

Efficient phylogenetic tree inference for massive taxonomic datasets: harnessing the power of a server to analyze 1 million taxa

César Piñeiro^{1,*} and Juan C. Pichel^{2,*}

¹Information Retrieval Lab, CITIC, Universidade da Coruña, A Coruña 15008, Spain

²CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela 15782, Spain

*Correspondence address. César Piñeiro, Information Retrieval Lab, CITIC, Universidade da Coruña, A Coruña 15008, Spain. E-mail: cesaralfredo.pineiro@usc.es; Juan C. Pichel, CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela 15782, Spain. E-mail: juancarlos.pichel@usc.es

Abstract

Background: Phylogenies play a crucial role in biological research. Unfortunately, the search for the optimal phylogenetic tree incurs significant computational costs, and most of the existing state-of-the-art tools cannot deal with extremely large datasets in reasonable times.

Results: In this work, we introduce the new VeryFastTree code (version 4.0), which is able to construct a tree on 1 server using single-precision arithmetic from a massive 1 million alignment dataset in only 36 hours, which is 3 times and 3.2 times faster than its previous version and FastTree-2, respectively. This new version further boosts performance by parallelizing all tree traversal operations during the tree construction process, including subtree pruning and regrafting moves. Additionally, it introduces significant new features such as support for new and compressed file formats, enhanced compatibility across a broader range of operating systems, and the integration of disk computing functionality. The latter feature is particularly advantageous for users without access to high-end servers, as it allows them to manage very large datasets, albeit with an increase in computing time.

Conclusions: Experimental results establish VeryFastTree as the fastest tool in the state-of-the-art for maximum likelihood phylogeny estimation. It is publicly available at <https://github.com/citiususc/veryfasttree>. In addition, VeryFastTree is included as a package in Bioconda, MacPorts, and all Debian-based Linux distributions.

Keywords: phylogenetics, very large datasets, performance, parallelism

Introduction

Inferring evolutionary relationships or phylogenies is a formidable challenge in computational biology. The growth of datasets from next-generation sequencing has made large-scale phylogeny estimation crucial. However, the computational complexity of inferring phylogenies and performing multiple sequence alignment (MSA) presents a significant obstacle. Established methods like maximum parsimony (MP), maximum likelihood (ML), and Bayesian approaches are computationally intensive due to the NP-hard optimization problems they tackle [1]. As the number of taxa increases, these methods face a common hurdle: an exponential increase in the number of possible trees to explore.

The leading heuristics for ML tree estimation, such as RAxML [2] and IQ-TREE [3], employ diverse strategies to search for the tree that maximizes the likelihood score. Although they have made considerable performance improvements to handle larger datasets, RAxML, for example, was unable to reach convergence on a 10,000-sequence dataset even after a week [4]. Note that these tools are primarily optimized for datasets with a limited number of sequences but a significant number of sites (i.e., phylogenomics). Therefore, when working with datasets comprising a large number of sequences, users must opt for tools such as FastTree-2 [5] and VeryFastTree [6], which are very fast heuristics,

but they do not make very substantial attempts to optimize the likelihood score, or they may explore divide-and-conquer strategies [7–9]. In particular, our tool VeryFastTree was a big step forward in terms of performance, building a tree on a standard server from a large 330k alignment, 3.5 times faster than FastTree-2. However, there was still room to improve its speed, scalability, and memory consumption and also to add new functionalities.

In this work, we introduce the latest VeryFastTree code, version 4.0, showcasing its potential advantages and new features compared to both its previous version and FastTree-2. The earlier iteration of VeryFastTree achieved high performance by parallelizing the most time-consuming phase of constructing the tree, specifically the nearest-neighbor interchanges (NNIs), in comparison to FastTree-2. However, the new VeryFastTree further enhances performance by parallelizing all tree traversal operations, including, among others, the subtree pruning and regrafting (SPR) operations, which are especially relevant in terms of computing time when dealing with massive datasets. After a thorough experimental evaluation, the new version proves to be several times faster than the previous one and FastTree-2 on a variety of large datasets. At the same time, VeryFastTree-4 incorporates significant new features, including support for new and compressed file formats, improved compatibility with a wider range of operating systems, and the addition of *disk computing* functionality. The lat-

Received: October 23, 2023. Revised: April 17, 2024. Accepted: July 11, 2024

© The Author(s) 2024. Published by Oxford University Press GigaScience. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

ter is especially valuable for users who do not have access to high-end servers, as it allows them to process massive datasets, albeit with an increase in computing time.

New Features and Optimizations

VeryFastTree-4 (VFT4) is a big step forward with respect to our first version introduced in [6], from now on VFT3, designed to further accelerate the inference of phylogenies for massive alignments. Building upon the strengths of its predecessor, VFT4 introduces a host of innovative features and optimizations aimed at achieving even greater speed and efficiency. While the core principles used by VFT4 remain consistent with its previous versions, significant enhancements have been made to push the boundaries of speed. Below are some of the most noticeable improvements, optimizations, and features incorporated into VFT4.

Parallelization strategy

VFT3 achieved high performance by parallelizing the most time-consuming phase in the construction of the tree with respect to FastTree-2 (FT2) [5], the NNIs. Building upon this progress, VFT4 continues this approach by parallelizing all operations involving tree traversal, including the SPR moves, further enhancing performance. These computations are parallelized using 2 strategies: *tree partitioning*, which divides the tree into multiple nonoverlapping subtrees, and *parallel traverse*, a *parallel breadth-first traversal*.

Tree partitioning

Once an initial phylogenetic tree is constructed or passed as input argument, it is necessary to split the tree into disjoint subtrees in order to perform computations in parallel using different threads. Note that VFT4 uses rooted trees. The tree partitioning method is an algorithm that, given an initial tree, prunes nodes from the root to generate several independent subtrees, which are then assigned to different threads to work with them independently. Once all threads have finished their work, those nodes that were removed in the pruning process are subsequently processed using just 1 thread. In this way, all nodes in the tree are updated. In the ideal scenario, the computation of a node does not have dependencies on other nodes or these dependencies are contained within the subtree assigned to the same thread. However, this is not the case frequently, so dependencies must be controlled to prevent computation errors. For instance, NNIs and SPR operations perform topology-modifying actions on the tree. They require tree partitioning to ensure that node exchanges performed by different threads do not overlap. Tree partitioning must ensure that these operations can be executed in parallel within separate subtrees, allowing them to work independently without interfering with each other.

Before getting into the details, it is necessary to understand how the tree is traversed and how data dependencies between nodes are taken into account. We will use the tree of Fig. 1 as example. Each node in the tree is visited before its parents, which means that a depth-first postorder traversal is used [5]. Let's assume that the tree is split into 2 subtrees assigned to threads t_1 and t_2 , respectively. Since the root node of the tree is not assigned to a thread, it will be labeled as *not assigned*. Blue and green nodes are processed in parallel by threads t_1 and t_2 , while red nodes, which have dependencies on other threads, are processed sequentially after all subtrees have been processed. This ensures that all nodes in the tree are updated effectively. The dependencies of a node are determined by its relationships with its parents and sib-

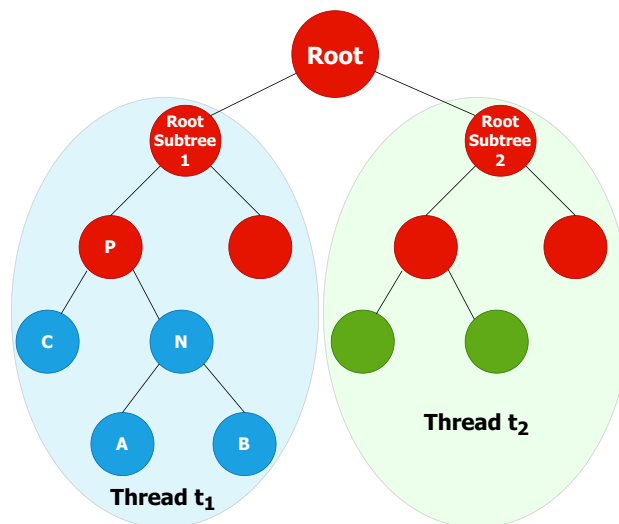


Figure 1: Example of tree partitioning and data dependencies with a penalty of 2. The graph represents only internal nodes (no leaves). Blue and green nodes are processed in parallel by threads t_1 and t_2 , respectively. Red nodes (labeled as *not assigned*) are processed sequentially after processing all subtrees.

lings. If these are not located within the same subtree, the node cannot be computed. This dependency can be quantified as the distance between a node and the farthest one required for computation. We refer to this distance as the *penalty* value. This means that nodes located at levels lower than the penalty value from the subtree's root cannot be processed in parallel, as they have dependencies on nodes outside the subtree. In the example of Fig. 1, the penalty is 2, so P cannot be computed since it depends on the root (its grandparent), which lies outside the subtree. On the other hand, C , N , and all their children have their grandparents within the subtree, allowing them to be computed without conflicts.

