19.1 is faster for Reference-RAxML-NG than for FT-RAxML-NG. On all other measurements, FT-RAxML-NG runs faster than Reference-RAxML-NG. We therefore suspect, that a compiler or linker optimization causes this effect but do not further investigate this as it is beyond the scope of the current paper.
5 Influence of Checkpointing-Frequency on Runtime

We measure the influence of the checkpointing frequency on the runtime of FT-RAxML-NG. We perform these experiments on the SuperMUC-NG supercomputer. As can be seen from Figure 2, the overhead of mini-checkpointing alone is $1.00 \pm 0.04$. We do therefore not expect any substantial improvements to the runtime when omitting checkpoints.

![Figure 3: Influence of checkpointing-frequency on the runtime of FT-RAxML. We show the mean runtime of FT-RAxML-NG with various checkpointing-frequencies relative to the mean runtime without checkpoints. The error bars indicate the standard deviation (3 repeats).](image)

FT-RAxML-NG has three types of checkpoints: Saving the model parameters, saving the tree topology, and writing checkpoints to disk. This poses the following challenges: Skipping for example a model parameter checkpoint and taking the next tree topology checkpoint would create an inconsistent state when restoring. This is, because this combination of tree topology and model parameters never existed during the tree search. The same observation holds for skipping a tree topology checkpoint but taking the following model parameter checkpoint. Also, we do not know in advance how many checkpoints of the other type will follow and therefore if we have to take the current checkpoint to arrive at a consistent state. We therefore decided to conceptually treat each mini-checkpoint as if it would checkpoint both, the tree topology, and the model parameters. This reduces the problem to one type of meta-checkpoint. The calls to the checkpointing routines set a flag, indicating that the corresponding data needs to be checkpointed regardless of whether we skip this particular checkpoint or not. Additionally, if this checkpoint is taken (i.e. not skipped), a checkpoint is created. For this, the checkpointing routine checks the flags for the tree topology and model parameter checkpoints and creates the appropriate copies. We provide an example in Listing 1.

When saving a checkpoint to disk, and the previous mini-checkpoints were skipped, we will write an outdated state to disk. We want to avoid creating a mini-checkpoint in the checkpoint-to-disk routine, as the latter is currently local and adding a synchronization point (possibility of detecting a rank failure) here would increase the complexity of the failure handling. In this experiment we are not recovering from a checkpoint which was written to disk. We therefore ignore that these may be inconsistent.
We measure the impact that the checkpointing frequency has on the runtime for dataset NagyA1 using 96 ranks and for dataset ChenA4 using 80 ranks as well as 4000 ranks (see Figure 3). The standard deviation of the runtimes shown in Figure 2 for FT-RAxML-NG without failures using OpenMPI v4.0.2 is 3.71 s and 13.1 s for NagyA1 using 96 ranks and ChenA4 using 4000 ranks, respectively. The standard deviations of all measured checkpointing frequencies together are 2.31 s (NagyA1 using 96 ranks), 4.18 s (ChenA4 using 800 ranks), and 2.03 s (ChenA4 using 4000 ranks). We therefore conclude, that the effect of checkpointing frequency on runtime is of the same order of magnitude as measurement noise. We hence decide against introducing this feature into FT-RAxML-NG, as it would increase the amount of lost work in case of a failure.

6 Memory Consumption of Mini-Checkpoints

The amount of additional memory used by the mini-checkpoints is highly dependent on the dataset and search settings. More precisely, this depends on the number of partitions, the chosen substitution model, the rate heterogeneity model, and the branch length scaling mode. There are three branch length scaling modes in RAxML-NG: linked, unlinked, and scaled. In the linked mode, all partitions share a common set of branch lengths. In the unlinked mode each partition has its own set of branch lengths. In the scaled mode a global set of branch lengths is estimated as for the linked mode. However, the per-partition branch lengths are then obtained by multiplying these global branch lengths with a per-partition scaler. Therefore, depending on the branch length scaling mode, the model has a different number of parameters that need to be saved during a mini-checkpoint. At present, FT-RAxML only supports the linked and scaled modes.

We measure the highwater memory consumption of FT-RAxML-NG and reference-RAxML-ng. We perform these experiments on the SuperMUC-NG supercomputer. The highwater memory consumption is the largest amount of memory a program allocated at any point in time during its execution. We use 

```
/usr/bin/time -v
```

to measure the average highwater memory consumption per rank (see Figure 4). This kind of measurement captures the amount of memory allocated and freed, but not the amount of memory used by the data structures.

