Efficient Memory Management in Likelihood-based Phylogenetic Placement

Pierre Barbera
Computational Molecular Evolution Group
Heidelberg Institute for Theoretical Studies
Heidelberg, Germany
pierre.barbera at h-its.org

Alexandros Stamatakis
Institute for Theoretical Informatics,
Karlsruhe Institute of Technology
Karlsruhe, Germany
and
Computational Molecular Evolution Group
Heidelberg Institute for Theoretical Studies
Heidelberg, Germany
alexandros.stamatakis at h-its.org

***Abstract*—Maximum likelihood based phylogenetic methods score phylogenetic tree topologies comprising a set of molecular sequences of the species under study, using statistical models of evolution. The scoring procedure relies on storing intermediate results at inner nodes of the tree during the tree traversal. This induces comparatively high memory requirements compared to less compute-intensive methods such as parsimony, for instance.**

The memory requirements are particularly large for maximum likelihood phylogenetic placement, as further intermediate results should be stored at all branches of the tree to maximize runtime performance. This has hindered numerous users of our phylogenetic placement tool `EPA-NG` from performing placement on large phylogenetic trees.

Here, we present an approach to reduce the memory footprint of `EPA-NG`. Further, we have generalized our implementation and integrated it into our phylogenetic likelihood library, `libpll-2`, such that it can be used by other tools for phylogenetic inference. On an empirical dataset, we were able to reduce the memory requirements by up to 96% at the cost of increasing execution times by ~23 times. Hence, there exists a trade-off between decreasing memory requirements and increasing execution times which we investigate. When increasing the amount of memory available for placement to a certain level, execution times are only approximately 4 times lower for the most challenging dataset we have tested. This now allows for conducting maximum likelihood based placement on substantially larger trees within reasonable times. Finally, we show that the active memory management approach introduces new challenges for parallelization and outline possible solutions.

***Index Terms*—phylogenetics, phylogenetic placement, memory management, parallelization**

I. Introduction

As in most areas of molecular biology, there is also a constant need to develop and adapt algorithms to the steadily increasing data volume in molecular phylogenetics. Work on scalability typically focuses on reducing the execution times of respective data analysis tools, as this constitutes the primary limiting factor for most analyses. However, depending on the resources available, the memory requirements can also constitute a limiting factor. Furthermore, high memory requirements can yield the deployment of hardware accelerators, such as General-Purpose Graphics Processing Units (GPGPUs), challenging as such devices typically have less RAM available than the host system. Even when access to high performance computing systems is available, the memory requirements of specific analyses can still be prohibitive. For example, third generation sequencing will further increase the number of available high length and high quality sequence assemblies, up to and including whole genomes.

For likelihood-based phylogenetic inference (Maximum Likelihood (ML) and Bayesian inference), improving memory efficiency represents a particular challenge due to the way we compute the likelihood on a phylogenetic tree. At each internal node of the tree, we calculate a conditional likelihood for each possible character state (typically DNA or protein data) of a site in the underlying input Multiple Sequence Alignment (MSA) and typically store it as a hard-to-compress floating point value. Realistic models of nucleotide substitution (e.g., the Γ model of rate heterogeneity [1] or other mixture models) further increase the memory requirements as we need to calculate and store conditional likelihood values for several rates of evolution per state, internal node, *and* MSA site. The data structure holding these conditional likelihoods at each inner node of the tree is called Conditional Likelihood Vector (CLV).

Apart from general phylogenetic tree inference, memory efficiency becomes even more performance-critical in the special case of likelihood-based ML phylogenetic placement (henceforth simply denoted as placement), as implemented, for instance, in our tool `EPA-NG` [2]. The goal of placement is to identify a set of likely insertion locations on a given, fixed reference phylogeny for a given query sequence. To accelerate this process, `EPA-NG` calculates and stores CLVs for all possible directions (i.e., all three outgoing branches) at each internal node of the tree in memory. Note that most tree search tools typically only store one CLV per node due to the distinct pattern of likelihood calculations they conduct. This memory organization in `EPA-NG` incurs a significant memory overhead, making placement infeasible for large reference trees [3], [4] containing thousands of sequences/species.

This work was financially supported by the Klaus Tschira Foundation.

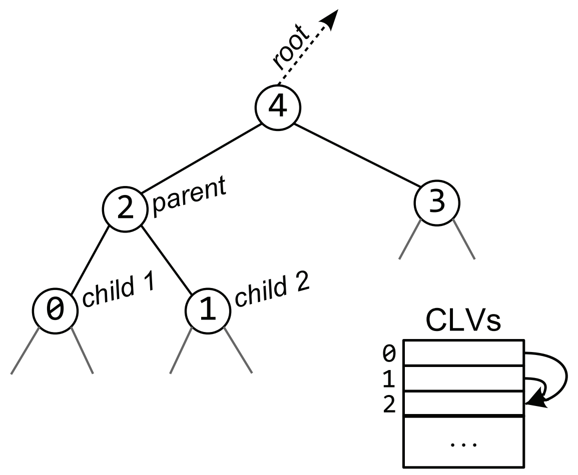

Fig. 1. We illustrate a subtree of the phylogenetic tree during a recursion step of the Felsenstein Pruning Algorithm (FPA). The Conditional Likelihood Vector (CLV) index number of the nodes is displayed inside the nodes. In the FPA step shown here, we combine the CLV information from the two nodes labeled *child 1* and *child 2* in the *parent* node. After this step, the two child node CLVs become obsolete. We repeat this recursion until we reach and calculate the CLV of the root node (not shown, direction of root indicated by arrow), that is then used to calculate the overall likelihood of the tree.

In previous work we have shown that the likelihood of a tree with n leaves can be calculated using a minimum of $log_2(n)+2$ CLVs [5]. Based on this property, we conceived strategies to reduce the overall memory requirements of ML likelihood calculations at the cost of additional computations. However, this approach was never integrated into our production-level tools.

Here, we show how this memory saving technique can be applied to placement and we outline its integration into our production-level placement software EPA-NG. More specifically, we describe the necessary adaptations to the parallelization approach of EPA-NG and we experimentally assess the respective memory versus inference time trade-offs.