NNIs and SPR operations, which are the most time-consuming in tree construction, are parallelized using the tree partitioning strategy. In particular, NNIs perform a node exchange with either its parent or its uncle, followed by a recalculation of the weights of the involved nodes and their respective parents. This process incurs a penalty of 2, as the parent or uncle will be 1 level above the exchanged node, while the new parent will be located at the next level. In the case of SPRs, the penalty is dynamic and constrained by the parameter `maxSPRLength`, which limits the maximum distance a node can move during the regrafting process. Other operations such as computing SH-like supports, updating all branch lengths, and optimizing all branch lengths will also make use of tree partitioning.

VFT4 implements an advanced partitioning algorithm when compared to its predecessor, VFT3. This upgraded algorithm significantly improves both speed and adaptability across various scenarios. Unlike the previous version, which limited partitioning to NNIs, the new algorithm introduces a more versatile approach, allowing its application to other operations, such as those previously mentioned. This new algorithm considers the minimization of the penalty parameter as an evaluation criterion. Subsequently, the resulting subtrees generated by this process, also referred to as *solutions*, are assigned to threads with the aim of achieving a good load balance.

Partitioning method

Our partitioning method is guided by an objective function, whose goal is 2-fold. First, it should balance the workload assigned to each thread (i.e., the number of internal nodes to be processed by each thread). To model the workload, we define the *weight* of a node as the number of internal nodes beneath it plus itself. This way, the *workload* associated to process a subtree is the weight of its root node. The second goal of the objective function should be to reduce to the minimum the amount of sequential work, which corresponds to processing the *not assigned* nodes. Note that according to the Amdahl's law [10], a small percentage of sequential work harms noticeably the performance and scalability of a parallel application.

Let T be the input tree to be partitioned, which only contains internal nodes. A partition $P(T)$ is a set of disjoint binary subtrees s_i included in T . We must highlight that not all nodes in the tree should be included in a partition. Let $|P(T)|$ denote the number of subtrees in the partition and n the number of threads. Each subtree $s_i \in P(T)$ is assigned to a particular thread in such a way that $P_j(T)$ contains all subtrees in the partition $P(T)$ assigned to thread j . This way, we can calculate the workload of the subtrees assigned to a particular thread j as

$$\text{workload}(P_j(T)) = \left\{ \sum \text{workload}(s_i) \mid s_i \in P_j(T) \right\} \quad (1)$$

We evaluate a partition $P(T)$ using the following objective function:

$$\frac{\text{sequential_workload}}{\max\{\text{workload}(P_j(T)) \mid 1 \leq j \leq n\}} \quad (2)$$

Note that *sequential_workload* refers to the workload of T when it is processed sequentially. The goal of our partitioning method will be to find a partition $P(T)$ that maximizes the value of the objective function. The pseudocode of the algorithm to find the best tree partitioning is detailed in Algorithm 1. The objective function aligns with the concept of *Speedup*, defined as the ratio of the execution time of a task when using multiple threads compared to the execution time when using a single thread. In this way, a value of 1 indicates equivalent speed to sequential processing, while a value of n signifies optimal workload distribution among n threads.

The tree partitioning process in VFT4 includes an initialization phase where the weight values for each node in the tree are computed. As previously mentioned, these weights represent the number of descendant nodes for each node, adjusted according to the penalty value. Then, the initial solution is constructed using the direct child nodes of the tree's root. In other words, the initial solution has as many subtrees as the root node has children. Nodes in the solution are always sorted by weight for efficiency. Next, an iteration process begins. During each iteration, the current solution is evaluated using Equation 2. To achieve this, it is necessary to assign subtrees to the different threads using a heuristic method, which will be explained later. Once the solution is evaluated, a new one is created in such a way that the subtree root node with the highest weight is replaced by its child nodes. The evaluation results of the last W iterations are stored to use them as stopping criterion. We refer to W as the *tendency window size*. Note that the algorithm continues as long as the current solution has nodes to split, there are threads without at least 1 subtree assigned, or the stopping criterion is not reached.

function TreePartition

Input :tree, threads_num, penalty, tendency_size
Output :best_solution

```
// Initialization: initial solution and weights computation
weights ← ComputeWeights (tree, penalty)
solution ← Children (RootNode (tree))
best_solution ← solution
solution_speedup ← Evaluate (solution)
best_solution_speedup ← solution_speedup
tendency ← []
// Iterations to find the best solution
while Len (solution) > 0 ∧ (Len (solution) < threads_num ∨
!StoppingCriterion (tendency, tendency_size)) do
  node ← GetAndRemoveNodeWithMaxWeight (solution)
  if node has children then
    Insert (Children (node)) into solution
    solution_speedup ← Evaluate (solution)
    Insert (solution_speedup) into tendency
  if solution_speedup > best_solution_speedup then
    best_solution ← solution
    best_solution_speedup ← solution_speedup
return best_solution
```

function StoppingCriterion

Input :tendency, tendency_size
Output :must_stop

```
if Len(tendency) = tendency_size then
  balance ← 0
  for i ← 1 to tendency_size do
    for j ← i to tendency_size do
      if tendencyi <= tendencyj then
        balance ← balance + 1
      else
        balance ← balance - 1
  RemoveFirst (tendency)
  return balance < 0
return false
```

function ComputeWeights

Output :Weight of each node in the tree taking into account the penalty

function Children

Output :Childrens of the node

function GetAndRemoveNodeWithMaxWeight

Output :Gets and removes the node with the highest weight

function Evaluate

Output :Quality of the solution calculated considering Equation 2

Assignment of subtrees to threads

The evaluation of the quality of a solution requires mapping the subtrees of a partition to the considered threads in such a way that the workload is balanced among them. This is a challenge itself and resembles the *k-partitioning* problem [11]. The *k-partitioning* problem is defined as follows: given a set of items $\{I_1, I_2, \dots, I_n\}$ where item I_j is of weight $w_j > 0$, find a partition S_1, S_2, \dots, S_m of this set with $|S_i| = k$ such that the maximum weight of all subsets S_i is minimal. Therefore, considering our case, items are subtrees and k is the number of threads. The *k-partitioning* problem is NP-hard, so algorithms/heuristics to approximate the solution should be used.

Since our procedure requires solving the *k-partitioning* problem every time a new partition is evaluated, we have opted for a simple greedy approximation based on the first-fit decreasing algorithm whose computational cost is low. The method starts sorting the subtrees in descending order according to their weight. Then, the k first subtrees are distributed among the threads. The remaining subtrees are iterated and assigned to the thread that currently has the minimum workload assigned, that is, the lowest value of $\text{workload}(P_j(T))$.