The numbers printed below the measurements indicate the absolute memory overhead compared to the reference. That is, for example for YangA8 with 594 partitions, checkpointing needs approximately 17 MB. As expected, the memory consumption increases with the number of partitions. The additional memory consumption of FT-RAxML ranges from 0.085 to 68 MB.

---

4 not the bash built-in
Figure 4: Average highwater memory consumption per rank relative to reference RAxML-ng as measured by /usr/bin/time -v. The numbers below the measurements indicate the absolute difference in memory consumption between FT-RAxML-NG and Reference-RAxML-NG. All measurements use proportional branch length scalers (+B). GTR: General Time-Reversible Model [24]. WAG: Whelan and Goldman model [18]. FC: Empirical base frequencies. G4m: Γ site-heterogeneity model [28] with 4 discrete rate categories; maximum-likelihood estimation of α.

7 Pseudocode for the Checkpointing Procedures

Mini-Checkpointing: Redistribution of Model Parameters

Each MSA partition has a set of model parameters associated with it. 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 (see Figure 1 in main text), the program redistributes them to all ranks. 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 1, synchronizing at each broadcast.

Upon broadcasting, the ranks gather the received models in a temporary copy. If no rank failure occurs during the broadcast, 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.

Broadcasting the model parameters is therefore a global operation and coordinated using the described two-phase commit protocol.

Mini-Checkpointing: Saving the currently best tree topology

All changes applied to the tree topology and branch lengths occur at all ranks simultaneously. Hence, we do not need to broadcast them. To create a mini-checkpoint of the tree topology and branch lengths, we copy the respective data structure in a local operation (see Algorithm 2). Saving the tree topology and branch
Algorithm 1 Broadcast of model parameters

```plaintext
procedure BroadcastModelParameters
    for each rank do
        Broadcast(all models this rank is responsible for) ▷ temporary copy
    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
```

Lengths is therefore uncoordinated in the sense that there is no two-phase commit protocol and no additional synchronization point in contrast to broadcasting the model parameters. Coordinating these checkpoints is thus not achieved by a conventional two-phase commit protocol but rather by the synchronisation steps between likelihood calculations which ensure that the tree search state is synchronized across all ranks at the time the checkpoint is created.

Algorithm 2 Create a tree topology checkpoint

```plaintext
procedure CheckpointTreeTopology
    Copy the tree topology and branch lengths using pll_utree_clone ▷ local operation
end procedure
```

Checkpointing to disk

Writing the checkpoints to disk is also a local operation as only a single rank is accessing the disk. Additionally all data it needs – the model parameters, the tree topology, and the search state – is up-to-date and consistent. We describe the algorithm in pseudo-code in Algorithm 3 and argue that it should be considered as being synchronized even without additional synchronisation calls analogous to the checkpointing of the tree topology (see above).

Algorithm 3 Write a checkpoint to disk

```plaintext
procedure CheckpointToDisk
    if I'm the master rank then
        Write tree, models and search state to disk ▷ local operation, everything up-to-date.
    end if
end procedure
```

8 Tree-based Compression of Multiple Sequence Alignments

In this section, we describe our modifications and implementation of the tree-based phylogenetic MSA compression method described by Ané and Sanderson [1]. A more detailed description can be found in the following Master’s thesis [6]. An Open Source implementation of the algorithm is available at https://github.com/lukashuebner/ft-raxml-ng on the “tree-based-msa-compression” branch.
An MSA consists of a set of aligned sequences. We will limit ourselves to Deoxyribonucleic Acid (DNA) sequences for now. This implies that every sequence has the same length, possibly including gaps. Sequences can be, for example DNA or Amino Acid (AA). 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. This is generally known as ambiguity encoding. 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 codes and a gap [7] code. For example the character K represents either a G or a T at a position in the sequence. We expand the previously described [1] compression scheme to include this ambiguity encoding as it is required for analyzing empirical datasets and describe an algorithm to encode and decode the given MSA below.

The sequences of an MSA are located at the tips of a corresponding phylogenetic tree. The main idea of the tree-based compression scheme is to fully store only one sequence. We store all other sequences as a set of changes along the edges/branches of the tree (see Figure 5). We annotate these changes at the edges, for example 5 → 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. For this compression approach, we evidently also need to store the corresponding tree topology for the specific compression as well. Using the most parsimonious tree (i.e., the tree that can explain the data by the least amount of mutations) guarantees the shortest encoding (best compression) [1].

