Computing the Phylogenetic Likelihood Function Out-of-Core

Fernando Izquierdo-Carrasco and Alexandros Stamatakis

The Exelixis Lab, Scientific Computing Group, Heidelberg Institute for Theoretical Studies,
Schloss-Wolfsbrunnenweg 35, D-69118 Heidelberg, Germany
{Alexandros.Stamatakis|Fernando.Izquierdo}@h-its.org

Abstract. The computation of the phylogenetic likelihood function for reconstructing evolutionary trees from molecular sequence data is both memory- and compute-intensive. Based on our experience with the user community of RAxML, memory-shortages (as opposed to CPU time limitations) are currently the prevalent problem regarding resource availability, that is, lack of memory hinders large-scale biological analyses. To this end, we study the performance of an out-of-core execution of the phylogenetic likelihood function by means of a proof-of-concept implementation in RAxML. We find that RAM miss rates are below 10%, even if only 5% of the required data structures are held in RAM. Moreover, we show that our proof-of-concept implementation runs more than 5 times faster than the respective standard implementation when paging is used. The concepts presented here can be applied to all programs that rely on the phylogenetic likelihood function and can contribute significantly to enabling the computation of whole-genome phylogenies.

1 Introduction

The on-going accumulation of molecular sequence data that is driven by novel wet-lab techniques poses new challenges regarding the design of programs for phylogenetic inference that rely on computing the Phylogenetic Likelihood Function (PLF [2]) for reconstructing evolutionary trees. In all popular Maximum Likelihood (ML) and Bayesian phylogenetic inference programs, the PLF dominates both, the overall execution time as well as the memory requirements by typically 85% - 95%.

Based on our interactions with the RAxML [6] user community, we find that, memory shortages are increasingly becoming a problem and represent *the* main limiting factor for large-scale phylogenetic analyses, especially at the genome level. At the same time, the amount of available genomic data is growing at a faster pace than RAM sizes.

While we have already addressed potential solutions to this problem by using single precision arithmetics [1] that can reduce memory requirements by 50% as well as by novel algorithmic solutions [7], those approaches remain dataset-specific, that is, their efficiency/applicability depends on the specific properties of the Multiple Sequence Alignment (MSA) that is used as input. Here, we introduce a generally applicable method, that does not depend on any MSA-specific characteristics and which can also be deployed in conjunction with the aforementioned methods to further reduce memory requirements and speed up PLF computations.

In cases where the data structures for computing a function do not fit into the available Random Access Memory (RAM), out-of-core execution may be significantly more efficient than relying on paging by the Operating System (OS). This is usually the case, because application-specific knowledge *and* 'page' granularity can be deployed to more efficiently exchange data between RAM and disk. Since the PLF is characterized by predictable linear data accesses to vectors, as we show, PLF-based programs are well-suited to the out-of-core paradigm.

The remainder of this paper is organized as follows: In Section 2 we briefly discuss related work in the general area of out-of-core computing and some applications to phylogenetic reconstruction using Neighbor Joining which exhibits substantially different data access patterns. In Section 3 we initially outline the necessary underlying principles of the PLF that allow for out-of-core execution and describe the optimization of the proof-of-concept implementation in RAxML. In the subsequent Section 4 we describe the experimental setup and provide respective performance results. We conclude and discuss future work in Section 5.

2 Related Work

The I/O bandwidth and communication between internal memory (RAM) and slower external devices (disks) can represent a bottleneck in large-scale applications. Methods that are specifically designed to minimize the I/O overhead via explicit, application-specific, data placement control and movement (e.g., between disk and RAM) are termed out-of-core algorithms (frequently also called: External-Memory (EM) algorithms; we will henceforth use the terms as synonyms).

EM data structures and algorithms have already been deployed for a wide range of problems in scientific computing including sorting, matrix multiplication, FFT computation, computational geometry, text processing, etc. Vitter provides a detailed review of work on EM algorithms in [8].