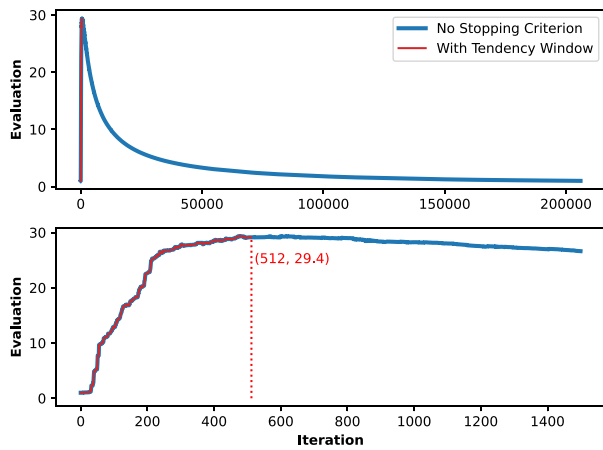


Figure 2: Comparison of iterations required for achieving near-optimal tree partitioning using our method versus evaluating all possible tree partitionings without a stopping criterion. The bottom figure provides a zoomed-in view of the top one. Experiments were conducted with 30 threads and a dataset consisting of 274,000 unique sequences.

Stopping criterion

The stopping criterion in the tree partitioning process determines when the algorithm should terminate its execution. This criterion is required because completing all iterations, until no more nodes are left to split, can sometimes result in a higher computational cost than working on the tree without partitions. Therefore, the stopping criterion ensures that the algorithm halts its execution when a near-optimal solution has been found. To verify if the stopping criterion is met, we assess the quality of the last W partitions using Equation 2 to determine whether the overall trend in recent iterations indicates an improvement or deterioration in the solutions (see the `StoppingCriterion` function in Algorithm 1). In order to achieve this, a counter compares each pair of elements within the tendency list, incrementing it if the earlier element is less than or equal to the later element and decrementing it otherwise. Consequently, the function returns true if the balance is negative, indicating a worsening trend, and false otherwise, signaling that the stopping criterion has not yet been met due to improving solutions. By default, W is set to 50, but users can modify this value through the interface parameter (see the `Commands Interface` section for details).

To demonstrate the advantages of our proposal, Fig. 2 illustrates the comparison between the number of iterations required by our method to achieve a near-optimal tree partitioning and the scenario where all possible tree partitionings are evaluated without any stopping criterion. These results were obtained using 30 threads and the *Large* dataset that contains 274,000 unique sequences (refer to Table 1 for dataset details). Since 30 threads were considered, according to Equation 2, the ideal (maximum) quality value of a partitioning is 30. Our method only requires 512 iterations to reach a value of 29.4, while without the stopping criterion, it is necessary to evaluate more than 200,000 partitionings (iterations).

Parallel breadth-first traversal

Tree partitioning enables parallel processing of a single tree by assigning disjoint subtrees to each thread, allowing them to compute independently and without interference. However, when an operation has no dependencies and solely modifies the node being computed, creating subtrees is inefficient, as each node can be directly assigned to a thread. So, starting from the deepest nodes of

the tree, each thread handles a portion of the nodes and computes them simultaneously, following a breadth-first traversal pattern, which involves visiting all nodes at a given depth level before moving on to the next level. Once finished with that level, the upper levels are distributed until the root of the tree is reached. This approach is specifically utilized in the following operations: computing initial profiles, recomputing profiles, recomputing ML profiles, computing tree length, and computing the likelihood for each site. It is worth noting that FT2 and VFT4 store profiles for the internal nodes in the tree instead of storing a distance matrix, which requires far less memory. Each profile includes a frequency vector for each position and the weighted average of its children's profiles. These profiles are used to compute the distances between internal nodes in the tree and also the total distance from a node to all other nodes (see [5, 12] for a detailed description).

New thread levels

To provide users control over parallelism and adaptability to various usage scenarios, VFT4 introduces 5 distinct thread levels. These levels, including the original levels from VFT3 (now referred to as levels 0, 1, and 2), allow users to finely adjust the degree of parallelization based on their specific requirements.

At level 0, VFT4 employs the same parallelization strategy as FT2, but with the addition of new parallel blocks to enhance performance. Level 1 introduces parallel blocks that require additional memory for computations, enabling more efficient processing. Level 2 utilizes the tree partitioning method to accelerate ML NNI rounds. Level 3 performs in parallel all the computations but SPRs. Last, level 4 leverages the tree partitioning method to expedite also SPR steps, but it only pays off with datasets containing a high number of alignments. For this reason, level 3 is the default option.

Each level in VFT4 is incremental with respect to the previous ones. However, it is important to note that computation at level 2 and above follows a different tree traverse order, which may result in different trees with respect to the sequential execution. Nevertheless, these results remain strictly correct. By incorporating this multilevel approach, VFT4 offers users the flexibility to optimize their parallelization strategy according to factors such as dataset size, performance needs, and desired trade-offs.

Disk computing

As previously mentioned, phylogenetic tree inference requires a significant amount of memory, which can become problematic when dealing with numerous sequences and limited resources (e.g., when considering a low-end server with reduced available RAM memory). To tackle this challenge, VFT4 introduces *disk computing*, a technique that utilizes the hard drive to supplement the memory requirements. While disk computing aids in handling large datasets, it does impact performance due to the slower access speed of hard disks compared to RAM. However, the advantages of being able to process larger datasets outweigh the performance trade-off.

We classified the dynamically allocated memory used in the tree construction process into 3 types: fixed memory, variable memory, and computation memory. In particular:

- *Fixed memory* is allocated at the beginning of the program execution once the properties of the input dataset (such as the number of sequences, length, etc.) are established and remains reserved until the program's completion. For instance, fixed memory is used to store sequences and weights in each profile.

Table 1: Characteristics of the datasets used in the experimental evaluation. Information obtained using *BigSeqKit* [15]

Dataset label	File name	Format	Type	Sequences	Unique sequences	Length
Large	<i>sel03n.masked</i>	FASTA	AA	331,550	274,401	1,287
Very Large	<i>gg_12_10_aligned</i>	FASTA	DNA	1,075,170	858,234	7,682
Ultra-Large	<i>1-million-taxon-run1</i>	NEXUS	DNA	1,000,000	989,109	21,946

- *Variable memory*, on the other hand, is allocated to store values during specific stages of the computation. For instance, it is the memory used to store the frequency vector (conditional probabilities) whose size changes dynamically and cannot be precisely predicted. Note that due to significant variations in data sizes, overallocation of variable memory is often impractical.
- *Computation memory* refers to private memory required by threads to perform their tasks. This includes stack memory, which handles function calls; local variables; and temporary data while the program is running. Additionally, computation memory includes some temporary variables, which store intermediate results and facilitate complex calculations, as well as data structures, which organize and manage data for efficient processing. For example, computation memory is used to temporarily store the new computed conditional probabilities for possible exchanges within an NNI iteration. It is important to emphasize that once the temporary values are utilized, they are subsequently released.

While fixed and variable memory can be offloaded to the disk, computation memory must be always maintained in RAM. This is because the performance of the computation process is critical. In addition, even for huge datasets, computation memory remains significantly smaller in comparison to fixed and variable memories.

On the other hand, using swap space as an alternative when a process exceeds RAM capacity has important limitations and drawbacks. First, the swap partition has a fixed size, potentially leading to out-of-memory errors if exceeded. This is especially relevant when dealing with very large datasets, as is our case. Therefore, if the dataset exceeds the size of the swap, only an administrator would have the authority to expand the swap partition to prevent the job from being aborted. In addition, since all programs/applications share the same swap space, excessive swapping by 1 program can adversely affect others, leading to contention for disk input/output (I/O) resources and reduced responsiveness. In contrast, the mechanism used by VFT4 in disk computing is based on memory-mapped files, which offers better I/O performance due to the optimized kernel-level caching mechanisms and efficient block-based access. Note that when using memory mapping, any available disk can be used to store data, avoiding the swap partition size limitation. Additionally, memory-mapped files are also controlled by the operating system, similar to swap space, and are not manually managed. This ensures efficient handling of data movement between memory and the disk, with the operating system making decisions based on system resources and requirements.