For details on how to encode the tree in a binary Newick format and more details on how to encode changes along the branches of the tree, see Ané’s and Sanderson’s publication [1].

To encode MSA’s sequences, we arbitrarily but deterministically choose one node as the root node. We thereby also defined define the order in which all nodes are visited during tree-traversal. We can thus encode the sequences in the MSA as a sequence at the root and the set of changes along the tree edges which result in the MSA’s sequences at the tips of the tree. We then store the ancestral states of the sequence at the chosen 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, thus distinguishing it from the state “any nucleotide” (N), which may also appear in the sequence. 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 for decoding the nucleotide states of one site are stored contiguously in memory, thereby enabling a cache-efficient decoding. Additionally, we do not need to store the index of the site this change modifies, which would need an additional log(sequenceLength) bits per change. We also

Figure 5: Encoding of the MSA. 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 XORing the nucleotide states before and after the change. The index data structure points to the beginning of each site’s encoding.
store an index data structure $I$ mapping the site identifier to the start of the encoding for the specific site. We implement this via an array. We thereby implement random read access to the sites of the 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 XORing 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 $mask = C \oplus T = 0100 \oplus 0010 = 0110$. This allows us to handle ambiguities.

We also identify the edges by a unique identifier. We obtain this identifier by enumerating the nodes in pre-order (parent, left child, right child) and assigning each edge the same id as the node it leads to. We chose pre-order instead of post-order as used by Ané and Sanderson [1] because this way we store the changes “root to tip”. This yields the decoding more straightforward as we can implement it using a linear sweep without look-back through the encoding. We store a dummy entry in the index data structure pointing just past the last valid change to mark the end of the encoding.

### 8.1 Description of the Algorithm

In this Section, we provide a description of the algorithm for compression and decompression of the MSA data. Compression consists of finding the ancestral states of the parsimony tree as well as encoding the changes along the tree. We refrain from encoding and decoding the tree itself and keep it in memory in an uncompressed format.

#### Computing the Ancestral States of the Parsimony Tree

We use Hartigan’s [5] algorithm to calculate an 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.

The algorithm takes a phylogenetic tree with fixed topology and fixed sequences 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.

**Algorithm 4** Hartigan’s [5] algorithm: Overview

| Given: A Tree $T$ with $i$ tips and the corresponding MSA $S$ with $i$ sequences $S_0 \ldots S_i$ |
|---|
| 1: for each $s_i \in S$ do |
| 2: \hspace{1em} PHASE1($s_i$, $T$) |
| 3: \hspace{1em} PHASE2($s_i$, $T$) |
| 4: end for |

To compute the possible ancestral states, we visit at each inner node in post-order (left child, right child, parent). This means that once we arrive at a node, we have already processed both children. For each tip, the set of possible states consists of the single fixed state in the input data. For all inner nodes, we check if the current node’s children have possible common states. If they do, the current node’s possible states are given by 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 6.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 visit 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 given, and we do not alter them. If the tip’s state differ from its parent’s state, a mutation occurred (see Figure 6.b and Algorithm 6).

In a bifurcating tree with vertex set $V$ and edge set $E$, $|E| = |V| - 1$ 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
Figure 6: Reconstruction of the ancestral states as described by Hartigan [5]. 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 randomly chose a state 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 set it as the node’s ancestral state. If it is not, we set a random state from the child’s possible ancestral states. In this case, a mutation occurred (+1).

Algorithm 5 Hartigan’s [5] 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:       $V(N) \leftarrow \{\text{nucleotide}(N)\}$  
5:     else  
6:       let $A$ and $B$ be the children of $N$.  
7:       if $V(A) \cap V(B) \neq \emptyset$ then  
8:         $V(N) \leftarrow V(A) \cap V(B)$  
9:       else  
10:         $V(N) \leftarrow V(A) \cup V(B)$  
11:     end if  
12:   end if  
13: end traversal  
14: end function

Algorithm 6 Hartigan’s [5] 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  
4:   let the current node be $A$ and its parent be $P$.  
5:   if $V(P) \subseteq V(A)$ then  
6:     $V(A) \leftarrow V(P)$  
7:   else  
8:     $V(A) \leftarrow \{\text{RANDOM Choice}(V(A))\}$  
9: end if  
10: end traversal  
11: end function
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 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 operations 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 of the Sequences

Given a 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 5 and Algorithm 7). We will denote the $s$-th site of the $j$-th sequence as $S^j_s$.