II. Methods

The likelihood of a strictly binary tree is computed bottom-up via a post order traversal, starting at the leaves of the tree and moving toward the virtual root via the Felsenstein Pruning Algorithm (FPA) [6]. The virtual root of the tree can be placed into any position of any branch of the tree for the sake of defining a traversal order. Its placement does not alter the likelihood score of the tree as long as the substitution process is time-reversible, which is the case for all standard statistical models of evolution. Each step of the FPA operates on a small subset of the overall tree, where the CLV of a parent node is computed by accessing the CLVs of the two child nodes. The parent CLV summarizes the data/signal contained in the subtree it roots. When this recursive algorithm terminates at the virtual root of the tree, the entries of the CLV(s) at the virtual root are used to calculate the overall likelihood of the tree. We illustrate one step in the FPA recursion in Fig. 1.

In standard phylogenetic likelihood implementations, memory is allocated for all CLVs visited during the FPA, that is, at all inner nodes of the tree. This memory allocation scheme yields 'good' computational efficiency, as a large fraction of CLVs can typically be reused in subsequent likelihood calculations, for instance, when only a part of the tree topology has changed. However, this comes at the expense of an increased memory footprint. In general, this trade-off is justified taking into account the input data sizes of typical phylogenetic analyses and the increasing amount of main memory available in modern computers. However, as already mentioned, for certain likelihood based tools there is pressing need to offer alternative solutions that allow to explicitly limit the amount of memory used by likelihood calculations. This is particularly the case in EPA-NG which stores $3 * (n - 2)$ CLVs, compared to $n-2$ CLVs in most standard phylogenetic tree inference programs. Note that, storing the CLVs clearly dominates the memory requirements of all likelihood-based phylogenetic inference tools.

A solution to reduce the number of CLVs that need to be concurrently held in memory is to exploit the recursive structure of the FPA. In particular, once a parent CLV has been successfully computed from its child CLVs, the data held by the children is no longer needed (hence the name 'pruning algorithm') to compute the overall likelihood of the specific, fixed tree. Thus, the memory allocated to the children CLVs can be overwritten by CLVs entries required at other internal nodes. Hence, for a given tree topology and (virtual) root, there exists some minimum required number of CLVs that need to be held in memory. In [5], we have shown that this minimum number of required CLVs is $log_2(n)+2$ in the worst case for a fully balanced binary tree with n leaves. We will henceforth refer to this method as the *logn* approach.

Further, in [5] we described a data structure and a CLV management approach, to dynamically determine which CLVs to overwrite. Central to this is the concept of a *slot*, which denotes the allocated memory that stores one CLV. Different CLVs occupy this set of slots at different stages of the tree traversal as induced by the FPA. Note that the number of available slots does not need to be set to the minimum of $log_2(n)+2$, but can also be set to a larger value. In particular, it can be set such that the CLV storage space corresponding to the number of slots matches the amount of memory available on the system. When CLVs can be reused between applications of the FPA, which is the case for placement, providing more memory than absolutely necessary reduces the number of CLVs that have to be recomputed. In other words, the memory versus runtime tradeoff can be tuned via the number of available slots.

Here, we implement a generalized version of this Active Management of CLVs (AMC) mechanism (described in more detail in Section IV) into our free, open source library for ML phylogenetic likelihood calculations libpll-2 on which EPA-NG relies. By deploying this approach, the user can now set an approximate explicit limit for the memory footprint in in EPA-NG. This allows for placing sequences on substantially larger reference trees than before.

In placement, the goal is to find a set of most likely edge(s) from a given Reference Tree (RT) that a Query Sequence (QS) belongs to. When using the maximum likelihood approach to placement, we calculate the likelihood of a QS placement as the likelihood of the RT, extended at a given branch by the QS. To save time, EPA-NG pre-calculates and stores the CLVs for each possible insertion branch in the RT in memory. This means that the placement of a QS comprises calculating (i) one CLV summarising the signal from the insertion branch for a specific insertion point along that branch, (ii) setting the CLV at the newly added leaf, and (iii) using these two CLVs to compute the placement likelihood.

While the above implementation exhibits good runtime performance and facilitates parallelization (i.e., offers a high degree of parallelism), it also requires a comparatively large amount of main memory as we need to allocate memory for all possible CLVs in the tree. To address this, we can apply the *logn* approach to substantially reduce the peak memory consumption of EPA-NG.

However, this comes at the cost of increased execution times, as for each iteration over the tree, the per-branch CLVs will have to be re-computed, potentially including the re-computation of CLVs in the respective subtrees defined by this branch. Further, using the *logn* approach to save memory substantially complicates the parallel placement procedure implemented in EPA-NG, as it relies on the assumption that immediate random access to any desired CLV is available at any time. To address this, we now split up the parallel placement of QSs into blocks of RT branches. The CLVs for the next branch block are pre-computed asynchronously, while we work on placing QSs on the current branch block.

Another challenge with such an active CLV memory management approach is to minimize the general computational overhead under memory constraints. EPA-NG utilizes additional memoization techniques, trading additional memory, beyond the CLVs storage space, for increasing speed. More specifically, we use a lookup table that contains constant, precomputed placement results for every branch that allow to rapidly pre-score putative placements. When executing EPA-NG in default mode with memory saving disabled, this lookup table already provides a substantial (~15 fold) speedup. As we show in our experimental results, executing EPA-NG with AMC, using this lookup table improves execution times by up to ~23 times (neotrop data). The reason for this is straight-forward: when using pre-scoring heuristics, the only time every QS is matched against every branch is during this first phase. Branch block precomputation, if done for every branch, has an extremely high computational cost compared to the rest of the program. Thus, we can eliminate the vast majority of the computational effort in the AMC case by using the lookup table, as we will only have to utilize the branch buffer precomputation for the second phase of placement, where each QS only gets matched against a small set of promising branches.

Incidentally, for the typical dataset and using the AMC approach, using the lookup table will result in a dramatic increase in speed. Thus, this lookup table should be used whenever the memory constraints allow for it.

An additional parameter affecting runtime performance is the number of QSs processed per iteration, called the *chunk size*. EPA-NG processes QSs in blocks (i) to overlap I/O with computations and (ii) to limit the impact of the sheer QS data volume on the overall memory footprint. Note that a comprehensive placement based analysis involves on the order of 10^7 QSs or greater [7].