VFT4 introduces the use of disk computing with 2 parameters:

- `-disk-computing`: it facilitates the transfer of fixed memory to disk.
- `-disk-dynamic-computing`: it facilitates the transfer of variable memory to disk.

Employing both parameters allows VFT4 to move data from memory to disk in such a way that very large datasets can be processed even on low-end servers with limited RAM. However, it is essential to consider that this feature may come at the cost of decreased computational performance. In this way, the performance of VFT4 is directly influenced by the amount of memory sent to the disk. Hence, it is advisable, whenever possible, to compute by transferring only fixed memory to the disk. This approach strikes a delicate balance between memory consumption and computational efficiency, optimizing the overall performance of the application.

Optimized memory consumption

VFT3 has been implemented using C++, which introduces additional memory overhead compared to the C implementation of FT2. Moreover, its more efficient utilization of threads leads to an increase in memory consumption as the number of threads grows. This increase is primarily caused by the replication of data structures, synchronization mechanisms, stack space for each thread, and caching effects, among other factors.

VFT4, on the other hand, has been redesigned to minimize the use of objects in memory-intensive sections and to release memory as soon as it is no longer needed. Consequently, in sequential execution, the memory requirements of VFT4 are even lower than those of FT2.

Additionally, to reduce the overhead introduced by each thread, VFT4 optimizes the storage of common temporary data, ensuring that it is stored only once. This significantly minimizes the overhead associated with multithreading, resulting in more efficient parallel execution compared to VFT3.

Other optimizations and functionalities

- *Support for new and compressed formats*: VFT3 and FT2 are limited to supporting the FASTA and Phylip formats, both stored as plain text. In contrast, VFT4 has extended its support to include the widely used Nexus and FASTQ formats. The Nexus format allows for storing sequences and the initial tree within a single file. Additionally, it is common for datasets downloaded from internet repositories to be compressed in formats such as `.gz` or `.bz`. Previously, it was required to manually decompress these files before using them. However, VFT4 can directly read compressed sequences in compatible formats, thanks to the integration of the Zlib and libBZ2 libraries. This approach saves time by removing the need for manual file decompression, resulting in improved performance and avoiding the additional effort of reading larger uncompressed files from the disk.
- *Broader compatibility*: VFT4 is a versatile tool that supports Linux, Windows, and macOS, including Windows executables. It is also conveniently available in the Bioconda package repository [13], making it easily accessible to the bioinformatics community. Furthermore, for macOS users, it is also available as a MacPorts package [14]. Finally, for Linux users, VFT4 is included as a package in all Debian Linux distribu-

tions, simplifying its installation and integration into various computing environments.

- **Better compilation support:** The compilation process has been enhanced by adding support for new compilers, such as *clang*, and incorporating new features like AVX512 in Windows builds. The code has been optimized to comply with the latest compiler standards. Furthermore, parallel compilation has been implemented to accelerate the overall compilation time.

Performance Evaluation

Next, we present experimental results that clearly demonstrate the superior performance of our tool, VFT4, compared to its predecessors, VFT3 and FT2. We have evaluated the following aspects: running time, memory consumption, CPU usage, and topological accuracy.

For the evaluation, we selected 3 very large datasets with varying number of taxa and alignment lengths. Please refer to Table 1 for specific details about these datasets. It is important to note that only unique sequences were considered when constructing the trees. To ensure test reproducibility, we have included information about the parameters used to build the trees for each dataset:

- *Large*: `-spr 4 -gamma`
- *Very Large*: `-nt -gamma -gtr`
- *Ultra-Large*: `-nt -gamma -gtr`

The arguments have the following meanings:

- `-nt`: nucleotide alignment used as input.
- `-spr`: sets the number of rounds of SPR moves (default value is 2).
- `-gtr`: utilizes the generalized time-reversible (GTR) model of nucleotide substitution.
- `-gamma`: after optimizing the tree with a fixed rate for each site (the CAT model), VFT (and FT2) will rescale the tree to optimize the Gamma20 likelihood [5].

Experiments were conducted using 1 server with two 32-core Intel Xeon Ice Lake 8352Y @2.2 GHz processors and 512 GB of RAM. This server is part of a cluster installed at CESGA (Galicia Supercomputing Center, Spain) [16] running Rocky Linux v8.4 (kernel v4.18.0). We have used in the performance comparison the following tools and versions: Fast-Tree v2.1.11 [17], VeryFastTree v3.0 (our previous version), and VeryFastTree v4.0.3.

Running times

First, we show in Fig. 3 the running times when building the trees using single precision. We must highlight that, even if single precision is selected in VFT4 and FT2, many intermediate calculations such as vector reductions are performed with double precision to minimize errors. In the case of the *Large* dataset, VFT4 outperforms FT2 and VFT3, achieving speed improvements of 2.6 times and 1.6 times respectively, resulting in an execution time of just 3.5 hours. When inferring the phylogenetic tree from the *Very Large* dataset, VFT4 completes the task in 17.2 hours, whereas FT2 requires 57.7 hours and VFT3 takes 35.4 hours. Finally, dealing with the *Ultra-Large* dataset, VFT4 is able to build the tree in 35.8 hours (i.e., 1.5 days). Note that the time required by FT2 and VFT3 increases noticeably to 4.8 and 4.5 days, respectively. In other words, VFT4 is 3.2 times and 3 times faster than FT2 and VFT3.

On the other hand, Fig. 4 presents the running times for tree construction but using double-precision arithmetic. In this case, for the *Large* dataset, VFT4 is 6.9 times and 2.4 times faster than FT2 and VFT3, respectively, reducing the execution time to less than 3 hours. Note that VFT4 builds the tree faster when using double precision than considering single precision (2.8 versus 3.5 hours). This is caused by the good behavior of the vectorization strategies used by VFT4 when considering protein alignments. On the other hand, inferring the phylogenetic tree from the *Very Large* dataset using VFT4 takes 18 hours, while FT2 requires 61.7 hours and VFT3 41.3 hours. As a result, VFT4 is again the fastest tool. In conclusion, when dealing with the *Ultra-Large* dataset, VFT4 can complete the tree-building process within 41.2 hours, or 1.7 days. Notably, the time required by FT2 and VFT3 increases significantly to 5.3 and 4.6 days, respectively. To put it simply, VFT4 outperforms FT2 and VFT3 in speed by 3.1 times and 2.7 times, respectively.

As mentioned in the Introduction, in addition to FT2 and VFT, there are other state-of-the-art tools for ML tree estimation, with RAxML and IQ-TREE being the most commonly used by the scientific community. However, they are limited in their ability to process datasets containing a large number of sequences due to their extensive running times. To validate the observations made in previous works [4], we assessed both tools, RAxML-NG v.1.2.0 and IQ-TREE v.2.1.3, using our *Large* dataset. Since RAxML-NG can be executed on a cluster, we conducted experiments with this tool using 4 computing nodes instead of just 1 server. We manually constrained the experiments to a maximum running time of 1 week. Both tools were unable to construct the tree within that time frame. It is worth noting that VFT4 can estimate the tree for the *Large* dataset in approximately 3 hours (see Figs. 3 and 4). This confirms that both RAxML and IQ-TREE are not well suited for datasets containing a very large number of sequences.

Therefore, to the best of our knowledge, these results establish VFT4 as the fastest tool in the state-of-the-art for ML phylogeny estimation. But moreover, VFT4 allows the processing of massive datasets that would otherwise be intractable or would require excessively high computing times.

Memory consumption and CPU usage

In the previous section, we pointed out how the more effective use of threads results in higher memory consumption as the number of threads increases. VFT4 addresses this issue by optimizing the storage of commonly used temporary data, ensuring it is stored only once. As a result, the overhead associated with multithreading is significantly reduced, leading to a more efficient parallel execution compared to VFT3.

