BIOINFORMATICS

Vol. 00 no. 00 2005
Pages 1–6

Supplementary Material: RAxML-Light: A Tool for computing TeraByte Phylogenies

A. Stamatakis [1,*], A.J. Aberer [1], C. Goll [1], S.A. Smith [2], S.A. Berger [1], F. Izquierdo-Carrasco [1]

[1] The Exelixis Lab, Scientific Computing Group, Heidelberg Institute for Theoretical Studies, Schloss-Wolfsbrunnenweg 35, D-68159 Heidelberg
[2]Blackrim Lab, Department of Ecology and Evolutionary Biology, University of Michigan, 2071A Kraus Natural Science Building, 830 North University Ann Arbor, MI 48109-1048

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

ABSTRACT

In this supplement we discuss a couple of implementation details that may be of interest/relevance to other software developers as well as users that want to get optimal performance from the code.

Availability:
`https://github.com/stamatak/RAxML-Light-1.0.5`
Contact: Alexandros.Stamatakis@h-its.org

1 THE FORK-JOIN MODEL WITH MPI

The fork-join paradigm is typically used for parallelizing code with OpenMP or PThreads on shared-memory architectures. In other words it is *not* the standard programming paradigm when using distributed memory and MPI.

RAxML-Light has different types of parallel regions (e.g., compute a Newton-Raphson step for optimizing a branch, evaluate the log likelihood score at the virtual root of the tree, or compute the conditional probability arrays for all or a part of the nodes in a tree) that require varying amounts of data to be transferred to the worker processes.

When using shared memory this is easy, because the data is already there and can be accessed via the shared memory.

The problem with MPI is that, when doing collective communication operations with `MPI_Bcast()` (to enter a parallel region, i.e., to execute a fork) the size of the data to be communicated needs to be known prior to invoking `MPI_Bcast()`.

However, if we communicate tree traversal descriptors (see Ott *et al.* (2007) and Stamatakis & Ott (2008) for details; a similar data-structure is used in BEAGLE) the length of those traversal descriptors (depending whether we are doing a full or partial tree traversal) will vary.

Hence, normally two `MPI_Bcast()` invocations will be required for each fork at each parallel region, one for communicating the length of the data and type of the parallel region and the second one to actually broadcast the data. This is evidently suboptimal, because collective communication operations are expensive and keep the worker processes from doing actual work, that is, doing likelihood computations.

The problem could be alleviated by applying some trade-off engineering. We analyzed the traversal descriptor length distribution in RAxML and found that the vast amount of traversal descriptors is relatively short, i.e., smaller than 5 to 10 nodes. This is because of the specifics of the RAxML search algorithm (the so-called lazy subtree rearrangements) that usually only apply local changes to the tree and therefore require only partial tree traversals (short traversal descriptors). Similar lazy techniques are used in GARLI and PHYML.

Based on this observation we tried to come up with a latency-bandwidth trade-off (two broadcast operations versus mostly one broadcast communicating more data). The idea is to broadcast a traversal descriptor of a fixed length of 5 entries (5 inner nodes) by default with the first broadcast and only execute a second broadcast if the traversal descriptor actually is longer. This optimization yielded a performance improvement of 10-15% during the initial code development tests.

2 THREAD-TO-CORE PINNING

An issue that users are typically not aware of is the potential negative impact of the thread-to-core pinning when less threads are used than are available on a multi-core node. A detailed analysis of such issues for several diverse scientific applications is provided in (Klug *et al.*, 2011).

Consider the following case: We want to execute the PThreads-version of RAxML-Light with 24 cores on a 48-core machine as used in our experiments in the paper. The two alternative options are to pin the 24 threads to cores $0, 1, 2, ..., 23$ or to cores $0, 2, 4, ..., 48$. In the latter case (pinning to every second core), since two cores are typically sharing a cache memory, each core will have a larger amount of cache memory available and thus each thread/core will potentially have to wait less frequently to retrieve data from RAM. Since likelihood-implementations are typically memory-bandwidth bound because of the linear memory accesses to conditional probability vectors, especially for data with few states (binary or DNA data), the core pinning can have a negative

