Scalable Phylogeny Reconstruction with Disaggregated Near-memory Processing

NIKOLAOS ALACHIOTIS, University of Twente, The Netherlands
PANAGIOTIS SKRIMPONIS, NYU Tandon School of Engineering, USA
MANOLIS PISSADAKIS, Technical University of Crete, Greece
DIONISIOS PNEVMATIKATOS, National Technical University of Athens, Greece

Disaggregated computer architectures eliminate resource fragmentation in next-generation datacenters by enabling virtual machines to employ resources such as CPUs, memory, and accelerators that are physically located on different servers. While this paves the way for highly compute- and/or memory-intensive applications to potentially deploy all CPUs and/or memory resources in a datacenter, it poses a major challenge to the efficient deployment of hardware accelerators: input/output data can reside on different servers than the ones hosting accelerator resources, thereby requiring time- and energy-consuming remote data transfers that diminish the gains of hardware acceleration. Targeting a disaggregated datacenter architecture similar to the IBM dReDBox disaggregated datacenter prototype, the present work explores the potential of deploying custom acceleration units adjacently to the disaggregated-memory controller on memory bricks (in dReDBox terminology), which is implemented on FPGA technology, to reduce data movement and improve performance and energy efficiency when reconstructing large phylogenies (evolutionary relationships among organisms). A fundamental computational kernel is the Phylogenetic Likelihood Function (PLF), which dominates the total execution time (up to 95%) of widely used maximum-likelihood methods. Numerous efforts to boost PLF performance over the years focused on accelerating computation; since the PLF is a data-intensive, memory-bound operation, performance remains limited by data movement, and memory disaggregation only exacerbates the problem. We describe two near-memory processing models, one that addresses the problem of workload distribution to memory bricks, which is particularly tailored toward larger genomes (e.g., plants and mammals), and one that reduces overall memory requirements through memory-side data interpolation transparently to the application, thereby allowing the phylogeny size to scale to a larger number of organisms without requiring additional memory.

CCS Concepts: • **Computer systems organization** → **Architectures**; **Reconfigurable computing**; • **Hardware** → **Integrated circuits**; **Reconfigurable logic and FPGAs**; **Reconfigurable logic applications;**

Additional Key Words and Phrases: Disaggregated datacenter, dReDBox, near-memory processing, phylogenetics, RAxML

Authors' addresses: N. Alachiotis, University of Twente, Enschede 7500, The Netherlands; email: n.alachiotis@utwente.nl; P. Skrimponis, NYU Tandon School of Engineering, Brooklyn, NY 11201; email: ps3857@nyu.edu; M. Pissadakis, Technical University of Crete, Chania 73100, Greece; email: epissadakis@gmail.com; D. Pnevmatikatos, National Technical University of Athens, Athens 15780, Greece; email: pnevmati@cslab.ece.ntua.gr.

1936-7406/2021/12-ART25
https://doi.org/10.1145/3484983

ACM Reference format:
Nikolaos Alachiotis, Panagiotis Skrimponis, Manolis Pissadakis, and Dionisios Pnevmatikatos. 2021. Scalable Phylogeny Reconstruction with Disaggregated Near-memory Processing. *ACM Trans. Reconfigurable Technol. Syst.* 15, 3, Article 25 (December 2021), 32 pages.
https://doi.org/10.1145/3484983

1 INTRODUCTION

In computational biology, phylogenetic inference methods are used to reconstruct the evolutionary history of a collection of organisms based on molecular genetic data (DNA or protein sequences). An inferred evolutionary history is represented by a phylogeny, or phylogenetic tree, which is an unrooted bifurcating (binary) tree where the organisms under investigation (taxa) are located at the leaves, and the inner nodes represent extinct common ancestors. Phylogenetic trees find practical application in a wide range of scientific and industrial fields, such as conservation biology (exposing illegal whale hunting [1], prioritizing populations in the wake of the global biodiversity crisis [2, 3]), epidemiology (studying the evolution and dynamics of infectious viruses and bacteria [4, 5]), forensics (characterizing HIV transmission networks [6]), and drug development (identifying new medicinal plant species [7], tracing the evolution of antibodies to develop precision vaccines [8]). Stamatakis [9] provides an overview of significant applications of phylogenetic trees in medical research, while Bader et al. [10] list prominent industrial applications of high-performance computing for phylogenetic inference, such as for commercial drug discovery.

Several computationally inexpensive methods to infer phylogenies based on clustering techniques are known, e.g., Neighbor–Joining [11] and the Unweighted Pair Group Method with Arithmetic mean [12]. However, considerably more advanced methods like **Maximum Likelihood Estimation** (**MLE**) and **Bayesian Inference** (**BI**) are preferred in real-world phylogenetic studies because they yield more robust phylogenies based on a stronger statistical foundation [13]. A fundamental computational kernel, employed by both MLE and BI phylogenetic methods, is the **Phylogenetic Likelihood Function** (**PLF**) [14], which is used for evaluating a phylogeny by computing a likelihood score for a given tree topology. The PLF dominates execution times and memory requirements of the most widely used phylogenetic inference tools, such as BEAST [15], FastTree [16], RAxML [17], and MrBayes [18] (over 150,000 citations collectively). For these tools, Izquierdo-Carrasco et al. [19] report that the PLF occupies between 80% and 95% of the total execution times and over 70% of the total RAM requirements.

Expectedly, numerous efforts have been made to accelerate the PLF, employing various technologies, from multi-core processors [20, 21] and supercomputers [22, 23], to FPGAs [24, 25] and GPUs [19, 26], to CGRA-based solutions [27, 28] and dedicated NoCs [29, 30]. These efforts, however, predominantly concentrated on accelerating computation, thus remaining bounded by the memory accesses since the PLF is a data-intensive, memory-bound operation. This problem is expected to intensify as phylogeny sizes continue to grow in terms of number of organisms [31], a trend that is sustained by continuous advances in DNA sequencing technologies [32], which improve sequencing accuracy and reduce costs. Izquierdo-Carrasco et al. [19] also report that excessive memory requirements, which increase linearly with the number of organisms, represent the main limiting factor for large-scale, real-world phylogenetic analyses. For instance, an MLE-based phylogenetic analysis that deployed the PLF to reconstruct the phylogeny of 1,481 species using genetic material (DNA sequences) that roughly corresponded to the total size of 20,000 human genes, required 1TB of main memory [19]. These excessive memory requirements indicate that traditional datacenter architectures, which typically provide between 256 GB and 768 GB of main memory per server tray, are unlikely to meet the memory demand in future large-scale analyses.

Disaggregated computer architectures [33–35], which enable virtual machines to employ resources (CPUs, memory, accelerators) that are physically located on different servers, represent a promising solution to effectively meet memory requirements of future analyses, since an application can potentially deploy all memory resources in a data center. This, however, poses a major challenge to the efficient deployment of hardware acceleration because input/output data can reside on different servers than the ones hosting accelerators, hence requiring time- and energy-consuming, remote-data transfers that can diminish the gains of hardware acceleration.

To this end, the present work explores the potential of performing PLF computations closer to the data within a FPGA-based computing environment with disaggregated memory to yield a scalable, as well as both time- and energy-efficient solution. We target a disaggregated computing architecture, similar to the **Disaggregated Recursive Datacentre-in-a-Box** (**dReDBox**) disaggregated datacenter prototype [34] recently presented by IBM, which allows to exploit the versatility that a software-defined datacenter provides in allocating and managing resources like compute, memory, and accelerator bricks (dReDBox terminology) on the cloud. Instead of employing high-performance accelerator bricks that induce remote-data transfers for the PLF, the underlying idea is to instantiate custom acceleration units, henceforth referred to as **Brick Processing Units** (**BPUs**), adjacently to the disaggregated-memory controller on each memory brick, which is implemented on reconfigurable logic. This paves the way for a future-proof, accelerated solution that efficiently accommodates larger phylogeny sizes by allowing to allocate the required amount of memory resources with near-data processing capacity in order to reconstruct a phylogeny of any size under given constraints, such as analysis time, energy consumption, and cost, among others.

The contribution of this work is three-fold:

— We present a streaming architecture for the BPU and an effective data allocation scheme that collectively extend the functionality of disaggregated memory bricks by computing the PLF using local data. This eliminates the remote-data transfers that would otherwise be required if discrete accelerator modules were deployed, and allows the aggregate throughput performance of multiple BPUs to scale linearly with the number of memory bricks.

— We describe a model of bulk-synchronous parallel processing [36] and devise a software/hardware barrier implementation that alleviates the overhead of synchronization for an increasing number of BPU-equipped memory bricks over several server trays.[1] A moderate to high number of memory bricks (in the hundreds) are expected to be required to accommodate analyses of ultra-long sequences [38] or large genomes (plants or mammals).