Figure 5 displays the CPU usage and memory consumption for all the considered tools when building the tree from the *Ultra-Large* dataset using single-precision arithmetic. The purpose of this illustration is to showcase the memory optimizations implemented in VFT4. Each time step in the graphs corresponds to 3,600 seconds. It can be observed that the maximum memory consumed by VFT4 is a bit higher with respect to FT2, 272 GB and 228 GB, respectively. This increase is attributed to a more efficient utilization of threads. It is important to note that VFT4 extensively utilizes parallelism, employing the maximum available number of threads (64) for most of the time. In contrast, FT2 is constrained to using only a few threads, as indicated by the blue lines in the figures. As a consequence, VFT4 requires some extra memory for multithreading, but it comes with the benefit of a remarkable decrease in the running time. On the other hand, VFT4 significantly reduces

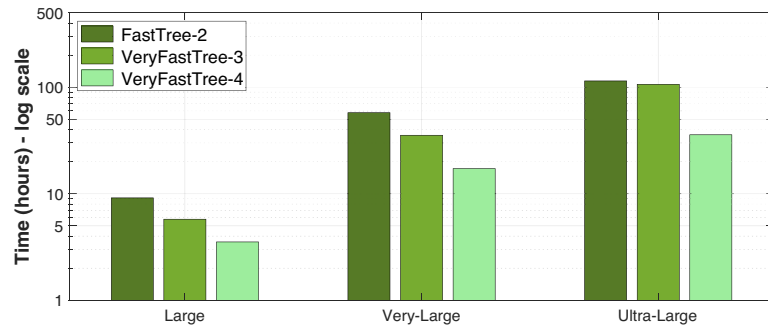


Figure 3: Running times of FT₂, VFT₃, and VFT₄ for building the trees using single-precision arithmetic and different datasets.

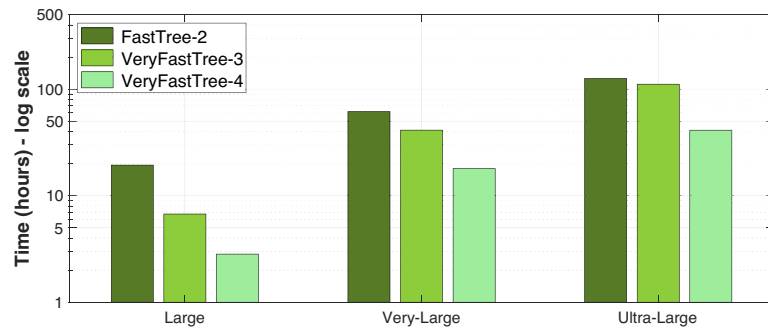


Figure 4: Running times of FT₂, VFT₃, and VFT₄ for building the trees using double-precision arithmetic and different datasets.

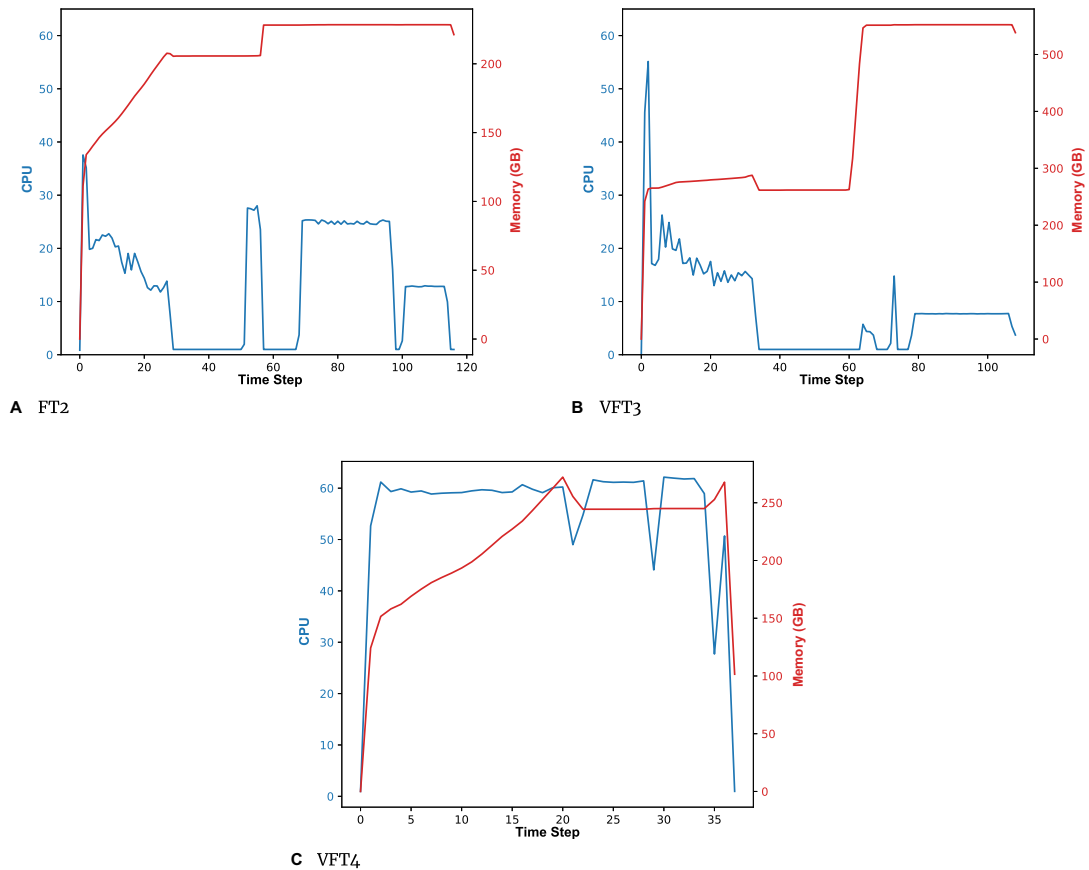


Figure 5: CPU usage (# of cores) and memory consumption when building the single-precision trees using as input the Ultra-Large dataset. Time step = 3,600 seconds.

memory consumption compared to VFT3. In particular, VFT3 requires twice the maximum memory used by VFT4. Furthermore, it also demonstrates greater efficiency in terms of parallelism.

As we pointed out previously, VFT4, like FT2, uses sequence profiles of internal nodes of the tree to implement neighbor joining instead of storing a distance matrix. Typically, as stated in [12], a distance matrix for an alignment with N sequences, L sites, and a different characters in the alphabet requires $O(N^2)$ space. However, the upper bound of memory space required by VFT and FT2 is $O(NLa)$. Each profile comprises a frequency vector for each position and the weighted average of its children's profiles.

If we take a look in detail, VFT4 requires 272 GB of memory to process the *Ultra-Large* dataset (see Fig. 5C). Its corresponding parameters are $N = 989$, $L = 21,946$, and $a = 4$ (nucleotides). As a consequence, taking into account that the resulting tree contains $2N$ nodes and single precision uses 4 bytes, the memory required by the profiles is the sum of the weights and frequencies for each node in the tree:

- Weights require exactly $2N \times L \times 4$ bytes = $(2 \times 989, 109) \times 21,946 \times 4$ bytes = 173.65 GB.
- The memory of frequencies can be estimated as $2N \times a \times g \times 4$ bytes, with “g” being the number of gaps in the sequence. Since “g” is not a constant value and changes in each node as the topology of the tree evolves, an exact value for the memory space used cannot be provided. In any case, most of the difference between the total memory used by VFT4 (272 GB) and the memory required only by the weights of the profiles (173.65 GB) is due to the frequencies.

Disk computing

In many cases, researchers are faced with limited computing resources, often having access only to low-end servers that possess a limited amount of memory. To deal with this issue, as was commented previously, we introduced *disk computing* in VFT4, a new feature that allows to offload static and dynamic memory to the disk with the aim of handling very large datasets even on small servers. The obvious drawback is an increase in the running times.