To facilitate read access to random sites, we store the start of the encoding of each site in an index data structure $I$.

For each site $i$, we store the nucleotide state $s^i_{\text{root}}$ 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.

Algorithm 7 MSA compression

```plaintext
1: function Encode(Tree T, Sequences S*)
2: let I be a vector mapping each site to its start location in the encoding
3: EncodeTree(T)  
4: Skip space for $|S^\text{root}| + 1$ pointers in the output stream to later store I in
5: for each $s^i_{\text{root}} \in S^\text{root}$ do
6:     I.pushBack(< i, current position >)
7:     Write $s_i$ to output stream
8:     traverse T in pre-order  \( \triangleright \text{ Skipping the root} \)
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 $s^A_j \neq s^j_{\text{parent}}$ then
12:         Write<$s^A_j$, $j$>
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 5 and Algorithm 8) we read the index data structure, mapping the site identifiers to the start of their encoding in the bitstream. For each site $s$ we intend to decode, we move to the specified location and start reading. The first four bits we read are the site’s nucleotide state at the root sequence $s^\text{root}$. 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, in pre-order. Thus, we will never have to move backwards during the tree traversal to apply a change.

We encode sites independently of each other. We can therefore compute and write the encoding for each site sequentially on a single Processing Element (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.
Algorithm 8 MSA decompression

1: \textbf{function} \textsc{Decode}(File $F$, Range of Sites $R \subseteq [1, |S^1|])$ \Comment{Runtime: $|R| \cdot T_{dfs} \in \mathcal{O}(m \cdot n)$}
2: \hspace{1em} $T \leftarrow \textsc{DecodeTree}(F)$
3: \hspace{1em} $I \leftarrow \textsc{ReadI}(F)$
4: \hspace{1em} \textbf{for each} $s \in R$ \textbf{do}
5: \hspace{2em} Go to start of the site’s encoding in the file \Comment{As indicated by $I$}
6: \hspace{2em} Read $S^s_{\text{root}}$ from the input stream \Comment{One hot-encoded}
7: \hspace{2em} Read $<\text{substitutionMask}, \text{edgeID}>$ from input stream
8: \hspace{2em} \textbf{traverse} $T$ in pre-order
9: \hspace{3em} Set the node’s nucleotide state to its parent nucleotide state
10: \hspace{3em} \textbf{if} next change is on the edge leading to the current node \textbf{then}
11: \hspace{4em} Apply change-mask to the current node’s state
12: \hspace{4em} Read $<\text{substitutionMask}, \text{edgeID}>$ from input stream
13: \hspace{3em} \textbf{end if}
14: \hspace{2em} \textbf{end traversal}
15: \hspace{1em} \textbf{end for}
16: \textbf{end function}

8.2 Experimental Results

We use the algorithm described in section 8.1 to compress eleven different DNA datasets with sequences with a pairwise-identity-score ranging of 0.38 to 0.86 (see Figure 7). The compression ratio we observe ranges from 1.9 to 21.7 with a median of 3.4 and a standard deviation of 6.5 (not normally distributed). We conclude, that the compression ratio is not sufficient to store the full MSA in memory on each node for all datasets.

\begin{center}
\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{compression_ratio.png}
\caption{Compression ratio of different datasets using the tree-based MSA-compression. The compression ratio is defined as the size of the compressed sequences divided by the size if the uncompressed sequence. The average pairwise-sequence-identity score (avg PIS) quantifies how similar two sequences are. It is defined as the average over the number of sites each pair of sequences has in common.}
\end{figure}
\end{center}

References