*to whom correspondence should be addressed

Table 1. Impact of thread pinning and hyperthreading on RAxML-Light Performance

Taxa	Patterns	Data Type	System	Pinning	Execution time
25	55,706	PROT	Magny-Cours	0,1,2,...,23	1099.75 secs
25	55,706	PROT	Magny-Cours	0,2,4,...,46	1137.59 secs
54	552,089	PROT	Magny-Cours	0,1,2,...,23	891.55 secs
54	552,089	PROT	Magny-Cours	0,2,4,...,46	879.68 secs
125	19,436	DNA	Intel Desktop	0,1,2,...,8	176.91 secs
125	19,436	DNA	Intel Desktop	0,1,2,3	187.28 secs
125	19,436	DNA	Intel Desktop	0,2,4,6	304.35 secs

The large 54 taxon dataset was not run to completion and stopped at an intermediate checkpoint.

impact on performance. In particular on DNA data, the number of computations per data access is unfavorable.

As test system we used an Intel 4-core desktop equipped with Intel i7-2600 cores running at 3.40GHz and 16GB RAM. We also used a stand-alone 48-core Magny-Cours server, equipped with 256GB RAM. The test results are summarized in Table 1.

Unfortunately, we are not aware of any portable or easy-to-compile/available on all systems tool or library that can be used under Linux to enforce thread pinning. What has worked well for us are some library calls that can be activated in RAxML-Light by commenting out `#define _PORTABLE_PTHREADS` in source file `axml.c`. This will make RAxML to call the following function in the same source file:

```
#ifndef _PORTABLE_PTHREADS

static void pinToCore(int tid)
{
  cpu_set_t cpuset;

  CPU_ZERO(&cpuset);
  CPU_SET(tid, &cpuset);

  if(pthread_setaffinity_np(pthread_self(),
        sizeof(cpu_set_t), &cpuset) != 0)
  {
   /* print error messages and exit */
  }
}

#endif
```

The above call will pin threads to cores $0, 1, 2, ...$ If modified as follows: `CPU_SET(2 * tid, &cpuset);` it will pin threads to cores $0, 2, 4,$ etc.

Initially, we tested thread pinning impact on the 48-core Magny-Cours server with a protein dataset comprising 25 taxa and 219,023 sites (55,706 distinct site patterns).

With a thread pinning to cores $0, 1, 2, ..., 23$ we executed RAxML-Light as follows:

```
raxmlLight-PTHREADS -T 24 -s ALL.aln -m PROTGAMMAWAG
  -t RAxML_parsimonyTree.START.0  -n T012345
```

and measured an execution time of 1099.75 seconds.

We then changed the core pinning to $0, 2, 4, ..., 46$ and executed RAxML-Light as follows:

```
raxmlLight-PTHREADS -T 24 -s ALL.aln -m PROTGAMMAWAG
  -t RAxML_parsimonyTree.START.0 -n T0246
```

to obtain an execution time of 1137.59 seconds.

As can be seen in this example, the naïve or cache-unaware pinning still performs better than the thread-to-core pinning that provides more cache to each thread.

We then executed RAxML-Light on the larger protein dataset (54 taxa, 552,089 site patterns) as follows:

```
raxmlLight-PTHREADS -T 24 -m PROTGAMMAWAG
  -G 54_962787.phy.binary
  -i 5  -m PROTGAMMAWAG -G 54_962787.phy.binary
  -t RAxML_parsimonyTree.54_962787_ML.RUN.1
  -n BIG_012345689
```

with a pinning to cores $0, 1, 2, ...23$ and obtained an execution time (until the second intermediate tree that was written; not executed to completion) of 891.55 seconds.

We then changed the thread pinning again $(0, 2, 4, ..., 46)$ and executed RAxML-Light as follows:

```
raxmlLight-PTHREADS -T 24 -m PROTGAMMAWAG
  -i 5 -m PROTGAMMAWAG -G 54_962787.phy.binary
  -t RAxML_parsimonyTree.54_962787_ML.RUN.1
  -n BIG_02468
```

to obtain 879.68 seconds until the second intermediate tree. Hence, in this case and on the specific system, thread-to-core pinning related differences are negligible. However, this depends on a lot of hardware parameters such as cache size (of the entire cache hierarchy), the level at which caches are shared, and the quality of the cache-line pre-fetching mechanisms that are implemented in hardware. On older systems (a SUN x4440 with 24 cores) we had observed larger performance discrepancies up to 25%. Thus, this is definitely an issue that should be thoroughly tested and assessed on a per-system basis prior to executing large production runs.