With respect to applications in phylogenetics, EM algorithms have so far only been applied to Neighbor-Joining (NJ) algorithms [9, 4]. NJ is fundamentally different to PLF-based analysis (see Section 3). NJ is a clustering technique that relies on updating an $O(n^2)$ distance matrix that comprises the pairwise distances of the n organisms for which an evolutionary tree is reconstructed. The size of this matrix becomes prohibitive for datasets with several thousand organisms. The data access pattern is dominated by searching for the minimum in the $O(n^2)$ distance matrix at each step of the tree building process. We are currently not aware of any EM algorithm for PLF computations.

3 Computing the PLF Out-of-Core

3.1 PLF Memory Requirements & Data Access Patterns

PLF memory requirements and data access patterns are radically different from those observed for NJ algorithms (see Section 2). Instead of a distance matrix, memory requirements are dominated by a set of vectors, that are usually termed ancestral probability vectors. The PLF is defined on unrooted binary trees. The n extant species/organisms of the MSA under study are located at the tips of the tree, whereas the $n-2$ inner nodes represent extinct common ancestors. The molecular sequence data in the MSA that has a length of s sites (alignment columns) is located at the tips of the tree. The memory requirements for storing those n tip vectors of length s is not problematic, because one 32-bit integer is sufficient to store, for instance, 8 nucleotides when ambiguous DNA character encoding is used.

The memory requirements are dominated by the ancestral probability vectors that are located at the ancestral nodes of the tree. Depending on the PLF implementation, at least one such vector (a total of $n - 2$) will need to be stored per ancestral node. For each alignment site $i, i = 1...s$, an ancestral probability vector needs to hold the data for the probability of observing an `A,C,G` or `T`. Thus, under double precision arithmetics and for DNA data, a total of $(n - 2) \cdot 8 \cdot 4 \cdot s$ bytes is required for the most simple evolutionary models. If the standard (and biologically meaningful) Γ model of rate heterogeneity [10] with 4 discrete rates is deployed, this number increases by factor of 4 ($(n - 2) \cdot 8 \cdot 16 \cdot s$), since we need to store 16 probabilities for each alignment site. Further, if protein data is used that has 20 instead of 4 states, under a Γ model the memory requirements of ancestral probability vectors increase to $(n - 2) \cdot 8 \cdot 80 \cdot s$ bytes.

The reason why the PLF is particularly well-suited for out-of-core execution is the regularity and predictability of data access patterns. The likelihood on the tree is computed according to the Felsenstein pruning algorithm [2]. Given an arbitrary rooting of the tree, one conducts a post-order tree traversal to compute the likelihood. The s values in an ancestral probability vector are computed recursively by combining the values in the respective left and right child vectors. Thus, such a tree traversal to compute the likelihood proceeds from the tips towards the virtual root in the tree. The ancestral probability vectors are accessed linearly and the ancestral probability vector access pattern is given by the tree topology.

In general terms, good I/O performance in EM algorithms is achieved by modifying an application such as to achieve a high degree of data locality. For the PLF, the straightforward candidate data structure (the 'page') for transfers between disk and RAM are the ancestral probability vectors. In RAxML they are stored linearly in memory. The replacement strategy and potential future pre-fetching procedure simply needs to exploit the access pattern induced by the tree. Note that, in current ML search algorithms (finding the optimal tree is NP-hard [5]) the tree is not entirely re-traversed for every candidate tree that is analyzed. A large number of topological changes that are evaluated are local changes. Thus, only a small fraction of ancestral probability vectors needs to be accessed and updated for each tree that is analyzed.

The typical minimum HW block is 512 bytes, although some operating systems use a larger block size of 8KB [8]. For the PLF this granularity is not an issue, since a representative ancestral probability vector is significantly larger than the block size. For instance, consider a typical, but still comparatively small, MSA of DNA data with length $s = 10,000$ and $n = 10,000$ species. To compute the PLF, $9,998$ ancestral probability vectors need to be stored. Each of these vectors is stored contiguously in memory and has a size of $10,000 \cdot 8 \cdot 4 \cdot 4 = 1,280,000$ bytes (1.28MB) under double precision arithmetics for a Γ model of rate heterogeneity with 4 discrete rates.