When AMC is enabled, using a larger chunk size decreases the number of times the CLVs of the tree need to be recomputed, as the CLVs have to be recomputed at least once per QS block. However, a higher QS block size also means that there is less memory available for other data structures. This is especially evident for large RT, as there are internal intermediate datastructures that save results for each combination of RT branch and QS, which can occupy a significant fraction of the available memory (see Section V) Hence, there is an additional trade-off to consider here.

III. Related Work

The *logn* approach to performing the FPA was originally implemented as proof-of-concept option in RAxML-Light, where it enabled phylogenetic inference of trees comprising more than $100,000$ leaves [8]. To our knowledge the only other ML phylogenetics software that offers AMC is IQ-TREE 2 [9]. IQ-TREE 2 also explicitly uses the *logn* [5] approach.

With respect to placement software there are, to our knowledge, no other programs that employ an active memory management strategy. Of those based on ML methods [2], [10]–[12], only pplacer offers a dedicated option to handle input data sets with large memory footprints. For large datasets, the user can specify the location of a memory-mapped file, which will be used for larger memory allocations, thereby reducing the peak main memory consumption. Consequently, the runtime performance of this approach depends on the latency and bandwidth of the underlying file system.

For RAPPAS [12], the operation of the program is split up into two phases. First, a database is constructed, involving the calculation of posterior probabilities for ancestral sequences on the given reference tree. For executing this step, the user can chose among several ML phylogenetic inference programs. Hence, the program choice directly influences the memory requirements. The constructed database is then used in the second phase to perform QS placement. While the database-construction phase exhibits the overall peak memory consumption, the intended use case is to build the database once (perhaps on more powerful hardware), and to subsequently use it to perform multiple placement runs on the same, fixed RT represented by the database.

Finally, there exist several placement tools that do not rely on ML methods [3], [4]. Characteristic for these programs is their extremely low memory consumption, that is typically several orders of magnitude lower than for ML methods. For example, APPLES [3] is a distance-based approach that uses least-squares minimization to determine the placement of a QS on a RT. APPLES was used to perform placement on a tree with $200,000$ leaves, by only using ~4GiB of main memory.

The most recent addition to placement methods is App-SpaM [4], which uses a phylogenetic distance based on filtered spaced word matches [13].

IV. Implementation and Parallelization

Our AMC implementation in libpll-2 comprises two major components, which we illustrate in Fig. 2. The first component is the mapping of a potentially large number of global CLVs to a substantially smaller set of physical memory locations available to hold them, called *slots*. This can be efficiently implemented by using two arrays that map the global index of a CLV to its slot index, and vice versa. When a slot is currently not associated with a CLV index, or when a given CLV index is not present in memory (not *slotted*), we use dedicated values to denote these special states.

The second major component is the mechanism for choosing which slotted CLVs to overwrite. This slotted CLV overwriting mechanism is in some ways analogous to cache line replacement policies, but under additional constraints as we can not overwrite any slotted CLV we wish, due to the tree traversal order that defines the data access pattern. Hence, designing an appropriate overwriting strategy is not trivial and can evidently have implications on runtime performance. It is beneficial to retain slotted CLVs that are probable to be accessed again in the near future during subsequent likelihood calculations. The overwriting strategy heavily depends on the order of CLV operations which might be highly program-/algorithm-specific. Thus, we have implemented a generic replacement strategy interface via a set of callback functions that allow the developer to fully customize how a slot is chosen/overwritten. As default strategy, we have implemented an algorithm that calculates the approximate cost for recomputing a given CLV, and that chooses which to replace based on this cost. We approximate the recomputation cost of a CLV as the number of descendant leaves the CLV summarizes (i.e., the size of the subtree).

Finally, an additional mechanism for maintaining consistency is required which is called *pinning*. As mentioned before, some intermediate CLVs *must* remain slotted/pinned to the slots during the FPA that traverses the tree in post-order to be able to compute the likelihood score. We illustrate this mechanism via the tree shown in Fig. 1. We assume that we have just calculated the CLV of node 2. Next, the FPA recurses into the subtree rooted by node 3. However, we can not yet discard the result stored at node 2, and thus need to pin the associated memory slot as it will be required to compute the CLV at node 4. The pinning mechanism can also be exploited beyond a single tree traversal in order to retain and re-use CLVs among successive tree traversals. However, care has to be taken to not pin too many slots, as this could cause the FPA to fail if an insufficient number of unpinned slots is available.

As EPA-NG operates on a static tree (i.e., the underlying RT is fixed and does not change), we are able to utilize the CLVs pinning mechanism between successive iterations over the tree to minimize recomputation cost. This is of particular use for the aforementioned branch block precomputation. Here, we traverse the RT, and for each branch we visit, we recompute the two CLVs at either end of the branch. To do so, we first determine on which CLVs in the respective subtree defined by that branch these depend, and consequently also have to be computed. For the CLVs in the subtrees, we construct a list of those that are currently slotted, along with a value reflecting their approximate recomputation cost. From this list we retrieve the entries with the highest recomputation cost, and subsequently pin the corresponding CLVs to their slots. We choose the number of slots we pin such that after this high-cost CLV pinning step, the number of unpinned slots is at least $log_2(n) + 2$ (Section I), which ensures that we can successfully execute the FPA.

The AMC strategy also has implications on the degree to which we are able to effectively parallelize the code. Normally, parallelization of placements is straight-forward: we merely perform the core placement procedure for each QS and branch pair to be evaluated via random accesses to the corresponding precomputed CLVs. However, when the AMC strategy is enabled, the number of these fine-grained placement tasks is limited by the block of branches (subset of branches in the RT) and corresponding CLVs available to the current iteration. Further, making available in memory the CLVs of the subsequent branch block is limited by the *logn* limited FPA execution, which is all but straight-forward to parallelize efficiently. As a consequence, this branch block precomputation for the subsequent branch block can constitute a bottleneck. This is especially true when we can not deploy the preplacement lookup table (Section II). In this case, the computational effort required to compute the CLVs of branch blocks dominates the execution time.