— We devise a BPU-based data interpolation engine that relies on networks of BPUs to trade memory for computation. Introducing topology-aware functionality within a memory brick enables the interpolation engine to operate transparently to the host processor to provide the illusion of a larger physical memory than actually present in the system. This paves the way for considerable improvements in energy consumption for large-scale phylogenetic analyses since only a fraction of the PLF data need to be stored and accessed in main memory, whereas the rest are computed on-the-fly when needed.

The remainder of this article is organized as follows: Section 2 describes the mathematical foundation for computing the PLF, while Section 3 presents the dReDBox disaggregated datacenter architecture. Section 4 discusses related work on previous acceleration efforts. Thereafter, Section 5 presents the BPU architecture, the data allocation scheme, and the bulk-synchronous parallel processing model, while Section 6 describes the BPU-based interpolation engine for lower memory

[1]A dReDBox server tray can contain up to 16 bricks [37].

Table 1. List of Acronyms and Abbreviations

AC	ACCESS
APV	Ancestral probability vector
AVX2	Advanced vector extensions 2
BI	Bayesian inference
BPU	Brick processing unit
DMC	Disaggregated-memory controller
dRedBox	Disaggregated recursive datacenter-in-a-Box
EX	EXECUTE
FPA	Felsenstein's pruning algorithm
II	Initiation interval
MLE	Maximum Likelihood Estimation
MSA	Multiple sequence alignment
PE	Processing element
PLF	Phylogenetic likelihood function
PTD	Partial traversal descriptor
RF	Register file
RT	Reduction tree
SPR	Subtree pruning and regrafting
VEUPS	Vector entry updates per second

```
sequence1    ATCATACCCCT-CCAACTAG-ATTCC
sequence2    --CCTAC-C-TCCCAACTAGGTT-CC
sequence3    ATCATAC-C-TCCCACG-AGGTTT--
sequence4    AT---AC-C-TCACAAGTAGGTTTC-
```

Fig. 1. Example of a multiple sequence alignment of 4 sequences ($n = 4$) with 26 nucleotide characters each ($m = 26$).

requirements. Section 7 provides implementation details, and Section 8 presents the experimental setup and performance evaluation. We conclude in Section 9. A list of abbreviations and acronyms used throughout the article is provided in Table 1.

2 PHYLOGENETIC LIKELIHOOD FUNCTION

This section introduces the mathematical foundation of the PLF. The starting point for a phylogenetic analysis is a list of organisms and their associated DNA sequences. Since these sequences can differ in length, a **multiple sequence alignment** (**MSA**) is computed prior to reconstructing a phylogenetic tree, i.e., an $n \times m$ matrix of n sequences with m nucleotide characters each. This is achieved by first determining which nucleotides among the organisms are homologous (share a common evolutionary history), and then inserting gaps into the sequences in a way that each alignment site (a column of the $n \times m$ matrix) conveys information about the history of the organisms under study. An example of an MSA of four simulated sequences in provided in Figure 1. Some of the most widely used MSA software tools are MUSCLE [39], MAFFT [40], and ClustalW [41].

When the MSA is created, a phylogenetic tree is reconstructed using an iterative procedure that employs a tree-search strategy in conjunction with a scoring function. The tree-search strategy, e.g., nearest neighbor interchange [42], **subtree pruning and regrafting** (**SPR**) [43], or tree bisection and reconnection [44], is a sequence of topology-rearrangement steps, with every step

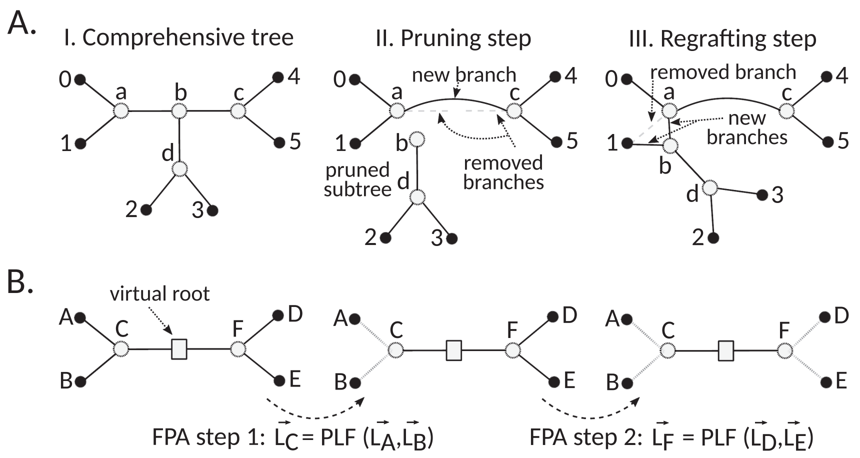

Fig. 2. (A) An SPR iteration applied on a phylogenetic tree with six tips (0–5) and four inner nodes ($a - d$). The comprehensive tree (tree before pruning) contains all tip and inner nodes (I). In the pruning step (II), the subtree that contains tips 2 and 3 and inner nodes b and d is pruned, along with the branches that used to connect inner node b to the rest of the tree. In the regrafting step (III), the pruned subtree is reattached to the rest of the tree on the branch that used to connect inner node a to tip 1, creating a new topology. (B) FPA allows the PLF to be recursively applied on a fixed tree topology to calculate the likelihood of the tree. For the phylogenetic tree with four tips (A, B, D, E), two inner nodes (C and F), and a random placement of the virtual root on the branch connecting inner nodes C and F, two FPA steps are needed before one can calculate the likelihood of the tree. In FPA step 1, the PLF is used to compute the probability vector of inner node C using the probability vectors of tips A and B. In FPA step 2, the PLF is used to compute the probability vector of inner node F using the probability vectors of tips D and E. Thereafter (not shown in the figure), the likelihood of the tree can be calculated using inner nodes C and F. The light gray lines show the gradual reduction of the tree to a single branch (the one with the virtual root) using the FPA.

leading to a new tree topology that is evaluated using the scoring function. An SPR step is illustrated in Figure 2(A), as an example. It consists of two phases: a subtree is first removed from the comprehensive tree (pruning, Figure 2(A.II)) and then reattached to another branch (regrafting, Figure 2(A.III)). This represents one full SPR step that starts from one tree topology and generates a new one. SPR moves are iteratively applied on a region of the overall tree topology (subtree) before moving on to another tree region. Throughout tree searching, the tree-search strategy generates different phylogenies that are qualitatively assessed using the scoring function, with the aim to find the best-scoring tree, i.e., the topology that best explains (fits) the aligned sequence data.

As already mentioned, the PLF is employed as the scoring function by all MLE and BI phylogenetic inference methods. Computing the PLF relies on **Felsenstein's Pruning Algorithm (FPA)** [14], which computes a likelihood score for a tree topology of any size by recursively applying the PLF on pairs of child nodes that share an immediate common ancestor until an arbitrarily placed virtual root is reached. The virtual root is an additional tree node that can be placed anywhere on a tree to direct the FPA algorithm in applying the PLF. Due to mathematical properties of the evolutionary models used in phylogenetics (time reversibility), the likelihood of the tree does not change with the position of the virtual root. Figure 2(B) shows the required sequence of PLF invocations according to FPA to compute the likelihood score of a 4-taxon tree. The process begins at the tips and proceeds until the initial tree is reduced to a single branch, i.e., the one with the virtual root. In the first step (FPA step 1 in the figure), the PLF computes the probability vector of

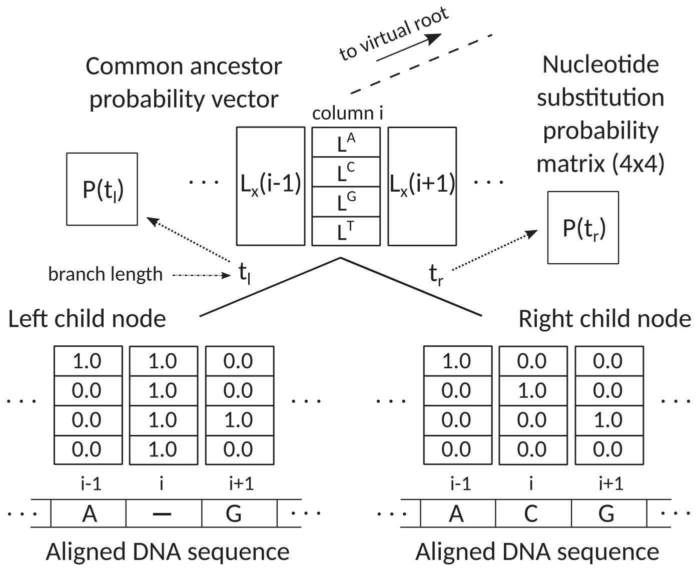

Fig. 3. Schematic representation of a single FPA step. The child nodes are tips, and the associated aligned sequences are shown below the probability vectors (adapted from [45]).

inner node C based on the tips A and B. In the next step (FPA step 2 in the figure), the PLF computes the probability vector of inner node F based on the tips D and E. Thereafter, the likelihood of the tree can be calculated using inner nodes C and F.