Note that, `-i 5` is a standard RAxML option, while via `-G 54_962787.phy.binary` we read in a pre-parsed binary alignment file that reduces the I/O time for the alignment. The binary alignment file was generated by invoking the *sequential* version of RAxML-Light as follows:

```
raxmlLight -B -m PROTGAMMAWAG -s 54_962787.phy
  -n GENERATE
```

At this point we also need to post a warning about hyperthreading. For RAxML-Light as well as for many other scientific applications, where all threads do homogeneous calculations (competing for the same resources on the processor), hyperthreading does typically not work too well. Unfortunately, the Linux `top` command often tricks users because it shows, for instance, 8 cores (including the virtual ones for hyperthreading) instead of just the 4 physical ones (as on our Intel test system) that are there. Nonetheless, hyperthreading can yield some speedups, but they will evidently not be linear.

On the Intel desktop, we executed the PThreads and AVX-based version of RAxML as follows to test hyperthreading performance:

```
raxmlLight-PTHREADS-AVX -T 8 -s 125
  -t RAxML_parsimonyTree.125.START.0
  -m GTRCAT -n T8
```

and measured an execution time of 176.91 seconds.

We then executed the same code again, but with 4 threads which were pinned to the 4 physical cores $(0, 1, 2, 3)$:

```
raxmlLight-PTHREADS-AVX -T 4 -s 125
  -t RAxML_parsimonyTree.125.START.0
  -m GTRCAT -n T4_0123
```

We measured an execution time of 187.28 seconds. Thus, hyper-threading can yield a slight performance improvement on this specific hardware.

Finally, we executed RAxML-Light with 4 threads again, but pinned only two threads to physical cores and the two other threads to virtual cores:

```
raxmlLight-PTHREADS-AVX -T 4 -s 125
  -t RAxML_parsimonyTree.125.START.0
  -m GTRCAT -n T4_0246
```

We obtained an execution time of 304.35 seconds.

3 INITIAL EXPERIENCES WITH AVX INTRINSICS

Our initial experiences with AVX were rather surprising. The target functions to optimize/vectorize first are the functions called `newview...()` in source file `newviewGenericSpecial.c`. This function computes the conditional probability vector at a node p, given the respective child nodes, q and r and accounts for 60-70% of total execution time. We started vectorizing the relevant functions directly with the AVX intrinsics and not relying on automatic vectorization by the compiler based on previous experiences.

After having vectorized the target functions with the intrinsics, we re-compiled all source files with the `-mavx` flag. However, the overall execution times were worse than for the SSE3-based version of the code. A detailed investigation revealed that, while indeed the `newview...()` functions had become faster due to AVX intrinsics, by defining the `-mavx` flag, the `gcc` compiler tried to automatically vectorize other parts of the code that normally do not contribute that much to the overall run times. However, by automatic vectorization, those insignificant parts of the code became so slow, such that an overall slowdown was observed. To keep the compiler from auto-vectorizing code that it shall not vectorize we moved the AVX-based `newview...()` function implementations to a separate source file and only compile this file with the `-mavx` flag.

This is the AVX version for which we report performance in the paper. Generally, it seems that as longer vector units will become, the more difficult it will be for compilers to automatically vectorize code. The next generation of general purpose CPUs will presumably already have 512-bit wide vector units per core. Thus, we advocate in favor of a more generic programming style, that formulates all computations as vector operations on vectors of arbitrary/user-defined length. Abstract vector operations can then be mapped to the respective intrinsics that are actually used. Such a generic vector-based programming style will also allow for easier and more straight-forward porting of codes to GPUs which are, in essence, very wide vector processors.

Finally, we have also integrated fused multiply-add (FMA) 256-bit vector intrinsics into RAxML-Light. At present these instructions can only be executed by the new AMD Bulldozer processors. The main application of the FMA-based version of RAxML-Light is for benchmarking purposes.

Table 2. Execution times of unvectorized, SSE3- and AVX-vectorized Parsimonator versions

unvectorized	SSE3 intrinsics	AVX intrinsics
4.75 secs	1.54 secs	0.95 secs

All times were measured on an Intel i7-2620M core running at 2.70GHz.

3.1 Parsimony performance of SSE3 and AVX intrinsics

For using RAxML-Light, one has to initially generate a randomized stepwise addition starting tree (or some other reasonable starting tree) either using the dedicated small helper program we have developed called Parsimonator (v.1.0.2, available at `https://github.com/stamatak/Parsimonator-1.0.2`) or the standard RAxML version (`https://github.com/stamatak/standard-RAxML`) with the `-y` option that computes a parsimony tree and then exits.