Thus, we can simply set the logical block size b (i.e., the 'page' size) to the size of an individual ancestral probability vector. Therefore, I/O operations can be amortized, that is, each read or write to disk will access a contiguous number of bytes on disk that is significantly larger than the minimum block size.

3.2 Basic Implementation

We store all ancestral probability vectors (see figure 1) that do not fit into RAM contiguously in a single binary file. Although our implementation allows for storing individual vectors in several files, we focus on single file performance, because the performance differences for the two alternatives were minimal (data not shown). We deploy an appropriate data structure to keep track of which vectors are currently available in RAM and which vectors are stored on disk.

Let n be the number of ancestral probability vectors and m the number of vectors in memory, where $m < n$ (i.e., $n - m$ vectors will be stored on disk). Due to the way the likelihood is computed by combining the values of two child vectors for obtaining an ancestral probability vector (see Section 3.1), we must ensure that $m \geq 3$. In other words, the RAM must be large enough to hold at least three ancestral probability vectors. To allow for easy assessment of various values of m with respect to n, we use a parameter f that determines which fraction of required RAM will be made available, that is, $m := f \times n$. Now, let w be the number of bytes required for storing an ancestral probability vector. This allows for enforcing the desired memory limitation to systematically analyze miss rates, that is, our proof-of-concept implementation will only allocate $m \times w$ bytes. We henceforth use the term *slot* (one may think of this as a page), to refer to a segment of available memory (an ancestral probability vector) with a size of w bytes.

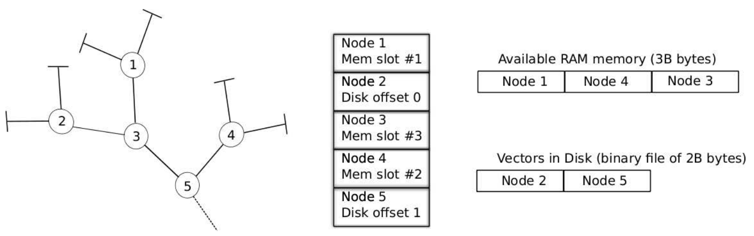

Fig. 1. *Each numbered ancestral node in the tree (left part of figure) needs to hold a vector of doubles. These vectors are either stored on disk or in memory. A node array (middle part of figure) maintains information on the current location (right part of figure) of each ancestral vector. The data structure also has an integer value for each node. When the ancestral probability vector corresponding to the node is available in RAM, this integer is used to index the corresponding RAM starting address. If the vector is on disk, the integer defines the offset of the ancestral probability vector in the binary file.*

To orchestrate data transfers and to control the location of vectors, we use the following C data structure (unnecessary details omitted).

```
typedef struct
{
  FILE                 **fs;
  unsigned int         num_files;
  size_t               slot_width;
  unsigned int         num_slots;
  unsigned int         *item_in_mem;
  unsigned int         num_items;
  unsigned int         disk_items;
  nodemap              *itemVector;
  double               *tempslot;
  boolean              skipReads;
  replacement_strategy strategy;
}map;
```

The array `itemVector` is a list of n pointers (see Figure 1) that is indexed using the (unique) ancestral node numbers. Each entry of type `nodemap` keeps track of whether the respective ancestral vector is stored on disk or in memory. More specifically, if the ancestral vector resides in RAM, we maintain its starting address in memory. If the vector resides on disk, we maintain an offset value for its starting position in the binary file.

We also maintain an array of n integers (`item_in_mem`) that keeps track of which vector is currently stored in which memory slot.

3.3 Replacement Strategies

In the standard implementation of RAxML, where $n = m$, all vectors are stored in RAM. Whenever an ancestral probability vector is required, we simply start reading data from the respective starting addresses for node `i` that is stored in the address vector `xVector[i]`. In the out-of-core version, we use a function `getxVector(i)`, which returns the memory address of the requested ancestral vector for node node `i`. The entire out-of-core functionality is transparently encapsulated in function `getxVector(i)`. The function will initially check, whether the requested vector is already mapped to RAM. If not, it will determine an adequate memory slot (according to some replacement strategy, see below), and swap the currently stored vector in that slot with the requested vector in the binary file.