Figure 3 depicts a schematic representation of a single FPA step, e.g., the first step in Figure 2(B). Each node x is represented by a probability vector $\vec{L_x}$ that comprises m entries, where m is the alignment length. When x is an inner node, such as node C in Figure 2(B), the probability vector entry $\vec{L_x}(i)$, $i = 1 \ldots m$, contains the four probability values L^A, L^C, L^G, and L^T of observing nucleotides A, C, G, and T, respectively, at location i of x based on the observed nucleotides in MSA column i. The respective probability values per location i of a tip node, such as nodes A and B in Figure 2(B), are set to 0.0 or 1.0, according to the observed nucleotide character at location i of the corresponding aligned sequence in the MSA.

Given probability vectors $\vec{L_A}$ and $\vec{L_B}$ of child nodes A and B, respectively (see FPA step 1 in Figure 2(B), for instance), each of the four probability values $\vec{L}^u_C(i)$, $u \in N$ and $N = \{A, C, G, T\}$, at location i of the probability vector $\vec{L_C}$ that describes the immediate common ancestor C is computed using Equation (1):

$$\vec{L}^u_C(i) = \left(\sum_{s \in N} P_{u \to s}(t_l) \times \vec{L}^s_A(i) \right) \times \left(\sum_{s \in N} P_{u \to s}(t_r) \times \vec{L}^s_B(i) \right), \tag{1}$$

where t_l and t_r are the lengths of the branches that connect parent node C with child nodes A and B, respectively, while $P_{u \to s}(t)$ is the nucleotide substitution probability for a nucleotide u to mutate to a nucleotide s given a branch length t. The nucleotide substitution probabilities are computed using Equation (2):

$$P(t) = e^{Qt}, \tag{2}$$

where t is the branch length (it essentially represents the evolutionary time between two nodes), and Q is a 4×4 matrix that describes a statistical model of nucleotide substitution by comprising

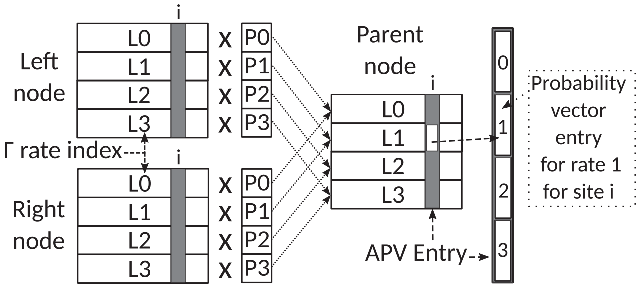

Fig. 4. PLF computation under the Γ model with four discrete rates. Equation (1) is independently computed for each rate, leading to each APV entry comprising 16 values when DNA data are processed with four discrete rates.

instantaneous transition probabilities for a nucleotide u to mutate to s within time dt. The most complex and commonly used nucleotide substitution model is the general time reversible [46].

In real-world analyses, the statistical model of nucleotide substitution is extended by additional parameters to account for rate heterogeneity among alignment sites, i.e., the biological fact that genes evolve at different rates. A widely used model to describe rate variation among sites in practical phylogenetic inference is the Γ model [47], which assumes that rates over sites are random variables drawn from a Γ distribution, integrating the log-likelihood over the Γ function. To yield a computationally tractable solution, this integral is approximated by discretizing the Γ function into, typically, four or eight discrete rates. Computing the PLF under the Γ model entails the calculation of Equation (1) independently for each discrete rate, resulting to the calculation of a probability vector $\vec{L_x}$ per discrete rate per inner node x, as illustrated in Figure 4 for the case of 4 discrete rates. When N discrete Γ rates are used for phylogenetic inference based on DNA sequences, each probability vector entry $\vec{L_x}(i)$ contains a total of $N \times 4$ probability values per genomic location i ($i = 1 \ldots m$, where m is the MSA length). The entire vector of $\vec{L_x}(i)$ entries ($i = 1 \ldots m$) represents a tree node and is henceforth referred to as **Ancestral Probability Vector** (**APV**), while an $\vec{L_x}(i)$ for a given i is an APV entry.

When the branch with the virtual root is reached, following a number of FPA steps, e.g., the branch that connects inner nodes C and F after the second FPA step in Figure 2(B), a likelihood score $l(i)$ per site i is computed based on the probability vector $\vec{L_{vr}}$ at the virtual root using Equation (3):

$$l(i) = \sum_{s \in N} \pi_s \times \vec{L}^s_{vr}(i), \tag{3}$$

where π_s, $s \in N$, and $N = \{A, C, G, T\}$, are the prior probabilities (typically referred to as base frequencies) of observing nucleotides A, C, G, and T at the virtual root, and are empirically drawn from the MSA. The final likelihood score of the tree is computed as the sum of the logarithm of the per-site likelihood scores using Equation (4):

$$LH = \sum_{i=1}^{m} log(l(i)). \tag{4}$$

It should be noted that state-of-the-art MLE inference programs, such as RAxML [17] (used in this work as the reference software), deploy a Newton–Raphson iterative procedure to optimize the branch lengths and improve the final likelihood score given the tree topology and the nucleotide substitution model. Izquierdo-Carrasco et al. [19] report that branch-length optimization in RAxML, which relies on the PLF to optimize a branch, accounts for approximately 30% of the total execution time. Our RAxML profiling analysis (results not shown) revealed that Equation (1) accounts for 85% of the total execution time, whereas Equations (3) and (4) together account for less than 4% of the analysis time (only employed at the root for the calculation of the final log-likelihood score). Thus, our efforts to shift computation to disaggregated memory bricks concentrated on Equation (1).

3 THE dReDBox DISAGGREGATED ARCHITECTURE

Typical data centers are usually built by replicating a single (or a few) baseline, monolithic building blocks (blades), with predetermined and fixed resource ration (i.e., CPU, memory and accelerator capacity). While this is effective and convenient for quick deployment and logistics, the impact of this rigidity has important ramifications: it limits system resource utilization, degrades energy proportionality, and mandates costly upgrade cycles [34, 48].

To overcome the fixed architectural design of traditional data centers and achieve better resource allocation and proportionality, dReDBox disaggregates all the major resource types into "bricks". A flexibly defined set of processor, and FPGA-based memory and accelerator bricks are packaged in a tray. Trays are used to build tightly-coupled rack-level systems and data centers. The center of the dReDBox architecture is a high-speed, low-latency opto-electronic fabric that brings physically distant bricks closely in terms of latency and bandwidth. A software-defined control plane performs resource allocation and orchestration, and exploits the system flexibility to fulfill the resource needs of the applications (or virtual machines) running in the system to form a modular, vertically-integrated system [49]. A key outcome of the dRedBox is the IBM Thymesis Flow [35] that achieves similar resource and resource-management flexibility in mainstream IBM systems.

Figure 5 illustrates the dReDBox tray architecture. A tray implements four different networks, a low-latency high-speed electrical network, an Ethernet network, a low-latency high-speed optical network, and a PCIe network that collectively provide brick-to-brick connectivity. Remote memory that is mapped to memory bricks on other trays is accessed using optical and electrical low-latency, high-speed networks. Memory bricks on the same tray are accessed through an electrical circuit crossbar switch (labeled as High Speed Electrical Switch in the figure) that connects directly to the GTH interface ports available on the **programmable logic** (**PL**) of the bricks. On a fully populated tray hosting 16 bricks, a maximum of 256 optical ports are used to fully interconnect the bricks of each tray. A **Mid-Board Optics** (**MBO**) device mounted on every brick converts the electrical signals coming from the GTH ports and aggregates them into a single fibre ribbon. An **Ethernet** (**ETH**) network is used for regular network communication and **board management communication BMC**. Bricks on the same tray interconnect via a PCIe interface. Communication between bricks on different trays within the same rack is provided via a PCIe switch. The PCIe interface is used for signalling and interrupts, as well as for attachment to remote peripherals.

Figure 6 illustrates the dReDBox compute brick that represents the main processing block in the system. It hosts local DDR memory for low-latency and high-bandwidth instruction read and read/write data access, and Ethernet and PCIe ports for data and system communication and configuration. The compute brick communicates with disaggregated memory and accelerator resources via the "Transaction Glue Logic, a dReDBox-specific IP implemented on the PL, as shown in the figure. System interconnection to disaggregated resources occurs via multiple ports leading to circuit-switched tray- and rack-level interconnects.