In principle, parsimony computations just require bit-wise operations (`and, or, nand` etc) and a population count, that is a method that counts the number of set bits in a word of 8 or 16 bytes for instance. Hence, the bit-wise operations provided by AVX or SSE3 intrinsics can easily be deployed to implement fast parsimony functions.

While the search strategy we use is relatively simplistic (randomized stepwise addition followed by a couple of Subtree Pruning and Re-Grafting moves), we believe that the implementations in Parsimonator and standard RAxML currently represent the fastest open-source implementation of parsimony.

We conducted some small tests with Parsimonator to highlight the performance differences between unvectorized, SSE3- and AVX-vectorized code. As hardware platform we used an Intel i7-2620M core running at 2.70GHz (the laptop of A.S.). We used the same DNA test dataset as above with 125 taxa and approximately 30,000 sites.

The test results are summarized in Table 2.

We first executed the non-vectorized version as follows:

```
./parsimonator -s 125.phy -p 12345 -n TEST_1
```

and measured a run-time of 4.75 seconds. Then, we executed the SSE3 version:

```
./parsimonator-SSE3 -s 125.phy -p 12345 -n TEST_2
```

and obtained a run time of 1.54 seconds. Note that, in the best case, we would expect to obtain a four-fold speedup here, since 4 32-bit integers fir into one 128 bit SSE3 register.

Finally, we measured AVX performance by invoking Parsimonator as follows:

```
./parsimonator-AVX -s 125.phy -p 12345 -n TEST_3
```

and obtained a run-time of 0.95 seconds.

The performance of the parsimony functions implemented in standard RAxML is analogous, albeit a bit slower, mainly because the implementation is more generic (it can compute the parsimony score on input data with an arbitrary number of states) and because

there is more overhead involved in reading, error checking, and parsing the input alignments.

4 CHECKPOINTING

Implementing checkpointing for RAxML-Light posed several challenges. Unlike search strategies like MCMC in Bayesian programs or the genetic algorithm implemented in GARLI, that essentially consist of state transitions between generations, such that the state data to be saved always entails the same data structures and variables, hill-climbing algorithms as implemented in RAxML or PHYML are more difficult to checkpoint and restart.

The problem with RAxML is, that as described in the original papers (Stamatakis *et al.* (2005) and Stamatakis (2006)) the search strategy executes two or three distinct iterative hill-climbing steps. Thus, when restarting the code, one has to save additional state data and even local loop variables to be able to jump back into exactly that iterative routine where the checkpoint was last written. While, in principle, this is relatively straight-forward, the potential for integrating bugs is enormous.

Another issue we dealt with was how to store the tree data structure, especially with respect to keeping the nodes ordered in exactly the same way as done internally in RAxML at the point in time where the checkpoint is written. For instance, assume a 4 taxon tree $((A, B), (C, D))$ and two inner nodes p and q. Then, either p can be assigned as parent of (A, B) or as parent of (C, D) when the tree is parsed again. However, because of the specifics of the RAxML implementation, if node p was parent of (A, B) it needs to be read-in or set-up for that matter as parent of (A, B) again, such that its is guaranteed that RAxML will produce exactly the same results when restarted from a checkpoint as a full, complete run would do.

Another issue to deal with is associated with Amdahl's law. One design objective is thus surely to minimize the tree parsing time (since we use the fork-join model only the master process will be parsing the tree) such as to prevent the worker process from waiting for the master to finish parsing. Hence, ideally, the tree should somehow be read in directly using a binary format without the delays associated to actually parsing it.

Related to this, is the checkpointing of branch lengths (64 bit double precision values) that can not be represented exactly as strings (with respect to the numerical values) in a typical Newick tree format.

The solution to this was to directly write the contiguous array of node data structures (containing branch lengths and correctly and consistently ordered inner tree nodes) in a binary file. The only problem is that, the tree data structure is generated by linking those node data structures via pointers, hence the stored address values will be invalid or point to the Nirvana. To solve this, we also store the starting address of the contiguous array of nodes in the checkpoint file and then correct all pointer target addresses by adding/subtracting the offset to the new starting address of the contiguous node array.