A constraint for the replacement strategy is that, we must ensure that the 3 vectors required to compute the values at the current ancestral node `i` (vector `i` and the two child nodes `j` and `k`) reside in memory. Using the example in Figure 1, let us assume that we are traversing the tree and that the virtual root is located in the direction of the dotted line. When we need to compute the values for vector 3 (`i`), vectors 1 (`j`) and 2 (`k`) need to reside in memory. Calling `getxVector(1)` will return the address of memory slot#1 (where the vector for node 1 is already available). However, vector 2 may be located on disk. A call to `getxVector(2)` will thus require a swap of vectors, but slots #1 and #3 must be excluded (pinned to memory) from the possible swap candidates, since the values of vectors 1 and 3 are required for the immediate computation. For this reason, `getxVector()` has two additional parameters that specify which inner nodes must be pinned to memory and can not be swapped out.

Even if we optimize data transfer performance between disk and RAM at a technical level, accessing data on disk has a significantly higher latency than accessing data in RAM. Therefore, it is important to minimize the number of I/O accesses (number of ancestral probability vector swaps).

As already mentioned, a vector is always either stored on disk or in RAM. Whenever RAxML tries to access a vector that resides on disk via `getxVector()`, we need to chose a vector that resides in RAM, and then swap the vectors. Therefore, we require a replacement strategy, that is conceptually similar to cache line replacement or page swap strategies.

To conduct a thorough assessment, we have implemented and tested the following four replacement strategies:

Random The vector to be replaced is chosen at random with minimum overhead (one call to a random number generator).

Least Recently Used (LRU) The vector to be replaced is the one that has been accessed the furthest back in time. This requires a list of n time-stamps as well as an $O(log(n))$ binary search for the oldest time-stamp. Note the use of n rather than m, because we only search among time stamps of vectors that are currently in RAM.

Least Frequently Used (LFU) The vector to be replaced is the one which has been accessed the least number of times. This requires maintaining a list of m entries containing the access frequency and an $O(log(n))$ binary search for the smallest value.

Topological The vector to be replaced is the most distant node (in terms of number of nodes along the path in the tree) from the node/vector currently being requested. The node distance between a pair of nodes in a binary tree is defined as the number of nodes along the unique path that connects them.

The rationale for the topological replacement strategy is that, due to the locality of the tree search and the computations, we expect the most distant node/vector to be accessed again the furthest ahead in the future.

3.4 Reducing the Number of Swaps

So far, our EM algorithm has been integrated into RAxML in a fully transparent way. We have shown that it is possible to modify the program and any PLF-based program for that matter, such that the complexity is entirely encapsulated by a function call that returns the address of an ancestral probability vector. However, it is possible to further reduce the number of I/O operations by exploring some implementation-specific characteristics of RAxML, that can also be deployed analogously in other PLF-based implementations.

For each global or local traversal (re-computation of a part or of all ancestral probability vectors) of the tree, we know, a priori (based on the tree structure), that some of the vectors that will be be swapped into RAM will be completely overwritten, that is, they will be used in write-only mode during the first access. Thus, whenever we swap in a vector from file, of which we know that it will initially be used for writing, we can omit reading its current contents from file. We denote this technique as *read skipping* and implement it as follows: We introduce a flag in our EM bookkeeping data structure that indicates whether read skipping can be applied or not, that is whether a vector will be written during the first access. We instrument the search algorithm such that, when the global or local tree traversal order is determined (this is done prior to the actual likelihood computations), the flag is set appropriately.

4 Experimental Setup & Results

4.1 Evaluation of replacement strategies

Assessing the correctness of our implementation is straight-forward since this only requires comparing the log likelihood scores obtained from tree searches using the standard RAxML version and the out-of-core version.

Hence, we initially focus on analyzing the performance (vector miss rate) of our replacement strategies and the impact of the *read skipping* technique as a function of f (proportion of vectors residing in RAM).