A key feature of dReDBox is the broad and extensible pool of memory resources. A software orchestrator is used to partition and (re)assign these resources among all the compute nodes and their respective virtual machines in real-time. Figure 7 depicts the memory brick architecture featuring a Xilinx Zynq Ultrascale+ MPSoC and external DDR/HMC memory chips. This brick supports multiple communication links that provide a higher aggregated bandwidth to a single compute

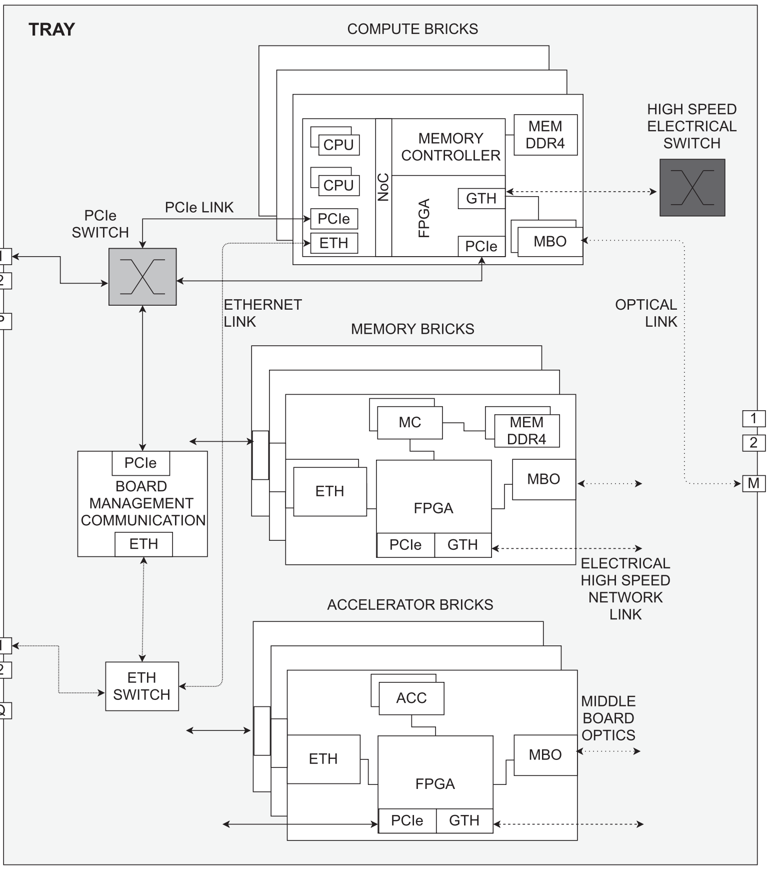

Fig. 5. Sample of the dReDBox tray architecture with multiple bricks interconnected through optical and electrical interconnection networks.

node or parallel connections to multiple compute nodes. This feature is utilized differently based on the resource allocation policy. The orchestrator can allocate either shared or private partitions for each client. Thus, the memory bricks facilitate the communication between multiple compute nodes through shared memory spaces. Private memory partitions require an extra protection mechanism. The glue logic of the memory brick implements both the protection and translation mechanism controlled by the orchestrator. The protection mechanism promotes fine-grained memory allocations, while the translation mechanism maps external memory requests to local addresses, increasing the flexibility of the system.

Another key element of dReDBox is the accelerator resources that can boost application performance on a near-data processing scheme [50]. The dReDBox aims at reducing the communication overhead between multiple compute nodes using local hardware accelerators deployed on the dReDBox accelerator bricks. This brick has been used to accelerated applications in various domains, such as machine learning [51], population genomics [52], and phylogenetic inference [53].

Figure 8 illustrates the dReDBox accelerator brick architecture featuring a Xilinx Zynq Ultrascale+ MPSoC. This system comprises static and dynamic infrastructure. First, the static infrastructure provides all the required modules to (i) support dynamic hardware reconfiguration;

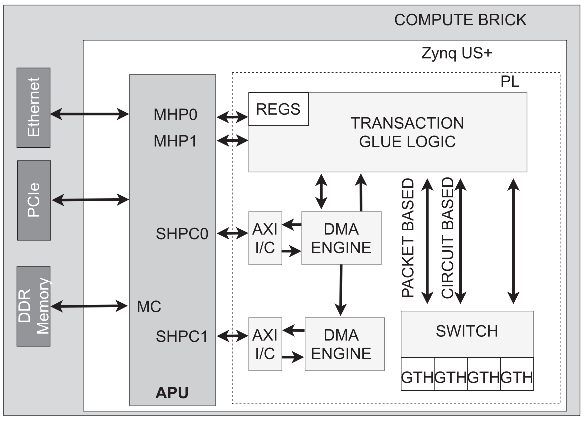

Fig. 6. Block diagram of a dReDBox compute brick. The MPSoC integrates an APU for software execution. The on-chip programmable logic on the SoC is used to host the transaction glue logic, housekeeping state, and communication logic, required for accessing disaggregated resources. The local DMA engines allow the system software to efficiently migrate pages from remote memory regions to local DDR memory.

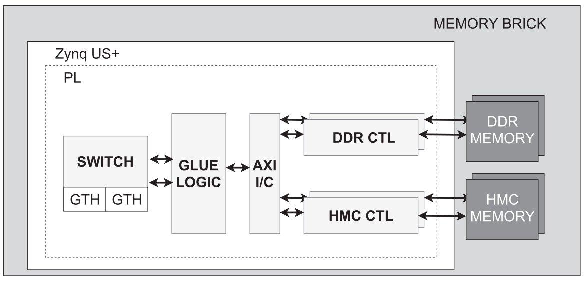

Fig. 7. The architecture of the dReDBox memory brick; the local switch forwards system/application data to the memory brick glue logic, which interfaces different memory module technologies

(ii) establish communication links with remote compute/memory bricks; (iii) interface with hardware accelerators. To facilitate dynamic partial reconfiguration, dReDBox is using a lightweight middleware [52] running on the **Application Processing Unit** (**APU**). Using the ReFiRe API [52], an application running on a virtual machine establishes a communication link with the middleware to control the hardware accelerators and perform basic operations. Through this API, the middleware is: (i) receiving the partial bitstreams, the configuration parameters, and the input data for each accelerator slot; (ii) storing the received data in the APU-DDR memory; (iii) reconfiguring the partial reconfigurable region (accelerator slot) through the PCAP-port; and (iv) collecting the results from each accelerator slot and sends them back to the host. Using the accelerator brick glue logic, the middleware interfaces with the network interconnects/switch for transferring the data from/to compute and memory bricks. This work utilizes the ReFiRe framework to deploy the BPUs in dReDBox as described in Section 7. Second, the dynamic infrastructure (accelerator slot) is a predefined partially reconfigurable region in the PL that hosts the hardware accelerators. Figure 8 shows this region that contains a set of high-speed transceivers (e.g., GTHs), registers, and AXI memory-mapped interfaces. The transceivers provide a direct connection between

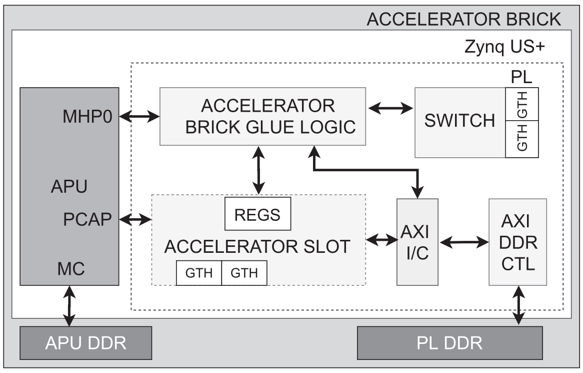

Fig. 8. The architecture of the dReDBox accelerator brick for accommodating application-specific accelerators.

the accelerator brick and remote compute/memory bricks. The AXI-Lite register file provides monitoring (e.g., debugging) and control of the hardware accelerator through the accelerator brick glue logic. The AXI memory-mapped interface provides a direct connection to the PL-DDR.

4 RELATED WORK

This section reviews FPGA accelerators for the PLF and algorithms that reduce memory footprint.

Mak and Lam [54] presented an accelerated system for MLE-based phylogeny reconstruction based on a genetic algorithm [55] that executes in software, coupled with a PLF accelerator mapped to reconfigurable hardware, reporting up to 100× faster execution than a pure software implementation [55]. The proposed architecture, however, implements the Jukes–Cantor model [56] of nucleotide substitution, which is rarely employed in real-world analyses due to oversimplifications in the statistical model of DNA substitution [57]. This work was later extended (Mak and Lam [58]) by introducing an embedded processor onto the FPGA fabric to reduce communication overhead, as well as by further parallelizing the likelihood computation on the chip. Yet, performance was measured on trees with as low as 4 taxa, and DNA sequences of up to 500 nucleotide characters.

Alachiotis et al. [24] described a pipelined architecture for computing the PLF on DNA data, reporting up to 7.5× faster execution than 16 CPU cores when processing 512 taxa. The proposed architecture, however, could only accommodate fully balanced trees, thereby preventing the seamless integration with tree-search strategies. The authors subsequently presented a topology-agnostic architecture [45] to address the aforementioned limitation by exploiting PLF parallelism across alignment sites, rather than tree nodes.

Zierke and Bakos [25] designed an FPGA accelerator architecture for the PLF, driven by the requirements of Bayesian Markov Chain Monte Carlo-based inference methods. The architecture performs numerical scaling to prevent underflow in the case of large phylogenies, achieving up to 8.7× faster processing than the sequential execution of the widely used software tool MrBayes [59]. Bayesian inference methods, however, do not employ numerical optimization routines for branch-length optimization. Consequently, the employed PLF is less complex than in MLE-based methods, which typically deploy Newton–Raphson branch-length optimization procedures that rely on the PLF to improve likelihood scores.