[1] Cécile Ané and Michael J. Sanderson. Missing the forest for the trees: Phylogenetic compression and its implications for inferring complex evolutionary histories. \textit{Systematic Biology}, 54(1):146–157, February 2005.
[2] 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. *Systematic Biology*, 64(6):1104–1120, August 2015.

[3] Diego Darriba, Tomáš Flouri, and Alexandros Stamatakis. The State of Software for Evolutionary Biology. *Molecular Biology and Evolution*, 35(5):1037–1046, 01 2018.

[4] William H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2:7–28, December 1985.

[5] J. A. Hartigan. Minimum mutation fits to a given tree. *Biometrics*, 29(1):53, March 1973.

[6] Lukas Hübner. Load-balance and fault-tolerance for massively parallel phylogenetic inference. Master’s thesis, Karlsruhe Institute of Technology, 2020.

[7] IUPAC. Nucleotide codes.

[8] Michael Kerrisk. Manual page of linux’s `kill`. [http://man7.org/linux/man-pages/man1/kill.1.html](http://man7.org/linux/man-pages/man1/kill.1.html).

[9] Michael Kerrisk. Manual page of linux’s `raise`. [http://man7.org/linux/man-pages/man3/raise.3.html](http://man7.org/linux/man-pages/man3/raise.3.html).

[10] Michael Kerrisk. Manual pages of Linux’s `signals`. [http://man7.org/linux/man-pages/man7/signal.7.html](http://man7.org/linux/man-pages/man7/signal.7.html).

[11] Bui Quang Minh, Heiko A Schmidt, Olga Chernomor, Dominik Schrempf, Michael D Woodhams, Arndt von Haeseler, and Robert Lanfear. IQ-TREE 2: New Models and Efficient Methods for Phylogenetic Inference in the Genomic Era. *Molecular Biology and Evolution*, 37(5):1530–1534, 02 2020.

[12] B. Misof, S. Liu, K. Meusemann, R. S. Peters, A. Donath, C. Mayer, P. B. Frandsen, J. Ware, T. Flouri, R. G. Beutel, O. Niehuis, M. Petersen, F. Izquierdo-Carrasco, T. Wappler, J. Rust, A. J. Aberer, U. Aspock, H. Aspock, D. Bartel, A. Blanke, S. Berger, A. Bohm, T. R. Buckley, B. Calcott, J. Chen, F. Friedrich, M. Fukui, M. Fujita, C. Greve, P. Grobe, S. Gu, Y. Huang, L. S. Jermiin, A. Y. Kawahara, L. Krogmann, M. Kubiak, R. Lanfear, H. Letsch, Y. Li, Z. Li, J. Li, H. Lu, R. Machida, Y. Misehiro, P. Kapli, D. D. McKenna, G. Meng, Y. Nakagaki, J. L. Navarrete-Heredia, M. Ott, Y. Ou, G. Pass, L. Podsiaiowski, H. Pohl, B. M. von Reumont, K. Schutte, K. Sekiya, S. Shimizu, A. Slipinski, L. Stamatakis, W. Song, X. Su, N. U. Szusich, M. Tan, X. Tan, M. Tang, J. Tang, G. Timelthaler, S. Tomizuka, M. Trautwein, X. Tong, T. Uchifume, M. G. Waizl, B. M. Wiegmann, J. Wilbrandt, B. Wipfler, T. K. F. Wong, Q. Wu, G. Wu, Y. Xie, S. Yang, Q. Yang, D. K. Yeates, K. Yoshizawa, Q. Zhang, R. Zhang, W. Zhang, Y. Zhang, J. Zhao, C. Zhou, L. Zhou, T. Ziesmann, S. Zou, Y. Li, X. Xu, Y. Zhang, H. Yang, J. Wang, J. Wang, K. M. Kjer, and X. Zhou. Phylogenomics resolves the timing and pattern of insect evolution. *Science*, 346(6210):763–767, November 2014.

[13] László G. Nagy, Robin A. Ohm, Gábor M. Kovács, Dimitrios Floudas, Robert Riley, Attila Gácsér, Mátyás Sipiczki, John M. Davis, Sharon L. Doty, G. Syben de Hoog, B. Franz Lang, Joseph W. Spatafora, Francis M. Martin, Igor V. Grigoriev, and David S. Hibbett. Latent homology and convergent regulatory evolution underlies the repeated emergence of yeasts. *Nature Communications*, 5(1), July 2014.

[14] Ralph S. Peters, Lars Krogmann, Christoph Mayer, Alexander Donath, Simon Gunkel, Karen Meusemann, Alexey Kozlov, Lars Podsiaiowski, Malte Petersen, Robert Lanfear, Patricia A. Diez, John Heraty, Karl M. Kjer, Seraina Klopfeinstein, Rudolf Meier, Carlo Polidori, Thomas Schmitt, Shanlin Liu, Xin Zhou, Torsten Wappler, Jes Rust, Bernhard Misof, and Oliver Niehuis. Evolutionary history of the hymenoptera. *Current Biology*, 27(7):1013–1018, April 2017.