Another issue we had to resolve was how to store hash tables when the `-D` option (ML search convergence criterion) is used. The hash table can become very large (in the 116,000 taxon case) and always contains the bipartitions of two trees, the one from the currently best tree and the best tree of the previous iteration of the search algorithm. Search convergence is assessed by computing the RF distance between those two trees; the search stops, when the RF distance is smaller than the admittedly completely arbitrary relative RF distance of 1%. Since the hash table itself can become very large (bipartitions are stored as bit vectors that have # taxa bits) here we decided to actually store the two trees (we are only interested in the topologies here, not the branch lengths or the inner node order) as Newick strings, parse them, upon restart and insert the respective non-trivial bipartitions into the hash table.

The `-D` option was used for all tree searches (and associated restarts) on the 116,000 taxon dataset.

5 116,000 TAXON TREES

The empirical dataset was constructed with the PHLAWD tool Smith *et al.* (2009). As a result of the construction of this dataset, PHLAWD was extended to allow for PThreads and OpenMP parallelization for sequence comparisons and multiple sequence alignments. Ten gene regions were used including 18S, 26S, *atp*B, ITS, *mat*K, *ndh*F, *phy*B, *rbc*L, *rpl*16, and the *trn*L-*trn*F intergenic region. These, concatenated, resulted in a dataset with 116,334 species and 16,079 sites. As discussed in the main text, the requirement to have more than 100,000 species in the dataset resulted in the inclusion of some gene regions with more complex histories in plants (e.g., *phy*B) and with less dense sampling. It also required including 18S and 26S and a large Fungi outgroup. This resulted in some peculiar biological relationships but served as a reasonable benchmark.

5.1 Post-Analyses with RAxML and RogueNaRok

The tree collection of 100 ML trees containing 116,000 taxa allowed us to test the scalability and stability of the post-analysis algorithms implemented in standard RAxML. More specifically, we tested the strict, majority rule, and extended majority rule consensus tree building algorithms on this tree collection.

We also computed all pair-wise RF distances between the 100 trees.

All of these post-analysis runs could be executed and completed with the sequential version of RAxML within less than a day on a single core of one of our 48-core Magny-Cours AMD servers that are equipped with 256GB RAM.

Finally, we also tested the scalability of our new, efficient algorithm (Aberer *et al.*, 2011) for rogue taxon identification on the same machine.

While, we need to re-assemble the datasets, the results of our tests show that, our tools for phylogenetic post-analysis scale to trees with more than 100,000 taxa.

6 ADDITIONAL MEMORY SAVING STRESS TESTS, DETAILS, AND TECHNICAL ISSUES

6.1 Combined Performance of Subtree Equality Vectors and Recomputation Approach

We also tested the stability of combining the Subtree Equality Vector technique with the recomputation technique on the large 116,000 taxon dataset using a single 48/core node. The results are summarized in Table 3.

Table 3. Execution times of RAxML-Light without memory saving techniques, with the SEV technique, and using the SEV as well as the recomputation technique

RAM used	Recomputation	SEVs	execution time
66 GB	OFF	OFF	29,627 secs
26.5 GB	OFF	ON	30,373 secs
19.4 GB	ON ($r := 0.5$)	ON	40,039 secs

Note that, tree searches were not executed until completion, but only until the 2nd checkpoint.

Table 4. Execution times of RAxML-Light using the SEV techniques with standard and multi-threaded memory allocators

Memory Allocator Used	Execution time (seconds)
malloc()	33,430 secs
jemalloc()	18,248 secs

Note that, the program has not been executed to completion.

Initially we executed the MPI version of RAxML-Light without any memory saving flags:

```
raxmlLight-MPI -D -i 25 -s 100K.phy -q 100K.models
  -n 100k_mpi48_std
  -t RAxML_parsimonyTree.100.0.0 -m GTRCAT
```

and obtained the following RAxML log file output until killing the process. Note that, the left column is the execution time in seconds and the right column is the log likelihood score at this point.

```
24218.63 -17189331.99
29626.91 -17189305.77
```

and we measured a memory utilization of 66GB.

Then, we executed RAxML-Light with the Subtree Equality Vector option enabled:

```
raxmlLight-MPI -S -D -i 25 -s 100K.phy -q 100K.models
  -n 100k_mpi48_std_S
  -t RAxML_parsimonyTree.100.0.0 -m GTRCAT
```

and obtained the following RAxML log file output until killing the process:

```
24750.63 -17189331.99
30372.83 -17189305.77
```

We measured a memory utilization of 26.5GB.