V. Experimental Setup and Results

We assessed the performance of EPA-NG with AMC enabled on 3 representative empirical datasets with distinct characteristics. We list these datasets in Table I. In this table, we show the number of leaves of the RT, the number of sites in the alignment of the reference sequences with the QS, the type of the underlying data (NT for nucleotide, AA for Amino Acid), as well as the reference to the source of the data.

The *neotrop* dataset covers the QS number/volume dimension. This dataset was used to evaluate the microbiome of neotropical soils, including a reference tree that was tailored to the studied environment [7]. The corresponding reference alignment and QSs are 16S rRNA nucleotide data, and their $95,417$ query sequences.

The *serratus* dataset aims to cover the alignment width and resulting CLV-size dimension by using a reference alignment with $10,170$ amino acid sites. This dataset was the result on our work on the Serratus open science project [14] which uncovered new sequence diversity from the Sequence Read

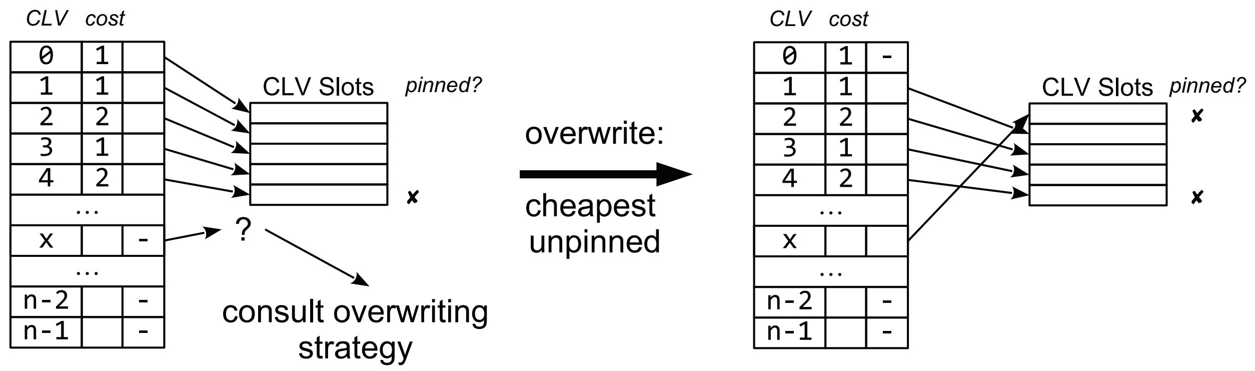

Fig. 2. Illustration of the Conditional Likelihood Vector (CLV) management and replacement strategy. We show a snapshot of the CLV management data structures. These include the *cost* of each CLV, which we approximate by the number of descendant nodes that the CLV summarizes (i.e., the nodes in the subtree rooted by the CLV). We also show the mapping of CLV indexes to their locations in physical memory, called CLV *slots*. If a CLV has not been assigned to a slot, it is marked by (-). For each slot, we also record if it is *pinned*, meaning that it can not be overwritten. Note that these illustrations correspond to the tree shown in Fig. 1. On the left, we show the situation where CLV `x` has to be computed and stored, while we have not yet assigned a physical memory location for `x`, and all slots are occupied. Thus, we need to invoke the *overwriting strategy* to select an appropriate slot among the unpinned slots. The strategy is to select the slot occupied by the CLV with the smallest recomputation cost. On the right we show the result of this operation: we assign `x` to the slot that previously belonged to the CLV with ID `0`. Finally, we also mark this slot as pinned, as this example is part of FPA execution where `x` is a current *parent*.

Archive [15]. More specifically, the reference alignment spans 546 sequences from the *Coronaviridae* virus family. Here, the QSs only comprise 136 RdRP sequences from assembled genomes that showed high sequence similarity to the Coronaviridae family.

Lastly, the *pro_ref* dataset covers the RT-size dimension. This dataset comprises the largest default RT from the `PICRUSt2` software, spanning $20,000$ reference sequences [16]. To this, we added a set of $3,333$ 16S QSs, sampled from the wild blueberry rhizosphere [17].

All scripts and datasets used for our experiments are available online at: https://github.com/pbdas/memsaver-paper

TABLE I
DATASETS

Name	leaves	sites	#QSs	type	reference
neotrop	512	4,686	95,417	NT	[7]
serratus	546	10,170	136	AA	[14]
pro_ref	20,000	1,582	3,333	NT	[16], [17]

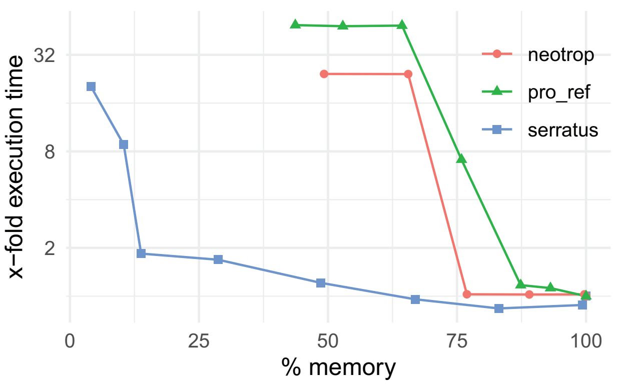

Fig. 3. Runtime characteristics of varying memory limitation settings (`--maxmem`), relative to the fastest execution *without* active CLV management (reference execution). The x-axis indicates the memory limitation setting as a percentage of memory used by the reference execution. The y-axis indicates the slowdown factor relative to the reference execution. The sharp sudden decline in execution times occurs when the memory limit is large enough to allow using the preplacement lookup table.

A. Execution Time

To assess the memory versus runtime trade-offs of `EPA-NG`, we used the `PEWO` testing framework [18]. In `PEWO`, we have extended an already available workflow to measure the runtime and the memory footprint with our new `EPA-NG` memory saving mode.

In our tests, we evaluated how constraining the memory available to `EPA-NG` increases overall execution times. We performed placement on the datasets described in Table I for different maximum memory settings (set by `--maxmem`) in each run. Every `--maxmem`/dataset configuration was executed five times, and the results we show are calculated as the mean of all five runs, both for execution time and memory footprint. Note also that `PEWO` limits individual executions to one worker thread per run (i.e., `--threads 1` for `EPA-NG`). We denote the fastest run using the default `EPA-NG` parameters (i.e., with AMC *disabled*) as *reference run*.