To evaluate the replacement strategies, we used 2 real-world biological datasets with 1288 species (DNA data, MSA length $s := 1200$ sites/columns), and 1908 species (DNA data, MSA length: $s := 1424$ sites/columns) respectively. Tree searches were executed under the Γ model of rate heterogeneity with four discrete rates. We used the SSE3-based [1] sequential version of RAxML v7.2.8. The out-of-core version of RAxML and all test datasets are available for download at `http://wwwkramer.in.tum.de/exelixis/src_and_datasets.tar.gz`.

For each of the four replacement strategies, we performed three different runs for the out-of-core version with $f := 0.25$ (25% of vectors memory-mapped), $f := 0.50$ (50% of vectors memory-mapped), and $f := 0.75$ (75% of vectors memory-mapped).

Given a fixed starting tree, RAxML is deterministic, that is, regardless of f and the selected replacement strategy, the resulting tree (and log likelihood score) must always be identical to the tree returned by the standard RAxML implementation. For each run, we verified that the standard version and the out-of-core version produced exactly the same results.

This part of our computational experiments was conducted on a single core of an unloaded multi-core system (Intel Xeon E5540 CPU running at 2.53GHz with 36 GB of RAM). On this system, the amount of available RAM was sufficient to hold all vectors in memory for the two test-datasets, both for the standard implementation or by using memory-mapped I/O for the out-of-core version.

We found that, with the exception of the LFU strategy, even mapping only 25% of the probability vectors to memory results in miss rates under 10%. As expected, miss rates converge to zero as the fraction of available RAM is increased (see Figure 2). In the trivial case ($f := 1.0$), the miss rate is zero, since all vectors reside in RAM. The Random, LRU, and Topological strategies perform almost equally well. Thus, one would prefer the random or LRU strategy over the topological strategy because it requires a larger computational overhead for determining the replacement candidate.

In Figure 3, we show the positive effect of the *read skipping* technique for analogous runs on the same dataset with 1288 species. Here, we quantify the fraction of ancestral vector reads from disk that are actually carried out per ancestral vector access. Note that,

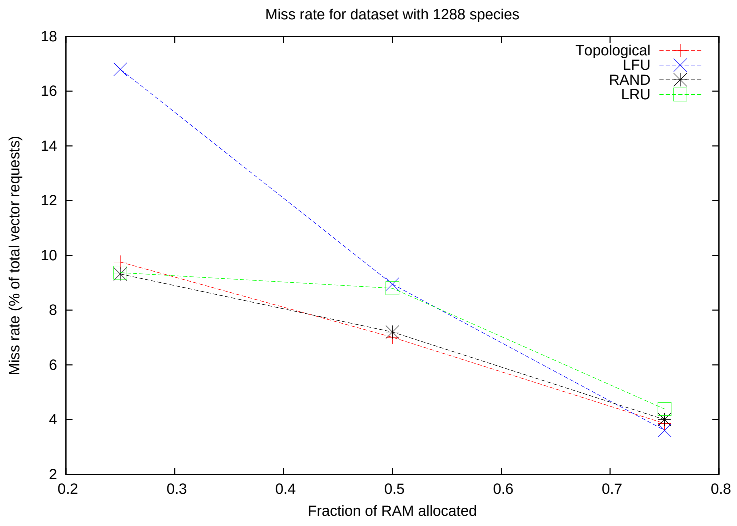

Fig. 2. *Vector miss rates for different replacement strategies using a dataset with 1,288 species. We allocated 25%, 50% and 75% of the required memory for storing ancestral probability vectors in RAM. Every time a vector is not available in RAM, we count a miss.*