Berger et al. [60] presented an accelerator architecture particularly optimized for 4-state input data, i.e., nucleotide characters, and described how to efficiently handle n-state data, with $n > 4$, e.g., protein sequences or RNA secondary-structure data. The authors also proposed a flexible communication mechanism to enable the widely used software RAxML [17] to offload computation to the FPGA accelerator, reporting up to 4.3× faster processing than a heavily optimized sequential implementation that employs 256-bit advanced vector extensions intrinsic instructions.

Jin and Bakos [61] extended the BEAGLE [62] library for statistical phylogenetics to support parallel computation of the PLF on a multi-FPGA platform. Using 32 pipelined **processing elements** (**PEs**) across four FPGAs, the proposed system delivered 40× and 3× faster execution than BEAGLE's CPU and GPU implementations, respectively. The authors reported per-PE arithmetic intensity of 2.03 floating-point operations per byte, and per-FPGA power consumption of 23 Watts, concluding that a prerequisite to the effective use of data-parallel architectures for improved PLF performance is to couple high memory bandwidth with high memory efficiency.

While the majority of the aforementioned acceleration efforts mostly focus on the computational challenges of the PLF, algorithmic solutions that adopt a data-centric approach to improve memory efficiency and reduce overall memory requirements have also been reported, albeit to a lesser extent. Stamatakis and Ott [20] initially observed that, when alignments with missing genes are analyzed, PLF computation times can be considerably reduced by omitting calculations on data that are not present in the MSA, which are commonly represented in memory as undetermined characters. Building upon this observation, Stamatakis and Alachiotis [63] introduced a set of algorithmic rules to search the tree space without requiring to allocate memory for the missing genes, thereby allowing to reduce the memory footprint proportionally to the amount of missing data in the MSA.[2] These approaches [20, 63], however, can only be applied to analyses of multi-gene MSAs with missing data, and thus do not represent generic memory-efficient solutions.

Izquierdo-Carrasco and Stamatakis [64] presented a generic method to compute the PLF with lower memory requirements, which does not depend on specific MSA traits, e.g., missing genes. Based on feedback by the RAxML community, the authors point out that "memory shortages are increasingly becoming a problem and represent *the* main limiting factor for large-scale phylogenetic analyses, especially at the genome level", and identify ancestral probability vectors as the dominant memory-consumption factor in present phylogenetic analyses. To alleviate this problem, the study introduced an out-of-core algorithm that only requires a fraction of the probability vectors to reside in memory at any point in time, whereas the rest are kept in the disk. Various replacement policies were used in order to exploit tree-search locality, observing up to 5× faster execution than relying on paging by the operating system.

Izquierdo-Carrasco et al. [65] observed, however, that computing the PLF out-of-core [64] is impractical, due the fact that a high number of slow read/write operations to/from the disk are required, leading to an order of magnitude longer execution times. The authors, therefore, proposed a more efficient approach to reduce memory requirements of large-scale phylogenetic analyses, which relies on trading memory for processing time. Similarly to the out-of-core approach [64], only a fraction of the probability vectors are stored in main memory, while the rest are recomputed, potentially several times, when required by the tree-search strategy, e.g., SPR [43]. The study reported between 14% and 40% longer execution times, on average, due to additional computation, when three simulated datasets with 1,500, 3,000, and 5,000 taxa are analyzed.

The review of literature on PLF challenges and solutions in this section reveals that research efforts have so far focused on either accelerating computation or reducing memory requirements. To the best of the author's knowledge, the work presented here is the first to investigate the potential of jointly accelerating the PLF while reducing memory requirements via near-memory computation within a disaggregated computing environment.

[2]The study reports up to 90% of so-called gappyness in real-world, multi-gene MSAs.

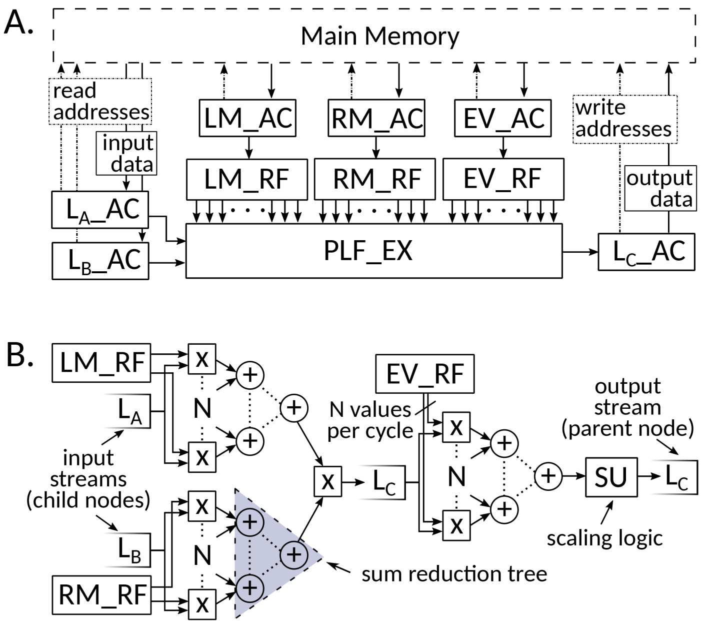

Fig. 9. (A) Top-level design of the decoupled access/execute BPU architecture. It consists of six ACCESS units (suffix AC), one EXECUTE unit (PLF_EX), and three register files (RF). The main memory is the off-chip DDR memory on the brick. (B) PLF_EX pipelined datapath. It computes Equation (1), performs multiplication with the inverted eigenvector, and scaling (when needed), using IEEE-754 double-precision floating-point arithmetic units.

5 BRICK PROCESSING UNIT (BPU)

This section introduces the BPU architecture in Section (5.1), presents a brick-aware memory layout in Section (5.2), and describes a bulk-synchronous parallel processing model in Section (5.3).

5.1 BPU Architecture

A practical PLF accelerator architecture that effectively meets critical requirements of large-scale, real-world analyses requires a highly versatile design that is agnostic to the tree-search strategy to facilitate cooperation with heuristic algorithms, which are inevitably employed to shorten inference times for phylogenies with many taxa.[3] Furthermore, arithmetic scaling is required to ensure that the accelerated system remains numerically stable as the number of taxa increases. Driven by these requirements, we devised a low-latency, highly parallel pipelined architecture to be deployed adjacently to the **disaggregated-memory controller** (**DMC**) on dReDBox memory bricks. The proposed BPU architecture is based on the RAxML PLF kernel, facilitating comparisons with previous PLF accelerators [24, 45, 60]. In addition, it performs a multiplication with the inverted eigenvector, a RAxML-specific numerical detail that is related to the calculation of the nucleotide substitution matrix *P* (see Equation (1)) based on an eigenvector/eigenvalue decomposition [67].

The BPU architecture, depicted in Figure 9(A), is based on a decoupled access-execute architectural paradigm [68, 69]. It is a dataflow streaming pipeline that consists of six ACCESS units and one EXECUTE unit, with the ACCESS units denoted in the figure with the suffix *_AC*, whereas the

[3]For as low as 50 taxa, the number of possible unrooted trees exceeds the number of atoms in the universe (approx. 10^{80}) [66].

EXECUTE unit is denoted with the suffix *_EX*. A single BPU invocation proceeds in two distinct sequential steps. First, the *LM_AC*, *RM_AC*, and *EV_AC* units retrieve the left and right probability matrices ($P(t_l)$ and $P(t_r)$ in Figure 3) and the inverted eigenvector from main memory, and store them into dedicated register files (denoted with the suffix *_RF* in the figure). Thereafter, the FIFO-based L_A*_AC* and L_B*_AC* units fetch the ancestral probability vectors L_A and L_B (account for all discrete Γ rates) that correspond to the left and right child nodes, respectively (see Figures 3 and 4), and stream them through the *PLF_EX* datapath. This computes Equation (1), performs a multiplication with the inverted eigenvector, and scales up the results when needed. The output, i.e., the ancestral probability vector L_C, is stored back to main memory through the FIFO-based L_C*_AC* unit. The ACCESS units that prefetch data into the register files do not contain FIFOs.

It should be noted that the FIFOs in the L_A*_AC*, L_B*_AC*, and L_C*_AC* ACCESS units are used to facilitate the data movement between the memory controllers and the EXECUTE unit in such a way that processing proceeds as fast as data arrive. Therefore, once a probability is read from a FIFO, it is used by the EXECUTE unit to perform in parallel all possible arithmetic operations that use that specific value. In resource-limited devices, however, this might not be possible. In this case, the FIFO output will have to be stored into dedicated register files for a longer period of time, and reused for as many clock cycles as required in order to compute the entire output APV entry (the probabilities for all output nucleotide states).