Finally, we also deployed the recomputation option by assigning only 50% of the required space for ancestral probability vectors (`-r 0.5`):

```
raxmlLight-MPI -S -r 0.5 -D -i 25 -s 100K.phy
  -n 100k_mpi48_std_S_r50 -q 100K.models
  -t RAxML_parsimonyTree.100.0.0 -m GTRCAT
```

and obtained the following RAxML log file output until killing the process:

```
31740.67 -17189331.99
40039.81 -17189305.77
```

We measured a memory utilization of 19.4GB. Note that, using `-r 0.5` does not mean that the memory footprint is reduced by 50% because storing the tip data and allocating the tree data structure itself requires constant memory (just the input alignment file size is already 2GB).

6.2 Memory Allocators for Subtree Equality Vectors

A technical issue that we have so far, not assessed is the impact of frequent `malloc()` and `free()` calls induced by the Subtree Equality Vector technique (see Izquierdo-Carrasco *et al.* (2011)). The standard implementations of these functions use global locks/mutual exclusion mechanisms that can deteriorate performance.

In these experiments we replaced the standard `malloc()` system call by the dedicated multi-threaded `jemalloc()` call developed by Facebook. The results are summarized in Table 4.

On a 48-core node with `-S` enabled using a gappy, partitioned dataset with 6 partitions, 33,476 taxa and 8502 sites that represents a difficult scenario in terms of very frequent `malloc()` calls, we obtained the following RAxML-Light log-file outputs until killing the processes:

With the standard `malloc()` and the following invocation:

```
raxmlLight-PTHREADS -T 48 -S -s biggeo.phy
  -q biggeo.models -m GTRGAMMA
  -t RAxML_parsimonyTree.st1.0 -n t48_default
```

we obtain:

```
3202.76 -5246558.52
4075.75 -5232593.92
6813.32 -5229403.68
14302.28 -5228344.64
33429.73 -5227982.73
```

With `jemalloc()` using

```
LD_PRELOAD=~/src/jemalloc-2.2.5/lib/libjemalloc.so
```

to replace the standard `malloc()` calls by `jemalloc()` and the following invocation:

```
raxmlLight-PTHREADS -T 48 -S -s biggeo.phy
  -q biggeo.models -m GTRGAMMA
  -t RAxML_parsimonyTree.st1.0 -n t48_jemalloc
```

we obtain:

```
2200.32 -5246558.52
2673.13 -5232593.92
4125.40 -5229403.68
8073.77 -5228344.64
18248.18 -5227982.73
```

Hence using `jemalloc()` can reduce execution times by almost a factor of two.

6.3 Computing a TeraByte Tree on a 256GB multi-core Node

We executed two tests on two distinct (but technically identical) 48-core Magny-Cours nodes with 256GB RAM. We used the MPI version and 44 as well as all 48 cores of these nodes respectively. The results are summarized in Table 5.

Table 5. Execution times of the MPI version of RAxML-Light with and without the recomputation technique

RAM used	Recomputation	# Cores	execution time (3rd checkpoint)
$\approx$ 1TB	OFF	672	25,852 secs
200 GB	ON ($r := 0.15$)	44	180,915 secs
220 GB	ON ($r := 0.2$)	48	163,896 secs

Note that, tree searches were not executed until completion, but only until the 3rd checkpoint.

We did not execute the runs to completion, but just checked that (i) they do not crash and (ii) they yield the same initial log likelihood scores (printed to the `RAxML_log.RUN_ID` file) as the reference run on 672 cores that does not use the recomputation technique.

Below, we provide the log file (left column: seconds of execution time, right column: log likelihood score) output of the 672-core run:

```
20457.97 -5924736430.47
21726.20 -5923054260.35
25851.57 -5923014209.68
37801.40 -5923013987.09
42838.57 -5922998983.27
45021.71 -5922998812.28
46478.81 -5922998783.15
47825.58 -5922998780.54
49117.08 -5922998775.48
50412.50 -5922998591.07
51678.61 -5922998585.29
52970.79 -5922998528.06
56629.70 -5922998515.88
66129.46 -5922605907.38
72415.82 -5922605894.28
78060.29 -5922605894.03
83679.40 -5922605894.03
89951.23 -5922605894.03
105603.20 -5922605894.03
142563.40 -5922605894.03
```

We then executed RAxML-Light using 44 out of 48 cores with OpenMPI and a memory reduction factor of 0.15 (`-r 0.15`) as follows:

```
raxmlLight-MPI -r 0.15 -m GTRCAT -G 1481.phy.binary_new
  -t RAxML_parsimonyTree.1481 -n TEST_1481_015_44
```

and measured a memory consumption of approximately 200GB. A rough extrapolation of the expected run time until convergence on just one instead of 14 nodes yields an approximate run-time of 11.5 days.