The results of these tests are shown in Fig. 3. In this graph we show the fraction of memory used, compared to the memory required by the reference run on the x-axis. On the y-axis we show the execution time slowdown of each run, relative to the reference run. For legibility we have scaled the y-axis using the binary logarithm.

Additionally, we show absolute execution time and memory footprint values for these runs in Table II. Here we distinguish between the reference runs which we mark *O*, the runs using AMC to the fullest extent (i.e., with the greatest memory limitation possible) marked *F*, and an intermediate setting marked *I*. We chose the intermediate run such that it represents the setting, unique to each dataset, for which we still observed comparatively low execution times (i.e., before the execution time rises sharply).

TABLE II
MEMORY FOOTPRINT AND EXECUTION TIME WITHOUT (O), WITH FULL (F), AND WITH INTERMEDIATE (I) AMC

dataset	**time (s)**			**memory (MiB)**		
	O	I	F	O	I	F
neotrop	160	165	3,908	416	319	205
serratus	18	34	370	6,344	875	258
pro_ref	104	123	5,134	8,701	7,600	3,800

We observe two distinct characteristics.

Firstly, there is a high variance with respect to the lowest possible memory footprint we can achieve. For the serratus dataset we are able to reduce the memory footprint to 4% of the reference run. In contrast, for the neotrop dataset we are only able to decrease the memory footprint to 48.7% of the reference run. This limitation of the lowest possible memory setting is in part due to the QS chunk size (see Section II), which, in this set of experiments, was set to the default value of 5,000 QS per chunk. A larger chunk size increases the size of intermediate result structures, which EPA-NG allocates proportionally to the number of QSs in each chunk. Thus, decreasing the chunk size, that is, reducing the number of QS that are processed in one pass over the tree allows to further decrease the minimum required memory footprint, but at the expense of increased execution time. We show this relationship in Fig. 4.

Secondly, as we approach this minimum possible memory footprint, there is a sharp, sudden increase in execution time. This effect is caused by not being able to allocate the lookup table memoization (Section II) any more when the available memory does not allow for it.

To further showcase the previously mentioned impact of the chunk size on both, the minimum possible memory footprint, as well as the increase in execution time, we repeated the above experiment using a chunk size of 500. For this additional experiment we also repeated the reference runs using the lower chunk size. We show the results of this test in Fig. 4. In general, we observe an analogous behavior as for the initial experiment. As expected, we now also observe a lower minimum memory footprint of ~25% for both, the neotrop, and pro_ref data. We also observe that the increase in execution time for the neotrop data remains largely unaltered. Both, in this test, and the initial test, we measured a ~23-fold increase in execution times relative to the respective reference runs. In contrast, for the pro_ref data and a chunk size of 5,000,

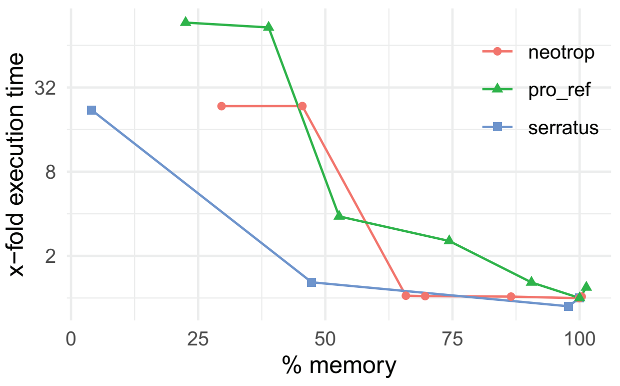

Fig. 4. Runtime characteristics of varying memory limitation settings, using a lower QS chunk size. The tests are identical to those presented in Fig. 3, with the exception that the QS `--chunk_size` was lowered from the default of 5,000, to 500. This illustrates the improved lowest possible memory footprint, and the impact of chunk size on the memory footprint, as well as the execution time.

we observe a ~49 fold increase in the execution time over the reference run. For the same data and with a chunk size of 500, we observe a ~90 fold increase in the execution time over its reference run. To provide some perspective, this particular run took ~2.4 hours on a single core, which should not present an issue to most users.

This substantial deviation between the behavior of the neotrop and pro_ref analyses is due to the large difference in respective RTs sizes (512 versus 20,000 taxa), as the latter requires an increased number of CLV (re-)computations. The results for the serratus data remain unchanged, as for both chunk size settings, all QSs fit into a single chunk. Note also that one pro_ref datapoint shows both, a higher memory consumption as well as a slightly increased execution time compared to the reference run. We attribute this behavior to imperfect memory consumption accounting of our implementation. This accounting is the basis of a memory budgeting with which we determine the amount of resources we are able to spend before exceeding the user set limit. Thus, a flaw in this accounting can lead to a larger than desired memory footprint. We intend to address this issue in the future.

Finally, as expected, execution times improve with increasing memory available. Once the memory limit allows allocating the lookup table memoization, the execution time rapidly approaches the execution time of the reference run. As described in Section II, computing the lookup table is a one-time computational overhead. Subsequently it is used to accelerate *all* QS pre-placement operations. Thus, if the QSs of a chunk are pre-placed on a small common subset of RT branches, the high computational overhead of successive re-computations of reference CLVs is substantially reduced. Analogously, if there is only one single QS chunk to be processed, the run time impact of re-computing the CLVs in the RT is less pronounced.

B. Comparison with `pplacer`

Next, we showcase the capabilities of `EPA-NG` when using AMC, compared to the closest competitor software `pplacer` [11].

To our knowledge, `pplacer` is the only other ML phylogenetic placement software that offers an option to reduce the memory footprint. `pplacer` does so by allocating a significant part of the required memory using a memory-mapped file, effectively extending the available main memory by using disk space. This is an on/off approach, meaning it does not allow the user to more finely control the impact on execution time.

For our showcase test, we chose the two datasets that have the highest memory footprint (serratus and pro_ref). For both datasets, `EPA-NG` requires more than 4GiB of memory, which is a common main memory size for older personal laptops, and which we believe represents a common limitation for users with limited access to newer hardware. Thus, the goal of this test was to show how limiting the memory to 4GiB affects the execution times of `EPA-NG` and `pplacer`.