The *PLF_EX* pipelined datapath is illustrated in Figure 9(B). It consists of arrays of double-precision floating-point multipliers to maximize arithmetic intensity. This is achieved by conducting all possible operations per received datum in parallel, and relying on sum reduction trees instead of accumulators to yield an efficient pipelined datapath with low initiation interval. Each multiplier array is composed of N multipliers, where N is the alphabet size in the MSA ($N = 4$ for DNA, $N = 20$ for proteins). Each logarithmic adder tree consists of $N - 1$ adders, yielding a critical path for calculating the sums of Equation (1) that is proportional to $log_2 N$ rather than N. The *PLF_EX* architecture can accommodate any number of discrete Γ rates for rate heterogeneity among alignment sites when per-rate data for each alignment site are consecutively provided through the two input streams.

5.2 Brick-aware Memory Layout

Application performance heavily relies on the way data are stored in memory. A common practice in high performance computing is to allocate an array within a large allocation that is performed only once, thereby increasing the likelihood that the array is allocated on contiguous physical pages [70]. In computing environments with disaggregated memory; however, a large allocation can span multiple physical memory nodes. In dReDBox, a customized system software stack allows virtual machines and orchestration software to dynamically attach remote memory to compute bricks. Memory hotplug, i.e., the resizing of memory at the OS level, is supported by an appropriately modified linux kernel for arm64 [71]. A software-defined memory controller supports the dynamic allocation/deallocation of memory resources to the host operating system running on a compute brick. Once the hardware glue logic (dReDBox IP termed Transaction Glue Logic) is configured, the kernel attaches new physical page frames to the page-table pool at run time. Subsequently, a Type-1 hypervisor is configured to dynamically expand the physical memory that is provided to the hosted virtual machine.

Since the allocated memory at the application level is transparently mapped to physical memory segments on dReDBox memory bricks, deploying BPUs to boost performance of phylogenetic analyses requires a brick-aware memory layout that eliminates remote-memory accesses. To achieve this, BPUs need to exclusively operate on local data (same memory brick), irrespective of the alignment size, the MSA data type (DNA or protein), the number of discrete Γ rates, and the tree-search

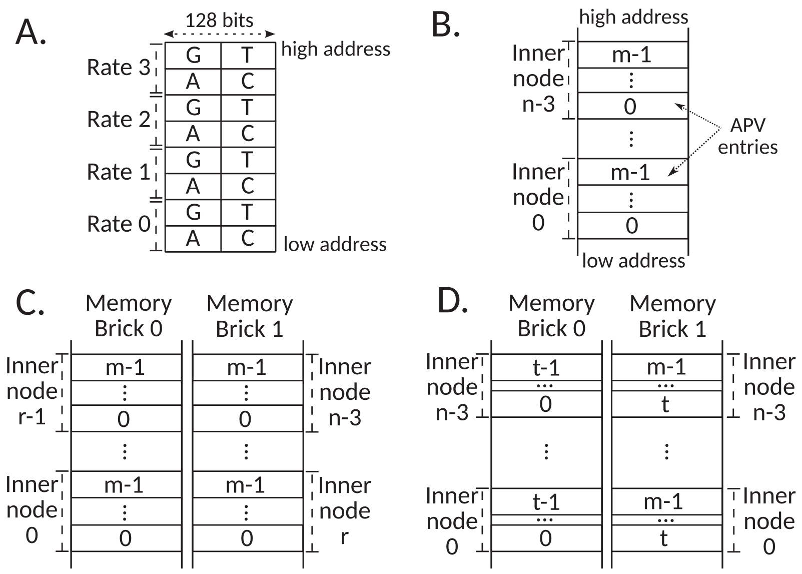

Fig. 10. (A) RAxML memory layout for an APV entry. Every access to external memory on the same brick reads/writes 128-bit words (bit-width of the memory controller bus interface). (B) Standard RAxML memory layout for $n - 2$ inner nodes and m sites. (C) Naive memory layout on two memory bricks. (D) Proposed brick-aware, interleaved memory layout.

strategy. Figure 10(A) shows the RAxML memory layout for an APV entry when processing DNA data (states A, C, G, and T) using the $GTR + \Gamma$ model with four discrete rates. It comprises 16 probability values, one per state per rate, thus occupying 128 bytes. Given an MSA of n sequences and m alignment sites, RAxML allocates a large contiguous memory space to store all $n - 2$ APVs, each comprising m entries, as depicted in Figure 10(B). A naive memory allocation will distribute APVs to memory bricks, as illustrated in Figure 10(C) for two memory bricks: one stores the first r APVs, while the other stores the remaining $n - 2 - r$ APVs. Expectedly, the possibility that all APVs involved in a single PLF invocation (two input child nodes and one output parent node) reside on the same memory brick diminishes with an increasing number of memory bricks, far less for all PLF invocations. Figure 10(D) demonstrates the proposed brick-aware memory layout that interleaves entries from all APVs on every memory brick. This distributes APV entries to memory bricks at site-level granularity: one memory brick stores the first t entries from all APVs in Figure 10, while the other stores the remaining $m - t$ entries (see also Figure 11(A) for another example).

Parallelism in computing the PLF is more efficiently exploited across sites, where computation is embarrassingly parallel, rather than across tree nodes, which imposes a sequential order of operations since a parent cannot be computed prior to its two child nodes. The proposed interleaved memory layout allows BPUs to operate in parallel and exclusively on local data, with each BPU only processing the APV entries that reside on the same brick. It also eliminates the need to relocate APVs when the topology is altered during tree searching, since each memory brick hosts corresponding fractions of all APVs.

5.3 Bulk-synchronous Parallel Processing

Processing requests to BPUs are issued sequentially while tree searching advances toward topologies with better likelihood scores. Every PLF invocation is served by all BPUs operating in parallel.

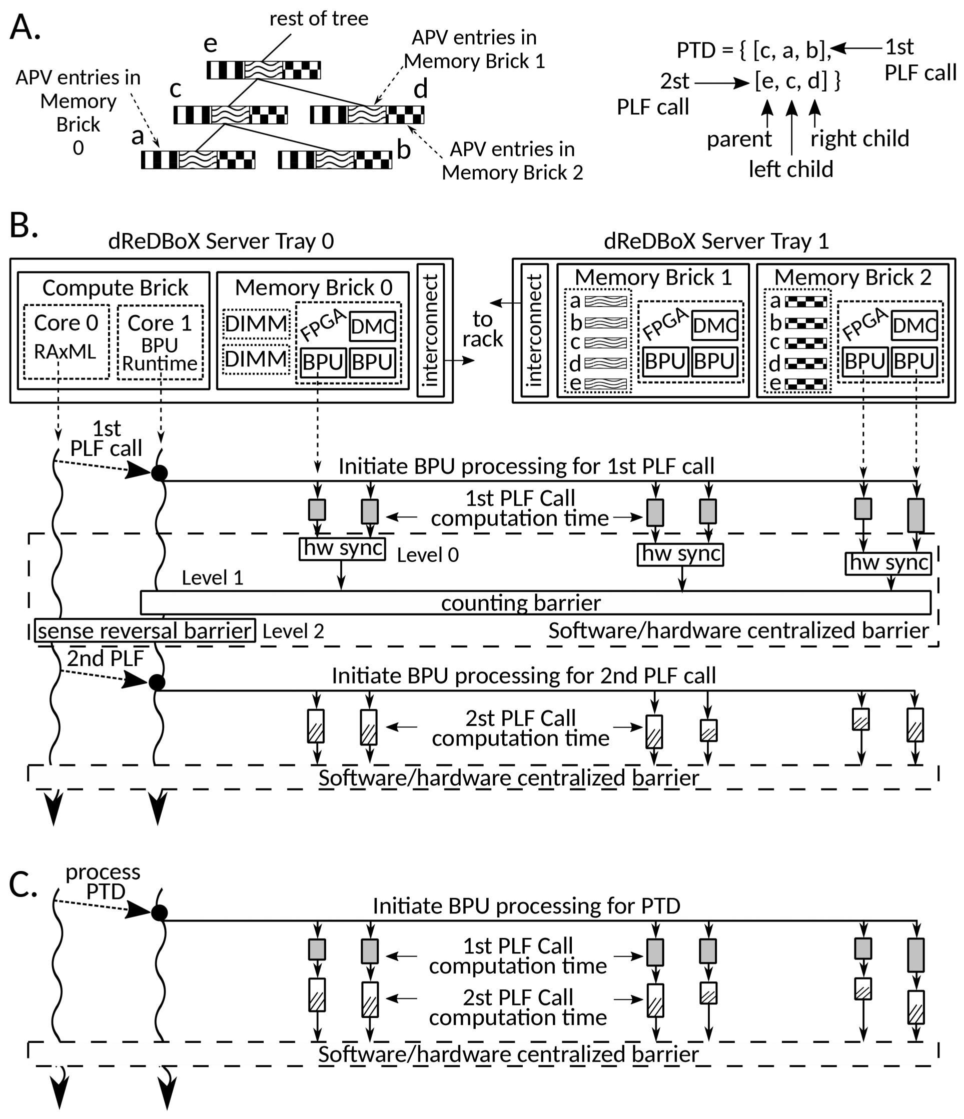

Fig. 11. (A) A 5-node subtree and a PTD with two PLF calls. (B) Two dReDBox server trays, one hosting a compute and a memory brick, and one hosting two memory bricks. Two BPUs per memory brick operate in data-parallel mode through dedicated memory ports. Synchronization is required after every PLF call, lowering the computation-to-synchronization ratio. (C) PTD-sized workload per BPU, allowing processing to proceed beyond a single PLF call within the same PTD to improve the computation-to-synchronization ratio.