The RAxML log file outputs until we killed the process were the following:

```
105005.91 -5924736430.47
121651.50 -5923054260.35
180914.96 -5923014209.68
```

We repeated the experiment on one of our 256GB cluster nodes using all 48 cores with `-r 0.2`, i.e., just allocating 20% of the total space that would be required to hold all ancestral probability vectors in RAM:

We executed RAxML-Light as follows:

```
raxmlLight-MPI  -r 0.2 -m GTRCAT -G 1481.phy.binary_new
  -t RAxML_parsimonyTree.1481 -n TEST_1481_020_48
```

and measured a memory consumption of 220GB.

The RAxML log file outputs until we killed the process were the following:

```
95774.88 -5924736430.47
110764.99 -5923054260.35
163896.06 -5923014209.68
326000.29 -5923013987.09
379180.49 -5922998983.27
402798.86 -5922998812.28
418582.81 -5922998783.15
432932.63 -5922998780.54
```

In this case, a rough extrapolation of execution times with respect to the 672-core run yields an expected run time of about 14 days.

7 TEST DATA USED

The test data used here is available for download at `http://www.exelixis-lab.org/onLineMaterial.tar.bz2`.

The archive contains all datasets and required input files for reproducing our results, except for the three datasets used in this supplement (`ALL.aln`, `54_962787.phy`, `biggeo.phy`) to assess performance impact of alternative thread-to-core pinnings and of using `jemalloc()`. Those datasets are currently unpublished and therefore not available (yet) for public release. However, similar simulated datasets were made available for the following paper Zhang & Stamatakis (2012). The impact of using `jemalloc()` can also be tested with the huge and gappy 116,0000 taxon dataset.

ACKNOWLEDGEMENT

Funding: Part of this work is funded by the DFG grant STA-860/3 and by the NSF iPlant tree of life grand challenge project.

REFERENCES

Aberer, A., Krompass, D. & Stamatakis, A. (2011). RogueNaRok: an Efficient and Exact Algorithm for Rogue Taxon Identification. Technical Report Exelixis-RRDR-2011-10 Heidelberg Institute for Theoretical Studies.

Izquierdo-Carrasco, F., Smith, S. & Stamatakis, A. (2011) Algorithms, data structures, and numerics for likelihood-based phylogenetic inference of huge trees. *BMC Bioinformatics,* **12** (1), 470.

Klug, T., Ott, M., Weidendorfer, J. & Trinitis, C. (2011) autopin–automated optimization of thread-to-core pinning on multicore systems. *Transactions on high-performance embedded architectures and compilers III,* , 219–235.

Ott, M., Zola, J., Aluru, S. & Stamatakis, A. (2007) Large-scale Maximum Likelihood-based Phylogenetic Analysis on the IBM BlueGene/L. In *Proc. of IEEE/ACM Supercomputing Conference 2007 (SC2007)*.

Smith, S., Beaulieu, J. & Donoghue, M. (2009) Mega-phylogeny approach for comparative biology: an alternative to supertree and supermatrix approaches. *BMC Evolutionary Biology,* **9** (1), 37.

Stamatakis, A. (2006) RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics,* **22** (21), 2688–2690.

Stamatakis, A., Ludwig, T. & Meier, H. (2005) RAxML-III: A Fast Program for Maximum Likelihood-based Inference of Large Phylogenetic Trees. *Bioinformatics*, **21(4)**, 456–463.

Stamatakis, A. & Ott, M. (2008) Efficient computation of the phylogenetic likelihood function on multi-gene alignments and multi-core architectures. *Philosophical Transactions of the Royal Society B: Biological Sciences,* **363** (1512), 3977–3984.

Zhang, J. & Stamatakis, A. (2012). The Multi-Processor Scheduling Problem in Phylogenetics. Technical report Heidelberg Institute for Theoretical Studies.