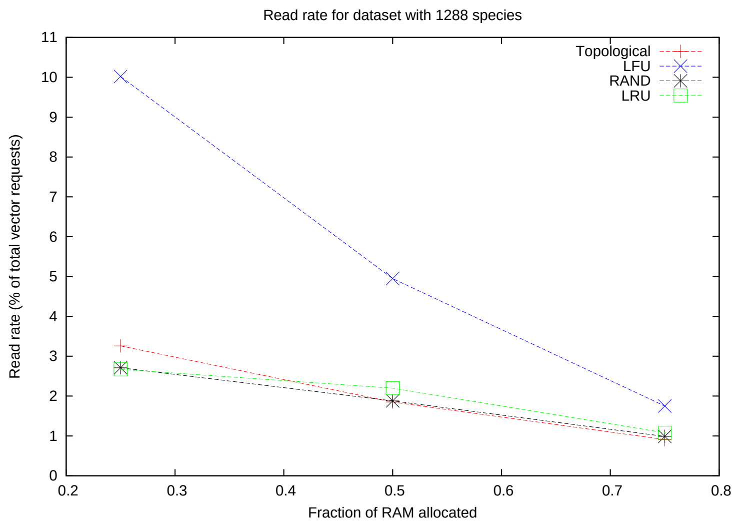

Fig. 3. *Effect of Read skipping: We count the fraction of vector accesses for which a vector needs to be actually read from file using the same parameters as in Figure 2. Without the read skipping strategy the read rate is equivalent to the miss rate.*

without the *read skipping* technique this fraction would be identical to the miss rate in Figure 2. Thus, by deploying this technique, we can omit more than 50% of all vector read operations and hence more than 25% of all I/O operations.

The plots for the dataset with 1908 species are analogous (with slightly better miss rates) to those presented in Figures 2 and 3 and are available as on-line supplement at `http://wwwkramer.in.tum.de/exelixis/supplementOOC.pdf`.

4.2 Miss Rates as a Function of f

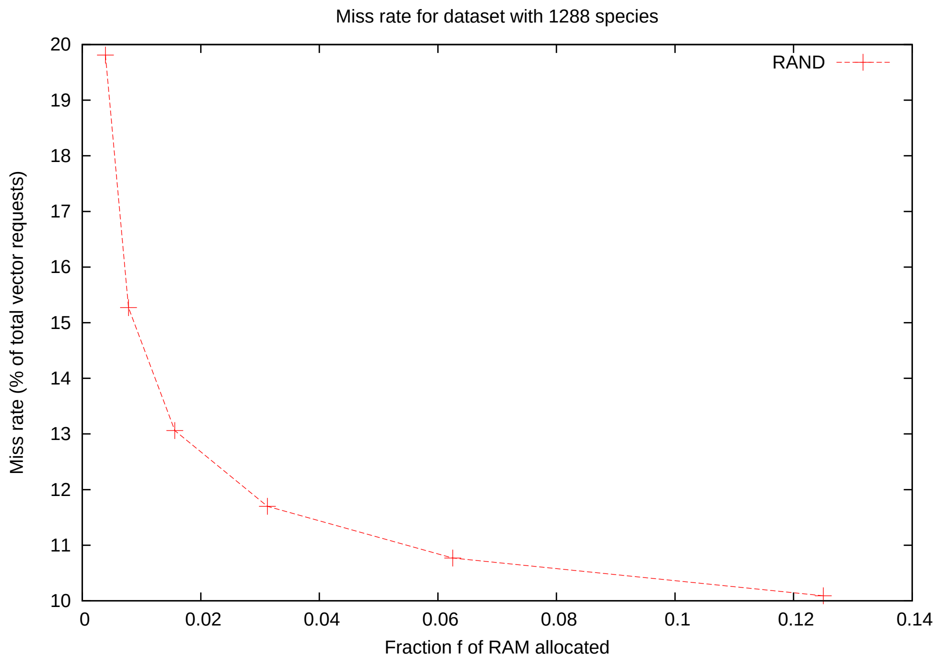

Fig. 4. *Miss rates for several runs using the random replacement strategy on the dataset with 1288 species. The fraction f of memory-mapped ancestral probability vectors was divided by two for each run.*

To more thoroughly assess the impact of f on the miss rate, we conducted additional experiments for different values of f using a random replacement strategy on the test dataset with 1288 taxa. The fraction f was subsequently divided by two. The smallest f value we tested corresponds to only five ancestral probability vector slots in RAM.