Each BPU computes the fraction of the parent APV that resides on the same memory brick using the respective fractions of the children APVs, which also reside in the same physical address space. BPUs are accessing local data through physical addressing, which simplifies system design and improves performance since no virtual address translation is required. The compute brick (runs RAxML) is accessing APVs via virtual addresses, deploying the POSIX-compliant Unix system call `mmap()` to create a new mapping in the virtual address space for each remote-memory segment in every memory brick. This introduces the need for synchronization to maintain a coherent view of the shared address space between the compute brick and the BPUs on the memory bricks.

To this end, we employ a form of bulk-synchronous processing [36] in which BPUs operate independently between barriers to ensure that the application does not read stale APV entries. However, synchronization overhead increases with the number of memory bricks, while the workload per BPU decreases, which can lead to an unfavorable computation-to-synchronization ratio for the PLF. To improve this, we employ a coarse-grained model of execution that increases the workload per BPU, and employ a software/hardware barrier implementation that reduces the overhead of synchronization.

We exploit the so-called **partial traversal descriptor** (**PTD**) that is created by RAxML to invoke the PLF after every SPR iteration. A PTD contains an ordered sequence of the PLF calls that are required in order to update a number of APVs such that they reflect all the topological changes introduced by the most recent SPR iteration. In Figure 2(A), for example, after the pruning step, the virtual root is placed on the new branch that connects nodes a and c to optimize its length. After the regrafting step, the virtual root is placed on each of the two new branches. When the virtual root (vr) is placed on the branch that connects nodes a and b, the PTD consists of three entries:

```
PTD = {[a, 0, c], [b, 1, d], [vr, a, b]},
```

with each entry $T_i = [\ parent, leftChild, rigthChild\ ]$ describing a single PLF call via a triplet of node indices for the parent and the two children (see also Figure 11(A) for another example). Extending the BPU architecture to proceed beyond a single T_i before synchronizing, as demonstrated in Figure 11(C), allows to amortize the cost of synchronization by enclosing PTD-sized workloads within barriers, rather than requiring synchronization after every PLF call (Figure 11(B)). This is possible because each memory brick stores corresponding fractions of APV entries from all tree nodes, as shown in Figure 11(A). These corresponding APV entries refer to the same genomic region in the input sequences and are used together in PLF computations. Thus, once the BPU has finished processing the APV parts of entry T_i (data that reside on the same memory brick where the BPU is located), it can proceed with processing the APV parts of the entry T_{i+1} in the PTD because the combination of tree nodes to be combined next is found in the PTD, and all input data already reside on the memory brick. Because PLF computation across different genomic locations is embarrassingly parallel, this extends to computation across BPUs on different memory bricks, which is enabled by the memory layout depicted in Figure 11(A). Therefore, synchronization is only required at the end of the PTD, as illustrated in Figure 11(C).

To reduce the synchronization overhead, we devised a software/hardware centralized barrier implementation that is composed of three levels, as shown in Figure 11(B). In Level 0, BPUs on the same memory brick (each assigned a dedicated memory port) synchronize at the hardware level. A `done` flag per memory brick is raised to notify the BPU runtime. A counting barrier is employed in Level 1 to synchronize memory bricks. The runtime deploys a dedicated thread in busy-wait mode to monitor progress through an array of `done` flags. Monitoring starts prior to initiating computation on BPUs to eliminate the risk of processing finishing before the runtime is in monitoring mode. In Level 2, the runtime synchronizes with RAxML through a sense-reversal barrier to avoid potential deadlock problems that arise when sequential barriers are used.

6 MEMORY COMPRESSION

In this section, we describe a PLF-specific data interpolation engine that employs BPUs to reduce the amount of allocated physical memory in phylogenetic analyses with many taxa. Section 6.1

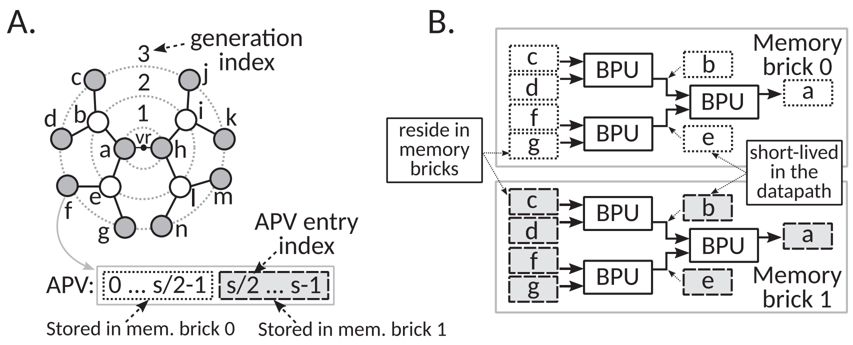

Fig. 12. (A) A fully balanced 8-taxon tree where only a subset of the inner nodes are stored in memory (light gray circles). The rest of the nodes (white circles) are recomputed when needed. (B) Two memory bricks with a 3-BPU reduction tree on each brick. Given s sites, memory brick 0 computes in the range $[0 .. s/2 - 1]$, while memory brick 1 computes in the range $[s/2 .. s - 1]$.

presents the underlying idea, while Section 6.2 introduces the required architecture extensions to create a solution that is transparent to the application running on a compute brick.

6.1 Underlying Idea

To reduce memory requirements, we exploit the observation by Izquierdo-Carrasco et al. [65] that trading memory resources for processing time is possible because a parent node can be recomputed at any point in time based on child nodes that reside in memory. The authors employed a CPU for recomputing parent vectors that are not stored in memory, expectedly observing slower processing overall (up to 40%). Our approach builds upon this observation to create an ecosystem of BPUs on disaggregated memory bricks that collectively allow to conduct large-scale phylogenetic analyses with lower memory requirements, and additionally alleviate the recomputation cost through parallel processing on deep pipelines of BPU reduction trees.

Figure 12 demonstrates the underlying idea based on an unrooted, fully balanced, 8-taxon phylogenetic tree (Figure 12(A)), and two disaggregated memory bricks hosting a 3-BPU reduction tree each (Figure 12(B)). Light gray circles represent tips or inner nodes with allocated memory space for their **ancestral probability vectors** (**APVs**), whereas blank circles indicate phylogenetic tree nodes that do not reside in memory. The dotted-line rings indicate which nodes belong to the same generation (equal distance from the virtual root in terms of number of branches). Computing the log-likelihood score for this topology requires the calculation of the APVs of nodes a and h. Before calculating the APV of node a, for instance, its child nodes need to be recomputed, since they are not stored in memory. Without a BPU-based datapath that computes across generations at our disposal (Figure 12(B)), the immediate children of node a, which are nodes b and e, need to be calculated and stored in memory. The depicted BPU-based datapaths, however, can directly compute the APV of node a, for which dedicated memory space is allocated, using the APVs of nodes c, d, f, and g, which also reside in memory, without the need to store the APVs of nodes b and e in memory at any point in time. These vectors are calculated and flow through the pipelined BPU-based reduction trees, temporarily residing in FIFOs until node a is computed. Exploiting the data allocation scheme described in Section 5.2, BPU reduction trees on distinct memory bricks can operate in parallel on different fractions of APVs that are stored locally, paving the way for a practical solution that reduces memory requirements in a time- and energy-efficient way.

An N-input processing datapath on a memory brick interconnects $N - 1$ BPU instances to construct a balanced BPU tree that can compute the APV of the common ancestor of X inner

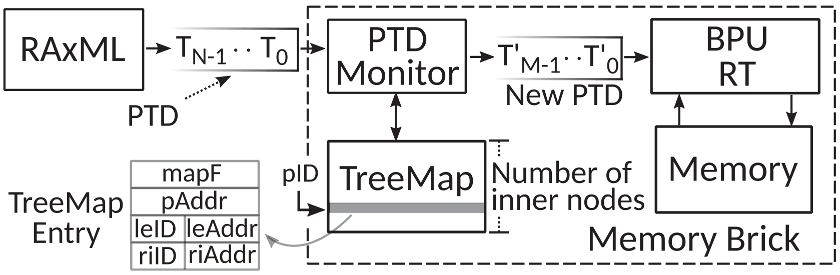

Fig. 13. Topology-aware, memory-brick architecture that hides its processing capacity from the phylogenetic software executing on a compute node. The BPU-based datapath facilitates transparent memory-side compression of PLF data.

nodes under any relationship among them with a latency of $O(\log_2 N)$, as long as the following is true: $X \leqslant N$. Given a fixed tree topology and placement of the virtual root, the BPU tree height reduced by one ($\log_2 N - 1$) indicates the number of generations that can efficiently be recomputed on the memory brick. As already mentioned; however, a practical phylogenetic analysis relies on a tree-search strategy to explore the tree space, which alters the tree topology after each step. Furthermore, optimizing the length of a branch requires to relocate the virtual root to the specific branch prior to initiating any number of optimization iterations. In the following section, we describe a set of extensions to the memory-brick architecture in order to facilitate the transparent deployment of BPU reduction trees for lowering overall memory requirements when the tree topology and the location of the virtual root change during tree searching.