[15] Richard O. Prum, Jacob S. Berv, Alex Dornburg, Daniel J. Field, Jeffrey P. Townsend, Emily Moriarty Lemmon, and Alan R. Lemmon. A comprehensive phylogeny of birds (aves) using targeted next-generation DNA sequencing. *Nature*, 526(7574):569–573, October 2015.

[16] Xing-Xing Shen, Yuanning Li, Chris Todd Hittinger, Xue xin Chen, and Antonis Rokas. An investigation of irreproducibility in maximum likelihood phylogenetic inference. *Nature Communications*, 11, 2020.

[17] Jeff J. Shi and Daniel L. Rabosky. Speciation dynamics during the global radiation of extant bats. *Evolution*, 69(6):1528–1545, June 2015.

[18] Nick Goldman Simon Whelan. A general empirical model of protein evolution derived from multiple protein families using a maximum-likelihood approach. *Molecular Biology and Evolution*, 18(5):691–699, May 2001.

[19] S. Song, L. Liu, S. V. Edwards, and S. Wu. Resolving conflict in eutherian mammal phylogeny using phylogenomics and the multispecies coalescent model. *Proceedings of the National Academy of Sciences*, 109(37):14942–14947, August 2012.

[20] Alexandros Stamatakis, Markus Göker, and Guido W. Grimm. Maximum likelihood analyses of 3,490 rbcl sequences: Scalability of comprehensive inference versus group-specific taxon sampling. *Evolutionary Bioinformatics*, 6:EBO.S4528, 2010. PMID: 20535232.

[21] Steinbruch Center for Computing (SCC). ForHLR - hardware and architecture. https://wiki.scc.kit.edu/hpc/index.php/ForHLR_-_Hardware_and_Architecture, 2020.

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

[23] James E. Tarver, Mario dos Reis, Siavash Mirarab, Raymond J. Moran, Sean Parker, Joseph E. O’Reilly, Benjamin L. King, Mary J. O’Connell, Robert J. Asher, Tandy Warnow, Kevin J. Peterson, Philip C. J. Donoghue, and Davide Pisani. The interrelationships of placental mammals and the limits of phylogenetic inference. *Genome Biology and Evolution*, 8(2):330–344, January 2016.

[24] S. Tavare. Some probabilistic and statistical problems in the analysis of dna sequences. *Lectures on Mathematics in the Life Sciences. Providence: Amer. Math. Soc*, pages 57–58, 1986.

[25] Norman J. Wickett, Siavash Mirarab, Nam Nguyen, Tandy Warnow, Eric Carpenter, Naim Matasci, Saravanaraj Ayyampalayam, Michael S. Barker, J. Gordon Burleigh, Matthew A. Gitzendanner, Brad R. Ruhfel, Eric Wafula, Joshua P. Der, Sean W. Graham, Sarah Mathews, Michael Melkonian, Douglas E. Soltis, Pamela S. Soltis, Nicholas W. Miles, Carl J. Rothfels, Lisa F. Pokorny, A. Jonathan Shaw, Lisa De Gironimo, Dennis W. Stevenson, Barbara Surek, Juan Carlos Villarreal, Béatrice Roure, Hervé Philippe, Claude W. De Pamphilis, Tao Chen, Michael K. Deyholos, Regina S. Baucom, Toni M. Kitchan, Megan M. Augustin, Jun Wang, Yong Zhang, Zhijian Tian, Zhixiang Yan, Xiaolei Wu, Xiao Sun, Gane Ka Shu Wong, and James Leebens-Mack. Phylotranscriptomic analysis of the origin and early diversification of land plants. *Proceedings of the National Academy of Sciences of the United States of America*, 111(45):E4859–E4868, November 2014.

[26] Zhenxiang Xi, Liang Liu, Joshua S. Rest, and Charles C. Davis. Coalescent versus concatenation methods and the placement of amborella as sister to water lilies. *Systematic Biology*, 63(6):919–932, July 2014.

[27] Ya Yang, Michael J. Moore, Samuel F. Brockington, Douglas E. Soltis, Gane Ka-Shu Wong, Eric J. Carpenter, Yong Zhang, Li Chen, Zhixiang Yan, Yinlong Xie, Rowan F. Sage, Sarah Covshoff, Julian M. Hibberd, Matthew N. Nelson, and Stephen A. Smith. Dissecting molecular evolution in the highly diverse plant clade caryophyllales using transcriptome sequencing. *Molecular Biology and Evolution*, 32(8):2001–2014, April 2015.
[28] Ziheng Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods. *Journal of Molecular Evolution*, 39(3):306–314, September 1994.