We show the results of this test in Fig. 5. We ran both softwares using their default parameters, with the exception of limiting the chunk size of `EPA-NG` to 500. We ran each combination of dataset and placement tool both with and without memory saving techniques enabled. As before, each run was repeated five times, and we report the mean of the runs. Arrows indicate the change in memory footprint and execution time going from memory saving disabled to having memory saving enabled.

We observe that, for the same data, `EPA-NG` performs significantly better than `pplacer` in both memory consumption and execution time, both for memory saving disabled and enabled runs. With their memory saving enabled, `pplacer` achieves a significant relative reduction in memory consumption at a moderate cost in execution time. However, when `pplacer` has memory saving enabled, its memory consumption is ~2-3 times higher than `EPA-NG` with its memory saving techniques (AMC) *disabled*.

Regarding `EPA-NG`, as in previous tests, we again observe the significant difference in the execution time impact of AMC between the serratus and pro_ref datasets.

C. Parallel Efficiency

Due to our adapted parallelization approach for the memory saving option, we also re-evaluated the per-node Parallel Efficiency (PE) of `EPA-NG`. For each dataset, we evaluated the PE under three scenarios: limiting the memory as much as possible (*full*), limiting the memory such that the maximum number of slots can be allocated (i.e., three CLVs per inner node), resulting in approximately the same memory footprint as without memory limitation (*maxmem*), and not limiting the memory at all, that is, disabling the memory saving mode (*off*). Furthermore, we again choose the fastest out of five runs run as the representative datapoint. We compare the run time of each parallel configuration to the fastest *serial* run under the same configuration. For the serial runs under each of the above three configuration, we compiled a dedicated version of `EPA-NG` under a setting that disables any kind of multithreading (`EPA_SERIAL=1 make ...`). We performed all tests on a shared memory system with 48 physical cores (two Intel® Xeon® Platinum 8260 Processors), and 754GB total available RAM.

The results of the PE test are shown in Fig. 6. In this graph we show the number of threads used in each run on the x-axis. Note that, when AMC is enabled, the asynchronous CLV precomputation means that we are using one additional worker thread. Thus, results using the *full* or *maxmem* setting include one additional thread. On the y-axis, we show the PE. To calculate the PE, we first calculate the parallel speedup $S(r)$ of a run r with execution time $T(r)$ as

$$S(r) = \frac{T(s)}{T(r)} \tag{1}$$

where s denotes the serial run, and $T(s)$ the execution time of the serial run, respectively. Subsequently we calculate the Parallel Efficiency (PE) $E(r)$ of a run r as

$$E(r) = \frac{S(r)}{P(r)} \tag{2}$$

where $P(r)$ denotes the total number of threads/processors utilized in the run.

When AMC is disabled, we observe similar results to our previous evaluation of `EPA-NG` in [2]. Further, we observe that PE improves with an increasing number of QS.

In contrast, when AMC is enabled, the PE decreases substantially. This is due to the overhead of CLV recomputations for the branch buffer, which is only parallelized insofar as running on a separate, asynchronous thread. The goal of this parallelization approach was to overlap the CLV computation for one branch block with the QS placement computations of another branch block. The key limiting factor is, that based on our experience with concurrent likelihood calculations, parallelizing the CLV recomputation per se on typical reference datasets, is difficult. Hence, the PE of such an additional parallelization at this level is expected to be sub-optimal. We nonetheless investigated the impact on PE when using a version of `libpll-2` that parallelizes CLV calculations over individual sites of the reference alignment. For this, we modified `EPA-NG` to perform the branch buffer precomputation *synchronously*. In this version, we first use *all* available worker threads for the CLV precomputation of one block and subsequently use *all* worker threads to perform the placement operations on this block. We show the results of this test in Fig. 7.

Due to time constrains we were only able to obtain data for the serratus dataset, which also presents the most promising candidate for this type of parallelization scheme. For this dataset we observe a significant improvement in PE. When using 32 threads and the *full* mode, the asynchronous approach showed ~4% PE, whereas our experimental approach yields ~16%. The *maxmem* setting performed nearly identical

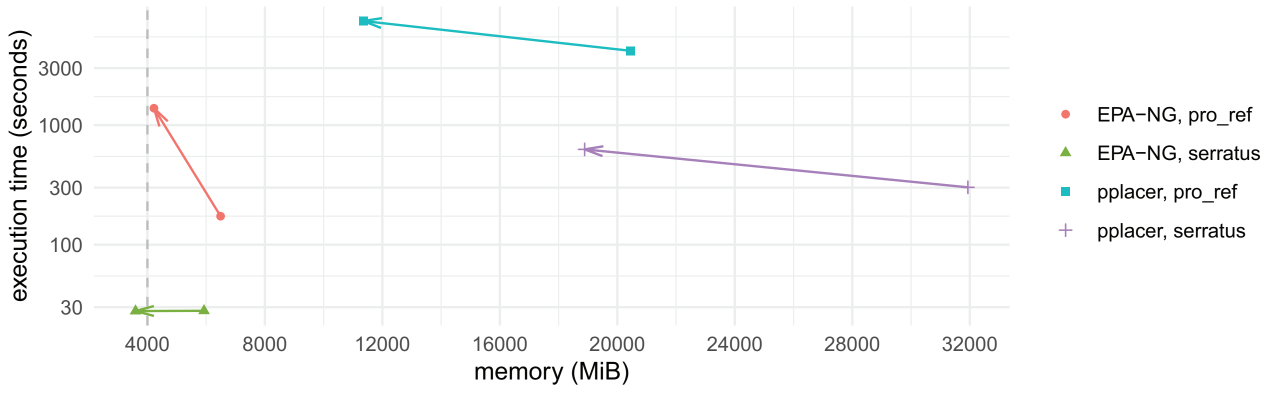

Fig. 5. Showcase comparison between `pplacer` and `EPA-NG`. We show runs for two different datasets (serratus and pro_ref). We ran each combination of dataset and placement tool both with and without memory saving techniques enabled. Arrows indicate the change in memory footprint and execution time going from memory saving disabled to having memory saving enabled.