An example of the effects of disk computing can be found in Fig. 6. Each time step in the graphs corresponds to 300 seconds. In particular, Figs. 6A and 6B show the CPU usage and the memory footprint in a normal execution of VFT4 when processing the *Large* dataset considering double-precision arithmetic and 1 and 64 threads, respectively. In these cases, the maximum memory consumed was 58.5 GB (1 thread) and 84.42 GB (64 threads). Note that the spike in the memory consumption is caused by the computation and storage of the profiles at each node of the tree. On the other hand, Figs. 6C and 6D also display the CPU usage and the memory footprint but using disk computing with 1 and 64 threads, as well as limiting the memory of the server to just 16 GB. It can be observed that thanks to this new feature, we can successfully build the tree on a small server. Considering 64 threads, for example, the maximum memory used is approximately 5.3 times less compared to a normal execution. It may seem apparent, but without disk computing, the processing of datasets that surpass the available memory would be impossible. It would result in an out of memory error, which would be the case of using both FT2 and VFT3.

On the other hand, the VFT4 running time increases, using 1 thread from 16.8 to 27.5 hours while using 64 threads from 2.8 to 53.4 hours. In other words, using disk computing on a 16 GB server is from 1.6 times (1 thread) to 19 times (64 threads) slower than the standard execution. An interesting observation is that the se-

quential execution with disk computing is faster than the corresponding execution with 64 threads. This is due to the limitations of disk and I/O bandwidth, which struggle to handle the extremely high number of requests generated when using 64 threads, leading to contention on the bus. Therefore, we recommend reducing the number of threads when utilizing the disk computing feature.

Topological accuracy

In the literature, several studies have examined the performance of leading ML tools, such as RAxML, IQ-TREE, and FT2, regarding accuracy and computational efficiency with large datasets. Liu et al. [18] compared RAxML and FT2 using simulated datasets for single genes containing 1,000 or more sequences across various estimated alignments. They concluded that both methods exhibited similar topological accuracy. Lees et al. [19], in their exploration of ML heuristics through simulations, found RAxML and IQ-TREE to be comparable in accuracy, both outperforming FT2. Moreover, 2 studies [20, 21] evaluated FT2 and RAxML on datasets containing fragmentary sequences. They observed that FT2 performed less accurately than RAxML when handling alignments with a high proportion of fragmentary sequences. Overall, these studies have shown that IQ-TREE and RAxML are both very good ML heuristics with respect to ML scores, while all of the studies have shown that FT2 is indeed very fast, but it is so good at ML score optimization.

On the other hand, as we explained in our previous study [6], VFT is a highly tuned implementation of FT2 that keeps the phases, methods, and heuristics used by FT2 to estimate the phylogenetic tree. For this reason, it is only necessary to compare the topological accuracies with respect to FT2 (and our previous version of VFT), since the works commented above already performed a thorough comparison between RAxML, IQ-TREE, and FT2.

Similar to [5], we defined the topological accuracy as the proportion of splits in the true trees that are successfully recovered by each respective tool. This metric is the inverse of the topological Robinson–Foulds distance [22], normalized to a range between 0 and 1.

Our findings demonstrate that both VFT3 and VFT4 exhibit determinism while maintaining the same level of accuracy as FT2. To validate this, we assessed topological accuracy using 5,000-sequences simulated protein alignments [17], which were also employed in the original FT2 study for the same purpose. Therefore, the true phylogeny is available. All simulations are based on trees from profile alignments of biological sequences and include variable rates across sites. The simulations are described in more detail in [23]. Trees were computed using 64 threads, double-precision arithmetic, and `-gamma -spr 4` parameters. All tools used the same seed for initializing the random number generator. The experimentation script, `treecmp.py`, is accessible in our repository. We show results for VFT4 considering 2 different levels of parallelism (i.e., *thread levels*). As we explained previously, level 3 performs in parallel all the computations but SPRs, while level 4 leverages the tree partitioning method to accelerate also SPR movements. For VFT3 and VFT4, topological accuracy values were computed by averaging 7 measurements (1 for each alignment in the dataset). Conversely, parallel executions of FT2 did not yield deterministic results. Consequently, we present a range of accuracy values. This range was derived by averaging the minimum and maximum values obtained from 10 executions for each alignment. Accuracy results expressed in percentages are shown in Table 2.

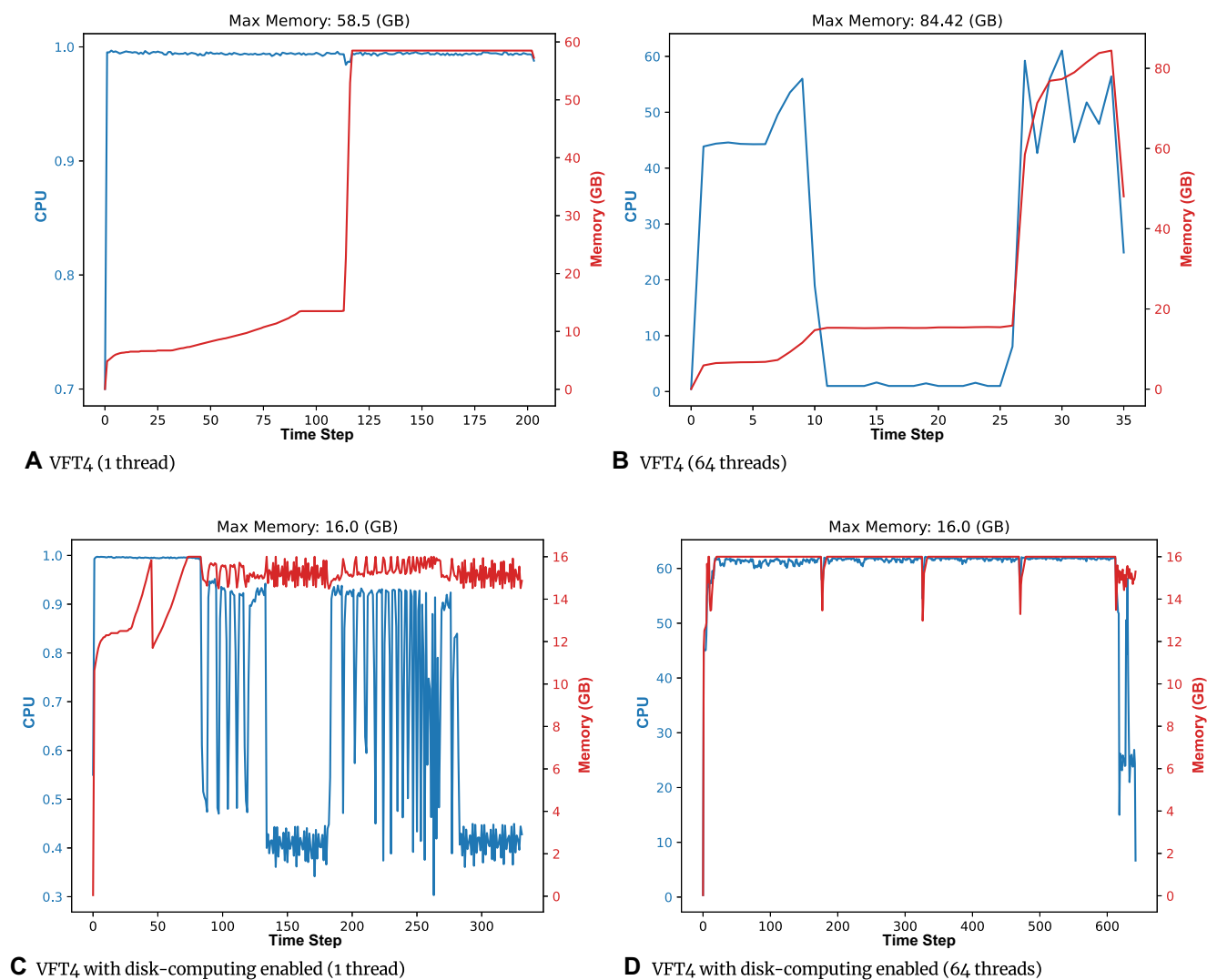


Figure 6: Effects of using disk computing when building the double-precision trees using as input the *Large* dataset. Time step = 300 seconds.