Figure 4 depicts the increase in miss rates for decreasing f. The most extreme case with only five RAM slots, still exhibits a comparatively low miss rate of 20%. This is due to the good locality regarding vector usage in the RAxML algorithm. One reason for this behavior, that is inherent to ML programs, are branch length optimization procedures. Branch length optimization is typically implemented via a Newton-Raphson procedure, that iterates over a single branch of the tree. Thus, only memory accesses to the same two vectors (located at either end of the branch) are required in this phase which accounts for approximately 20-30% of overall execution time. In RAxML, access locality is also achieved by —in most cases— only re-optimizing three branch lengths after a change of the tree topology during the tree search (Lazy SPR technique; see [6]).

4.3 Real Test Case

Finally, we conducted realistic tests by analyzing large data sets on a system with only 2GB of RAM. Here, we compare execution times of the standard algorithm (potentially using paging) with the out-of-core performance.

For these tests, we generated large simulated DNA datasets using INDELible [3]. We intentionally generated datasets, whose memory requirements for ancestral probability vectors (see Figure 5) exceed the main memory available on the test system (Intel i5 running at 2.53 GHz with 2GB RAM configured with 36 GB of swap space). To achieve this, we deployed INDELible to simulate DNA data on a tree with 8192 species and varying alignment lengths s. We chose values of s such that, the simulated datasets had (ancestral probability) memory requirements ranging between 1GB and 32GB. Because of the prohibitive execution times for full tree searches on such large datasets, we did not execute the standard RAxML search algorithm. Instead, we executed a subset of the PLF as implemented in RAxML (`-f z` option in the modified code available at: `http://wwwkramer.in.tum.de/exelixis/src_and_datasets.tar.gz`) by simply reading in a given, fixed, tree topology and computing five full tree traversals (recomputing all ancestral probability vectors in the tree five times) according to the Felsenstein pruning algorithm. This represents a worst-case analysis, since full tree traversals exhibit the smallest degree of vector locality. Full tree traversals are required to optimize likelihood model parameters such as the α shape parameter of the Γ model of rate heterogeneity.

The out-of-core runs were invoked with the `-L 1,000,000,000` flag to force the program to use less than 1GB of RAM for ancestral probability vectors.

Figure 5 demonstrates that the execution times of the out-of-core implementation scale well with dataset size (ancestral probability vector space requirements). As expected, the standard approach is faster for datasets that still fit into RAM, while it is more than five times slower for the largest dataset with 32GB. For ancestral vector memory footprints between 2GB and 32GB, the run-time performance gap between the out-of-core implementation and the standard version is increasing, since the standard algorithm starts using virtual memory (e.g., then number of page faults increases from 346,861 for 2GB to 902,489 for 5GB). Note that, for the out-of-core runs, we only use 1GB of RAM, that is, better performance can be achieved by slightly increasing the value for `-L`. Thus, under memory limitations and given the runtimes in Figure 5 using the out-of-core approach is significantly faster than the standard approach for computing the likelihood on large datasets. Thus, given enough execution time and disk space, the out-of-core version can be deployed to essentially infer trees on datasets of arbitrary size. While the increase in execution times in the out-of-core implementation is still large, we are confident that the miss rate and miss penalty can be further improved by low-level