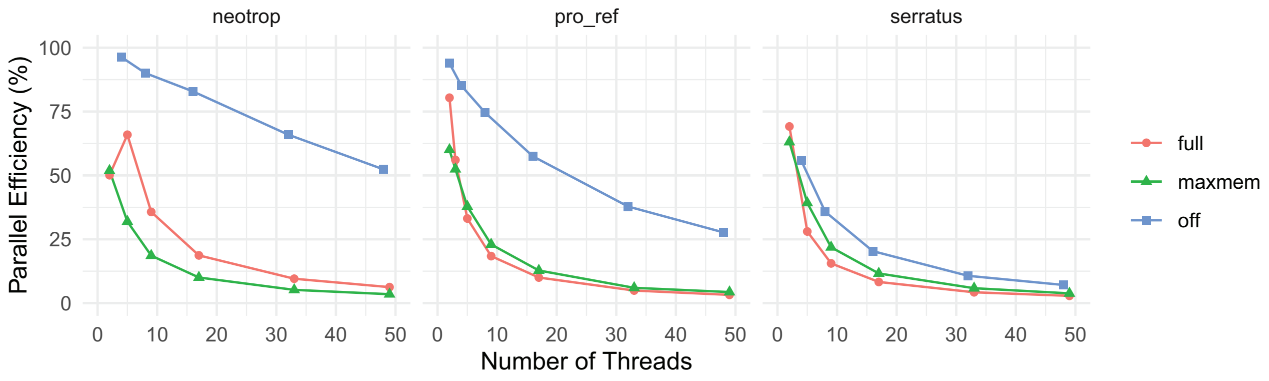

Fig. 6. Parallel efficiency, measured across the three datasets, with varying memory limitation settings: no Active Management of CLVs (AMC) (off), minimum memory AMC (full), and maximum memory AMC (maxmem).

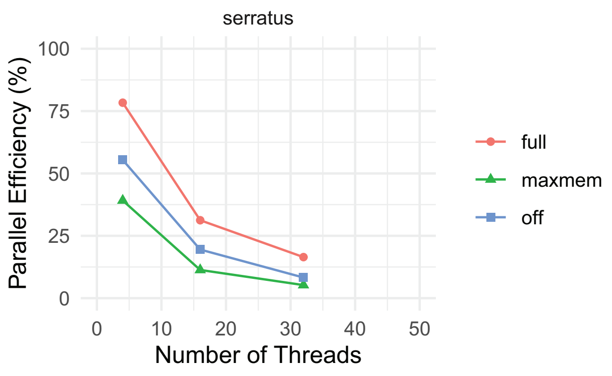

Fig. 7. Parallel efficiency using an experimental across-site parallelization scheme to accelerate branch buffer precomputation. We measured the serratus dataset, with varying memory limitation settings: no Active Management of CLVs (AMC) (off), minimum memory AMC (full), and maximum memory AMC (maxmem).

for 32 threads for both the asynchronous and experimental approaches. However for the same number of threads and when not using AMC, the asynchronous approach had a PE of ~10%, whereas the experimental approach achieved ~8%. Note however, that the serratus data presents the rather atypical case of a very wide alignment. Wide alignments with a large number of alignment sites are known to increase the efficiency of the across-site parallelization approach we deploy.

In contrast, supplying too few sites per thread can be detrimental to the overall execution time [19]. Our preliminary results for combining the experimental parallelization suggest the same behaviour for the neotrop dataset. With the neotrop data and the *full* mode, we observed the run using 32 threads to be ~20% slower than the serial reference. Thus, we can only recommend an across-site parallelization for sufficiently wide alignments.

D. Verification

Both `EPA-NG` and `libpll-2` include regression testing to ensure the validity of any changes made to the code.

Additionally, EPA-NG includes a suite of unit tests. We executed all available tests for the versions of EPA-NG we evaluated here, and found no fault or deviation from previous results.

VI. Conclusion and Future Work

In this work we presented the implementation of a novel memory saving approach for likelihood-based placement by example of EPA-NG. By default, this approach does not require any sort of user intervention as the limit for the amount of available memory is determined automatically by our program. However, when the user desires to do so, he/she can explicitly set a specific memory limit via a simple command line option (`--maxmem`).

Thereby, we enable maximum likelihood based phylogenetic placement on large scale datasets under resource limitations and/or on extremely large reference trees. We have shown that, while in some cases, the slowdown of EPA-NG induced by aggressive memory saving can be substantial, this only presents the most extreme case. In particular, when the allotted memory allows for using the EPA-NG memoization technique, the memory saving to runtime trade-off is acceptable and practical. For example, using the most aggressive memory saving setting, when applied to the pro_ref data, we are able to reduce the memory footprint by $\sim 77\%$, increasing the execution time to 2.4 hours. With the memoization technique enabled, we are able to reduce the execution time to ~ 6 minutes, while still reducing the memory footprint by $\sim 43\%$. This holds in particular for large datasets where otherwise using EPA-NG would merely not have been possible due to memory limitations.

As we have generalized and implemented the CLV memory management in our free, open source phylogenetic maximum likelihood library libpll-2 [20], other likelihood-based tools such as, for instance, RAxML-NG [19] can now also deploy this technique.

In the future, we will exploit the new ability to explicitly decrease the memory footprint of EPA-NG for GPGPU integration, as on-card memory is typically limited. We will further investigate alternative options to improve the parallel efficiency under the memory saving option.

Finally, we believe that there exists potential for further improving the execution times under memory constraints, both in terms of improving the memory saving-specific changes to the parallelization strategy, as well as by using different (e.g., adaptive or machine learning based) replacement strategies.

Acknowledgment

We extend our thanks to Benjamin Linard for his help in extending the PEWO framework to incorporate our new mode. This work was financially supported by the Klaus Tschira Foundation.