Table 2: Topological accuracies (in %) obtained by FT₂, VFT₃ and VFT₄ using 5,000-sequences simulated protein alignments. Note that the parallel version of FT₂ is nondeterministic, so minimum and maximum values are displayed between brackets

Tool	COG438	COG583	COG596	COG642	COG1028	COG1309	COG2814	Average
VFT4 (level 3)	85.35	79.11	88.14	82.66	86.19	86.55	79.59	83.94
VFT4 (level 4)	85.43	79.13	87.72	82.19	86.37	87.29	79.92	84.01
VFT3	85.51	79.57	87.82	81.89	86.29	87.09	80.18	84.05
FT2	[85, 85.83]	[79.21, 79.91]	[88.04, 88.78]	[82.11, 82.95]	[86.30, 87.03]	[86.19, 87.59]	[79.06, 80.66]	[83.70, 84.68]

Based on the results obtained, we conclude that VFT4 produces trees with an accuracy level within the same range as FT₂. Additionally, minor differences were observed when using different thread levels. In particular, the most aggressive one in terms of parallelism, level 4, shows a slightly better behavior than level 3.

Commands Interface

The new VeryFastTree version 4.0 was designed with extensive cross-platform compatibility, offering support for a variety of operating systems, including Linux, Windows, and macOS. This

broad compatibility ensures that users can access and utilize the tool seamlessly across different computing environments. Windows users, in particular, benefit from the availability of executable versions of VFT4, simplifying the installation process and widening its user base. Additionally, for those in the bioinformatics field, VFT4 can be effortlessly located within the Bioconda package repository, streamlining both installation and integration into bioinformatics workflows. VFT4 is also available for macOS users as a MacPorts package. Finally, Linux users will appreciate that VFT4 is included as a package in all Debian Linux distributions, making installation straightforward and facilitating its integration into diverse computing setups.

Just like the previous version of VeryFastTree, it implements the same command interface as FT2. This means that the arguments behave exactly the same as in FT2. To check all these arguments, the “-h” or “-expert” option can be used. Consequently, to benefit from the performance advantages provided by VFT4, it is only necessary to replace the call to FT2 with a call to VFT4, using the same options.

On the other hand, VFT4 has its own extra arguments, which have been grouped in the *Optimizations* section. These arguments are related to the parametrization of the different parallelization, vectorization, and optimization strategies included in VFT4. Next we list and explain the new arguments available:

- **-threads [n]**
It allows specifying the number of threads (*n*) used in the parallel execution. If this option is not set, the corresponding value will be obtained from the environment variable `OMP_NUM_THREADS`. This is the same approach followed by FT2. If *n* = 1, VeryFastTree behaves in the same way as FT2 compiled without the `-DOPENMP` flag.
- **-threads-level [level]**
It allows changing the degree of parallelization.
 - If level is 0, VeryFastTree uses the same parallelization strategy as FT2 with some new parallel blocks.
 - If level is 1, VeryFastTree uses parallel blocks that require additional memory for computation.
 - If level is 2, VeryFastTree accelerates the rounds of ML NNIs using its tree partitioning method.
 - If level is 3 (default), VeryFastTree performs more computations without preserving sequential order.
 - If level is 4, VeryFastTree also accelerates the rounds of SPR steps using its tree partitioning method (it can only be used with datasets larger than $2^{\max\text{SPRlength}+2}$).

Note: Each level includes the previous ones, and computation at level 2 and above is performed in a different tree traverse order, so the result may change.
- **-threads-mode [mode]**
Changes the mode of parallelization.
 - If mode is 0, VeryFastTree uses nondeterministic parts, some inspired by FT2 but improved.
 - If mode is 1, VeryFastTree only uses deterministic parallelization.

Since version 4.0, deterministic algorithms are at least faster than nondeterministic ones, making deterministic the preferred choice.
- **-threads-ptw [n]** (Partitioning Tendency Window)
It sets the size of the partitioning tendency window used by the tree partitioning algorithm to determine when to stop searching. The window stores the last solutions and checks if a better solution can be found. Increasing the value allows the algorithm to explore the tree deeper and potentially find better solutions. The default value is 50.
- **-threads-verbose**
It shows subtrees assigned to the threads and theoretical speedup, only with `verbose > 0`.
- **-double-precision**
It uses double-precision arithmetic. Therefore, it is equivalent to compile FT2 with `-DUSE_DOUBLE`.
- **-ext [type]**
It enables the vector extensions:

- **AUTO:** (default) selects AVX2 when `-double-precision` is used and SSE3 otherwise. If 1 extension is not available, the previous level is used.
- **NONE:** Operations are performed with the native programming language operators. In addition, loops are unrolled with the aim of providing hints to the compiler for applying some optimization (including vectorization).
- **SSE3:** Arithmetic operations are performed using SSE3 vector intrinsics. Each instruction operates on 128-bit registers, which could contain four 32-bit floats or two 64-bit doubles.
- **AVX:** Arithmetic operations are performed using AVX vector intrinsics. Each instruction operates on 256-bit registers, which could contain eight 32-bit floats or four 64-bit doubles.
- **AVX2:** Similar to AVX, but some arithmetic operations are performed using additional AVX2 vector intrinsics not included in the AVX instruction set. Each instruction operates on 256-bit registers, which could contain eight 32-bit floats or four 64-bit doubles.
- **AVX512:** Arithmetic operations are performed using AVX512 vector intrinsics. Each instruction operates on 512-bit registers, which could contain sixteen 32-bit floats or eight 64-bit doubles.
- **-disk-computing**
If there is not enough available RAM to perform the computation, disk will be used to store extra data when it was not needed. Using disk to perform the computation will substantially increase the execution time.
- **-disk-computing-path [path]**
Like `-disk-computing` but using a custom path folder to store data.
- **-disk-dynamic-computing**
By default, disk computing only creates files associated with static data in RAM, which means that there is no significant impact on performance as long as there is available RAM. This option further reduces memory usage by storing dynamic data on disk. However, even if there is enough RAM, it will have a negative impact on performance due to the creation and deletion of files.
- **-seed [seed]**
Set the seed for the random number generator. Default value is 1,253.
- **-fastexp [implementation]**
This option is used to select an alternative implementation for the exponential function (e^x), which has a significant impact on performance:
 - 0: (default) Use the `exp` function included in the built-in math library with double precision.
 - 1: Use the `exp` function included in the built-in math library with simple precision (not recommended together with the `-double-precision` option).
 - 2: Use a very efficient and fast implementation to compute an accurate approximation of e^x using double-precision arithmetic.
 - 3: Use a very efficient and fast implementation to compute an accurate approximation of e^x using simple precision arithmetic (not recommended together with the `-double-precision` option).

Conclusions

In the field of bioinformatics research, phylogenies are of utmost importance. Regrettably, the search for the optimal phylogenetic tree imposes significant computational requirements, and most modern cutting-edge tools struggle to handle exceptionally large datasets in a timely manner.

In this work, we introduce the latest version of VeryFastTree, which incorporates numerous performance optimizations and new features. Experimental results establish VeryFastTree as the fastest tool in the state-of-the-art for ML phylogeny estimation. For instance, it is capable of processing massive datasets containing 1 million taxa in just 36 hours, which is several times faster than other tools. In this way, VeryFastTree enables the processing of datasets that would otherwise be intractable or require excessively high computing times. Despite its exceptional speed, it produces trees with an accuracy level comparable to that of FastTree-2. On the other hand, a noteworthy new characteristic of VeryFastTree is what we call *disk computing*, which allows the processing of extremely large datasets on low-end servers with limited memory resources.

Finally, we would like to emphasize that VeryFastTree can also serve as the foundation for building an initial tree, which can subsequently be optimized using the latest developments in online phylogenetics [24, 25] or be used as a first step of disjoint tree mergers [4, 8, 9, 26].