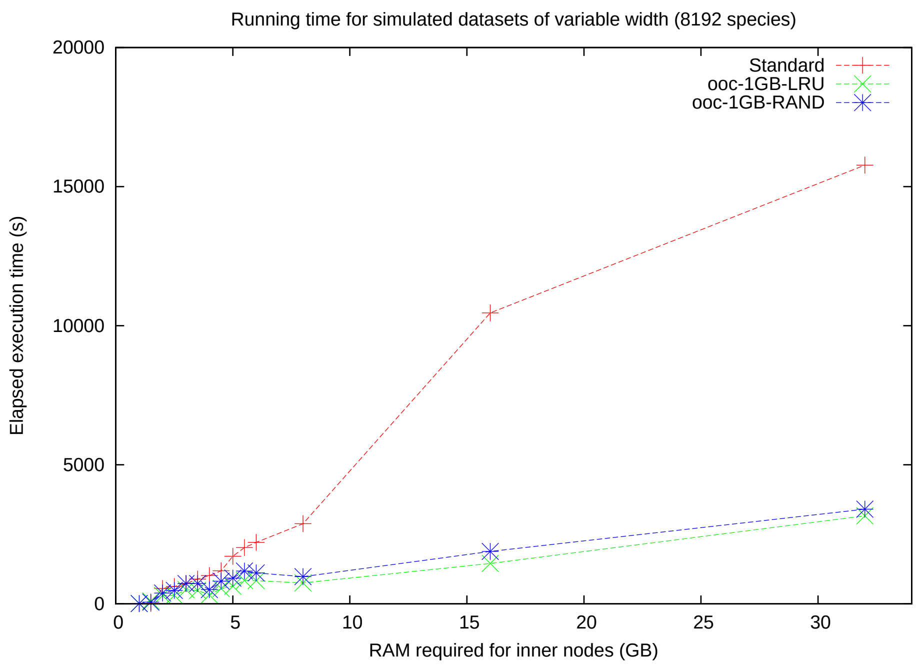

Fig. 5. *Execution times for 5 full tree traversals on a tree with 8192 sequences and variable dataset width s for the standard RAxML version (using paging) and the out-of-core version.*

I/O performance optimization and developing further techniques similar to *read skipping* by using topological information.

5 Conclusion & Future Work

We have presented the first implementation of the PLF that relies on out-of-core execution for accommodating the large memory requirements of ancestral probability vectors. Accommodating such huge memory requirements is necessary for analyzing emerging phylogenomic datasets.

We find that, given the locality of ancestral probability vector access patterns, miss rates are very low, even if the amount of available RAM is limited to a small fraction of the actually required memory. We demonstrate that our out-of-core implementation, performs substantially better than the standard implementation that relies on paging. The concepts developed here can be applied to all PLF-based programs (ML *and* Bayesian) and are not limited to interactions between RAM and the disk. They can also be deployed for exchanging vectors between the relatively small memory of an accelerator card, a GPU or FPGA for instance, and the main memory of a general-purpose CPU. One may also envision a three-layer architecture, where ancestral probability vectors partially reside on disk, in RAM, or the memory of an accelerator card.

Future work will address these issues, but we will also explore two additional directions: We will assess if pre-fetching can be deployed by means of a prefetch thread *and* we will explore the behavior of dedicated parallel file-systems in combination with a fine-grain MPI version of RAxML.

Acknowledgements

Fernando Izquierdo-Carrasco is funded by the German Science Foundation (DFG). The authors wish to thank Julian Kunkel for discussions on optimizing I/O performance.

References

1. S. Berger and A. Stamatakis. Accuracy and performance of single versus double precision arithmetics for Maximum Likelihood Phylogeny Reconstruction. *Springer Lecture Notes in Computer Science*, 6068:270–279, 2010.
2. J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *J. Mol. Evol.*, 17:368–376, 1981.
3. William Fletcher and Ziheng Yang. INDELible: a flexible simulator of biological sequence evolution. *Molecular biology and evolution*, 26(8):1879–1888, August 2009.
4. Thomas Mailund Martin Simonsen and Christian N. S. Pedersen. Building very large neighbour-joining trees. Proceedings of Bioinformatics 2010, to appear in Springer bioinformatics lecture notes.
5. S. Roch. A Short Proof that Phylogenetic Tree Reconstruction by Maximum Likelihood Is Hard. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 92–94, 2006.
6. A. Stamatakis. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006.
7. A. Stamatakis and N. Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics*, 26(12):i132, 2010.
8. Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, January 2008.
9. Travis Wheeler. Large-scale neighbor-joining with ninja. In Steven Salzberg and Tandy Warnow, editors, *Algorithms in Bioinformatics*, volume 5724 of *Lecture Notes in Computer Science*, pages 375–389. Springer Berlin / Heidelberg, 2009.
10. Z. Yang. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites. *J. Mol. Evol.*, 39:306–314, 1994.