References

[1] Z. Yang, "Among-site rate variation and its impact on phylogenetic analyses," *Trends in Ecology & Evolution*, vol. 11, no. 9, pp. 367–372, Sep. 1996. [Online]. Available: https://doi.org/10.1016/0169-5347(96)10041-0

[2] P. Barbera, A. M. Kozlov, L. Czech, B. Morel, D. Darriba, T. Flouri, and A. Stamatakis, "EPA-ng: Massively Parallel Evolutionary Placement of Genetic Sequences," *Systematic Biology*, vol. 68, no. 2, pp. 365–369, 09 2018. [Online]. Available: https://doi.org/10.1093/sysbio/syy054

[3] M. Balaban, S. Sarmashghi, and S. Mirarab, "APPLES: Scalable distance-based phylogenetic placement with or without alignments," *Systematic Biology*, vol. 69, no. 3, pp. 566–578, Sep. 2019. [Online]. Available: https://doi.org/10.1093/sysbio/syz063

[4] M. Blanke and B. Morgenstern, "Phylogenetic placement of short reads without sequence alignment," Oct. 2020. [Online]. Available: https://doi.org/10.1101/2020.10.19.344986

[5] F. Izquierdo-Carrasco., J. Gagneur., and A. Stamatakis., "Trading running time for memory in phylogenetic likelihood computations," in *Proceedings of the International Conference on Bioinformatics Models, Methods and Algorithms - Volume 1: BIOINFORMATICS, (BIOSTEC 2012)*, INSTICC. SciTePress, 2012, pp. 86–95.

[6] J. Felsenstein, "Maximum Likelihood and Minimum-Steps Methods for Estimating Evolutionary Trees from Data on Discrete Characters," *Systematic Biology*, vol. 22, no. 3, pp. 240–249, 09 1973. [Online]. Available: https://doi.org/10.1093/sysbio/22.3.240

[7] F. Mahé, C. de Vargas, D. Bass, L. Czech, A. Stamatakis, E. Lara, D. Singer, J. Mayor, J. Bunge, S. Sernaker, T. Siemensmeyer, I. Trautmann, S. Romac, C. Berney, A. Kozlov, E. A. D. Mitchell, C. V. W. Seppey, E. Egge, G. Lentendu, R. Wirth, G. Trueba, and M. Dunthorn, "Parasites dominate hyperdiverse soil protist communities in Neotropical rainforests," *Nature Ecology & Evolution*, vol. 1, no. 4, p. 0091, 2017. [Online]. Available: https://doi.org/10.1038/s41559-017-0091

[8] A. Stamatakis, A. J. Aberer, C. Goll, S. A. Smith, S. A. Berger, and F. Izquierdo-Carrasco, "RAxML-light: a tool for computing terabyte phylogenies," *Bioinformatics*, vol. 28, no. 15, pp. 2064–2066, May 2012. [Online]. Available: https://doi.org/10.1093/bioinformatics/bts309

[9] B. Q. Minh, H. A. Schmidt, O. Chernomor, D. Schrempf, M. D. Woodhams, A. von Haeseler, and R. Lanfear, "IQ-TREE 2: New models and efficient methods for phylogenetic inference in the genomic era," *Molecular Biology and Evolution*, vol. 37, no. 5, pp. 1530–1534, Feb. 2020. [Online]. Available: https://doi.org/10.1093/molbev/msaa015

[10] S. A. Berger, D. Krompass, and A. Stamatakis, "Performance, accuracy, and web server for evolutionary placement of short sequence reads under maximum likelihood," *Systematic Biology*, vol. 60, no. 3, pp. 291–302, 2011.

[11] F. A. Matsen, R. B. Kodner, and V. E. Armbrust, "pplacer: linear time maximum-likelihood and Bayesian phylogenetic placement of sequences onto a fixed reference tree," *BioMed Central Bioinformatics*, vol. 11, no. 1, pp. 1–16, 2010.

[12] B. Linard, K. Swenson, and F. Pardi, "Rapid alignment-free phylogenetic identification of metagenomic sequences," *Bioinformatics*, vol. 35, no. 18, pp. 3303–3312, Jan. 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz068

[13] C.-A. Leimeister, S. Sohrabi-Jahromi, and B. Morgenstern, "Fast and accurate phylogeny reconstruction using filtered spaced-word matches," *Bioinformatics*, p. btw776, Jan. 2017. [Online]. Available: https://doi.org/10.1093/bioinformatics/btw776

[14] R. C. Edgar, J. Taylor, T. Altman, P. Barbera, D. Meleshko, V. Lin, D. Lohr, G. Novakovsky, B. Al-Shayeb, J. F. Banfield, A. Korobeynikov, R. Chikhi, and A. Babaian, "Petabase-scale sequence alignment catalyses viral discovery," *bioRxiv*, 2020. [Online]. Available: https://www.biorxiv.org/content/early/2020/08/10/2020.08.07.241729

[15] R. Leinonen, H. Sugawara, and M. S. and, "The sequence read archive," *Nucleic Acids Research*, vol. 39, no. Database, pp. D19–D21, Nov. 2010. [Online]. Available: https://doi.org/10.1093/nar/gkq1019

[16] G. M. Douglas, V. J. Maffei, J. R. Zaneveld, S. N. Yurgel, J. R. Brown, C. M. Taylor, C. Huttenhower, and M. G. I. Langille, "PICRUSt2 for prediction of metagenome functions," *Nature Biotechnology*, vol. 38, no. 6, pp. 685–688, Jun. 2020. [Online]. Available: https://doi.org/10.1038/s41587-020-0548-6

[17] S. N. Yurgel, J. T. Nearing, G. M. Douglas, and M. G. I. Langille, "Metagenomic functional shifts to plant induced environmental changes," *Frontiers in Microbiology*, vol. 10, Jul. 2019. [Online]. Available: https://doi.org/10.3389/fmicb.2019.01682

[18] B. Linard, N. Romashchenko, F. Pardi, and E. Rivals, "PEWO: a collection of workflows to benchmark phylogenetic placement," *Bioinformatics*, 07 2020, btaa657. [Online]. Available: https://doi.org/10.1093/bioinformatics/btaa657

[19] A. M. Kozlov, D. Darriba, T. Flouri, B. Morel, and A. Stamatakis, “RAxML-NG: A fast, scalable, and user-friendly tool for maximum likelihood phylogenetic inference,” *Bioinformatics*, 05 2019. [Online]. Available: https://doi.org/10.1093/bioinformatics/btz305

[20] T. Flouri, D. Darriba, A. Kozlov, M. T. Holder, B. Morel, and A. Stamatakis, “libpll - The Phylogenetic Likelihood Library,” Web page, accessed November 2020. [Online]. Available: https://github.com/xflouris/libpll-2