Availability of Source Code and Requirements

- Project name: VeryFastTree
- Project homepage: <https://github.com/citiususc/veryfasttree>
- BiotoolsID: biotools:veryfasttree
- RRID:SCR_023594
- Operating system(s): Linux, Windows, and macOS
- Programming language: C/C++
- License: GNU GPL-3.0

Abbreviations

FT2: FastTree-2; ML: maximum likelihood; MP: maximum parsimony; MSA: multiple sequence alignment; NNI: nearest-neighbor interchange; SPR: subtree pruning and regrafting; VFT3: first version of VeryFastTree; VFT4: latest version of VeryFastTree introduced in this work.

Acknowledgments

The authors thank CESGA (Galicia, Spain) for providing access to their supercomputing facilities.

Author Contributions

César Piñero (Methodology [Equal], Software [Lead], Validation [Lead], Writing—original draft [Equal])
 Juan C. Pichel (Conceptualization [Lead], Methodology [Equal], Revision [Lead], Supervision [Lead], Writing—original draft [Equal])

Funding

This work was supported by MICINN (PLEC2021-007662, PID2022-137061OB-C22), Xunta de Galicia (ED431G 2019/04, ED431F

2020/08, ED431C 2022/16), and the European Regional Development Fund.

Data Availability

The datasets supporting the results of this article were obtained as follows:

- *Large* dataset was obtained from the FastTree-2 tool website [17].
- *Very Large* dataset was obtained from the Greengenes database [27].
- *Ultra-Large* dataset was obtained from the Kim Lab for Computational Evolutionary Biology (University of Pennsylvania) repository [28].
- 5,000-sequences simulated protein alignments dataset was obtained from the FastTree-2 tool website [17].

Supporting data and an archival copy of the code are available via the GigaScience repository, GigaDB [29].

Competing Interests

The authors declare that they have no competing interests.

References

1. Warnow T. Computational Phylogenetics: An Introduction to Designing Methods for Phylogeny Estimation. Cambridge, UK: Cambridge University Press; 2017.
2. Kozlov AM, Darriba D, Flouri T, et al. RAXML-NG: a fast, scalable and user-friendly tool for maximum likelihood phylogenetic inference. *Bioinformatics* 2019;35(21):4453–55. <https://doi.org/10.1093/bioinformatics/btz305>.
3. Minh BQ, Schmidt HA, Chernomor O, et al. IQ-TREE 2: new models and efficient methods for phylogenetic inference in the genomic era. *Mol Biol Evol* 2020;37(5):1530–34. <https://doi.org/10.1093/molbev/msaa015>.
4. Park M, Zaharias P, Warnow T. Disjoint tree mergers for large-scale maximum likelihood tree estimation. *Algorithms* 2021;14(5):148. <https://doi.org/10.3390/a14050148>.
5. Price MN, Dehal PS, Arkin AP. FastTree 2—approximately maximum-likelihood trees for large alignments. *PLoS One* 2010;5(3):1–10. <https://doi.org/10.1371/journal.pone.0009490>.
6. Piñero C, Abuín JM, Pichel JC. VeryFastTree: speeding up the estimation of phylogenies for large alignments through parallelization and vectorization strategies. *Bioinformatics* 2020;36(17):4658–59. <https://doi.org/10.1093/bioinformatics/bta582>.
7. Nelesen S, Liu K, Wang LS, et al. DACTAL: divide-and-conquer trees (almost) without alignments. *Bioinformatics* 2012;28(12):i274–82. <https://doi.org/10.1093/bioinformatics/bts218>.
8. Molloy EK, Warnow T. TreeMerge: a new method for improving the scalability of species tree estimation methods. *Bioinformatics* 2019;35(14):i417–26. <https://doi.org/10.1093/bioinformatics/btz344>.
9. Smirnov V, Warnow T. Unblended disjoint tree merging using GTM improves species tree estimation. *BMC Genomics* 2020;21:1–17. <https://doi.org/10.1186/s12864-020-6605-1>.
10. Heath MT. A tale of two laws. *Int J High Perform Comput Appl* 2015;29(3):320–30. <https://doi.org/10.1177/1094342015572031>.

11. Babel L, Kellerer H, Kotov V. The k-partitioning problem. *Math Method Oper Res* 1998;47(1):59–82. <https://doi.org/10.1007/BF01193837>.
12. Price MN, Dehal PS, Arkin AP. FastTree: computing large minimum evolution trees with profiles instead of a distance matrix. *Mol Biol Evol* 2009;26(7):1641–50. <https://doi.org/10.1093/molbev/msp077>.
13. Piñeiro C. Veryfasttre. Bioconda package. 2023. <https://anaconda.org/bioconda/veryfasttree>. Accessed 16 October 2023.
14. Piñeiro C. Veryfasttree. MacPorts package. 2023. <https://ports.macports.org/port/veryfasttree/>. Accessed 16 October 2023.
15. Piñeiro C, Pichel JC. BigSeqKit: a parallel big data toolkit to process FASTA and FASTQ files at scale. *GigaScience* 2023;12:giad062. <https://doi.org/10.1093/gigascience/giad062>.
16. CESGA (Galician Supercomputing Center). Computing infrastructures. <https://www.cesga.es/en/infrastructures/computing/>. Accessed 16 October 2023.
17. Price MN. FastTree. <http://www.microbesonline.org/fasttree/>. Accessed 16 October 2023.
18. Liu K, Linder CR, Warnow T. RAXML and FastTree: comparing two methods for large-scale maximum likelihood phylogeny estimation. *PLoS One* 2011;6(11):1–11. <https://doi.org/10.1371/journal.pone.0027731>.
19. Lees JA, Kendall M, Parkhill J, et al. Evaluation of phylogenetic reconstruction methods using bacterial whole genomes: a simulation based study. *Wellcome Open Res* 2018;3:1–37. <https://doi.org/10.12688/wellcomeopenres.14265.2>.
20. Smirnov V, Warnow T. Phylogeny estimation given sequence length heterogeneity. *Syst Biol* 2021;70(2):268–82. <https://doi.org/10.1093/sysbio/syaa058>.
21. Sayyari E, Whitfield JB, Mirarab S. Fragmentary gene sequences negatively impact gene tree and species tree reconstruction. *Mol Biol Evol* 2017;34(12):3279–91. <https://doi.org/10.1093/molbev/msx261>.
22. Huerta-Cepas J, Serra F, Bork P. ETE 3: reconstruction, analysis, and visualization of phylogenomic data. *Mol Biol Evol* 2016;33(6):1635–38. <https://doi.org/10.1093/molbev/msw046>.
23. Tatusov RL, Natale DA, Garkavtsev IV, et al. The COG database: new developments in phylogenetic classification of proteins from complete genomes. *Nucleic Acids Res* 2001;29(1):22–28. <https://doi.org/10.1093/nar/29.1.22>.
24. Ye C, Thornlow B, Hinrichs A, et al. matOptimize: a parallel tree optimization method enables online phylogenetics for SARS-CoV-2. *Bioinformatics* 2022;38(15):3734–40. <https://doi.org/10.1093/bioinformatics/btac401>.
25. Turakhia Y, Thornlow B, Hinrichs A, et al. Pandemic-scale phylogenomics reveals the SARS-CoV-2 recombination landscape. *Nature* 2022;609(7929):994–97. <https://doi.org/10.1038/s41586-022-05189-9>.
26. Zhang QR, Rao S, Warnow T. New absolute fast converging phylogeny estimation methods with improved scalability and accuracy. In: 18th International Workshop on Algorithms in Bioinformatics (WABI). Parida L, Ukkonen E, eds. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik; 2018.
27. Second Genomes Inc. The StrainSelect and Greengenes Databases. <https://greengenes.secondgenome.com>. Accessed 16 October 2023.
28. University of Pennsylvania. Kim Lab for Computational Evolutionary Biology. <https://kim.bio.upenn.edu/software/index.shtml>. Accessed 16 October 2023.
29. Piñeiro C, Pichel JC. Supporting data for “Efficient Phylogenetic Tree Inference for Massive Taxonomic Datasets: Harnessing the Power of a Server to Analyze 1 Million Taxa.” *GigaScience Database*. 2024. <http://dx.doi.org/10.5524/102551>.