6.2 Memory-brick Architecture

The primary aim at devising a transparent memory-side interpolation engine for APVs is to create a disaggregated computing system that provides the illusion that all computation is performed on processors using more memory than actually installed in the system. The phylogenetic application that runs on a compute brick and calls the PLF while searching the tree space is oblivious to the existence of processing units (BPU reduction trees) on the memory bricks. This requires that all the BPU reduction trees that are deployed on different memory bricks collectively operate as one interpolation engine that computes parent APVs using child probability vectors stored in memory. A prerequisite to allow disaggregated BPU reduction trees to operate in this manner is to devise a topology-aware memory-brick architecture that keeps track of the newly constructed evolutionary relationships after each topology-alteration step.

Our approach exploits again the RAxML PTD that is created after every SPR iteration to call the PLF (recall Section 5.3). Figure 13 illustrates the proposed memory-brick architecture that monitors the PTD and initiates computation locally, and only when required, in order to ensure that the phylogenetic software remains unaware of the memory brick's processing capacity, while at the same time a reduced amount of memory operations are issued. A dedicated memory space, denoted `TreeMap` in the figure, provides the required topology-aware functionality per memory brick. A `TreeMap` entry (`TME`) describes an inner node through the following fields: (a) the start address of its probability vector (`pAddr`), (b) the node indices of the left and right child nodes (`leID` and `riID`), (c) the start addresses of the probability vectors at the left and right child nodes (`leAddr` and `riAddr`), and (d) a 1-bit flag that indicates whether memory is allocated for the node's probability vector (`mapF`). Given S taxa, the `TreeMap` contains $S - 1$ entries to account for the $S - 2$ inner nodes (unrooted tree topology) and the virtual root. The `PTD_Monitor` parses each T_i entry of the

PTD, and updates the TreeMap accordingly, ensuring that the memory brick is aware of the tree topology at all times. For every T_i, the PTD_Monitor calculates the index p_{ID} for the TreeMap entry to update as follows:

$$p_{ID} = parentID - S,$$

where *parentID* is the node index that is retrieved from T_i.

If the mapF flag in the entry to be updated is not set, indicating no allocated memory space for the node's probability vector, the memory brick does not perform any computation, despite the fact that RAxML has issued a PLF call. In this case, only the fields that describe the child nodes are updated. RAxML practically perceives this call as a no-latency operation, since updating a TreeMap entry takes only a few clock cycles. When memory has been allocated for the *parentID* node, i.e., mapF = 1, the *PTD_Monitor* executes the recursive operation *iterativePLF*(p_{ID}) (see pseudocode for the case of a datapath with a single BPU in Algorithm 1) to infer the sequence of PLF operations that need to be conducted on the *BPU_Tree* to update the probability vector at the *parentID* node.

ALGORITHM 1: Pseudocode of the *iterativePLF* recursive function that is implemented by the *PTD_Monitor* to construct the partial traversal descriptor (*partial_PTD*) and compute the PLF on a memory brick.

```
Data: $TreeMap$ memory and $pID$ node index to compute its probability vector
Result: PTD to be processed on the memory brick
$TME_i \leftarrow TreeMap(pID)$
$leftNode \leftarrow TME_i.leID$
if $leftNode.mapF == 0$ then
    $iterativePLF(leftNode)$
end
$rightNode \leftarrow TME_i.riID$
if $rightNode.mapF == 0$ then
    $iterativePLF(rightNode)$
end
$T'_j.pAddr \leftarrow TME_i.pAddr$
$T'_j.leAddr \leftarrow TME_i.leAddr$
$T'_j.riAddr \leftarrow TME_i.riAddr$
$BPU\_Tree(T'_j)$
```

The total amount of physical memory used for storing APVs of inner nodes is a function of the desired compression ratio. The assignment of memory slots to inner nodes is arbitrary, since the tree topology changes dynamically through heuristic-based tree searching, and hence no *a-priori* assumption about inferred evolutionary relationships among inner nodes can be exploited. This raises the possibility of constructing partial traversal descriptors that comprise consecutive PLF operations for nodes without allocated memory space, the size of which is larger than the BPU_Tree depth. This requires the temporary allocation of memory slots to store these probability vectors, thereby slightly reducing the effect of compression, as will be explained in Section 8.

7 IMPLEMENTATION

The BPU harware architecture is described using high-level synthesis based on Xilinx Vivado HLS 2018.3. It is a dataflow architecture (HLS DATAFLOW) of interconnected pipelined units (ACCESS and EXECUTE) that communicate via AXI4-Stream channels (HLS INTERFACE axis). All ACCESS units are fully pipelined with initiation interval of 1 clock cycle (HLS PIPELINE II=1). The EXECUTE unit is also pipelined, with an initiation interval of 16 clock cycles (HLS PIPELINE II=16), due to the fact that an entire probability vector entry (4 discrete rates × 4 DNA states) is checked for arithmetic scaling. Recall that arithmetic scaling is required to prevent arithmetic underflow when the number of taxa increases. The BPU implements the same scaling logic used by RAxML, which scales up all 16 values in an APV entry when needed, i.e., when one probability is below a predefined threshold. While this is not necessary, and one could easily scale up only the probabilities under the threshold, scaling up the entire APV entry simplifies the arithmetic operations required for the final likelihood calculation at the virtual root. The BPU pipeline in the EXECUTE unit exploits the maximum degree of parallelism for Equation (1), calculating one probability per clock cycle, thus requiring a total of 16 clock cycles before the entire APV entry is calculated and the decision to scale up or not can be taken. The EXECUTE unit comprises multiple double-precision floating-point arithmetic units operating in parallel (`HLS ARRAY_PARTITION` and `HLS UNROLL`), achieving arithmetic intensity of 1.88 and 2.81 floating-point operations per byte when a single BPU and a 3-BPU reduction tree are used, respectively.

To test the BPU in a disaggregated-computing environment, we employed an FPGA-based emulation platform [72] formed by two ZCU102 evaluation boards interconnected over a Small Form-factor Pluggable 10-Gbps link. Each board features a Zynq Ultrascale+ MPSoC, which is the same MPSoC architecture that hosts the Transaction Glue Logic (Section 5.2) on every dReDBoX brick. In our experimental setup, one board assumes the role of a compute brick and runs Ubuntu 16.04 on its application processing unit, an ARM Cortex-A53 64-bit quad-core processor, while the other represents a memory brick that hosts memory and BPUs. The emulation platform runs the dReDBoX system software stack using the modified linux kernel for memory hotplug support [71]. We have also integrated the REMAP remote-memory manager IP [73] that enables the operating system to perceive remote-memory segments (physical memory on the memory brick) as paged memory.

We used Xilinx SDSoC 2018.3 to design and optimize the data-motion network that serves memory requests from the ACCESS units. The data-motion network provides an AXI master bus interface for prefetching the transition probability matrices and the inverted eigenvector (`ZERO_COPY` directive). For the APVs, we sequentialized the access pattern and indicated that data reside in physically contiguous memory, which, in combination with the `COPY` directive, deploys an efficient DMA engine that generates burst transfers through a streaming interface. The initiation interval (II) of the BPU pipeline is 16 clock cycles, which is defined by the APV entry size (16 double-precision floating-point values). When a BPU operates at 200 MHz, this leads to a maximum theoretical throughput performance of 12.5×10^6 APV entry updates per second, which is frequently measured in VEUPS (Vector Entry Updates Per Second). The BPU effective throughput is 12.41×10^6 VEUPS (99.2% of the theoretical peak), which indicates that the data-motion network introduces negligible overhead.

To assess BPU performance as a dedicated accelerator under a memory system with a wider interface, we used Xilinx SDAccel 2018.3 to create a BPU design point for an Amazon EC2 F1 instance, targeting the Virtex Ultrascale+ AWS-VU9P-F1 datacenter-level acceleration board that provides four 512-bit memory interfaces. We modified the ACCESS/EXECUTE units and all internal AXI4-Stream channels to map to the full data width on the memory controller, and employed three memory interfaces for the APVs. The transition probability matrices and the inverted eigenvector are prefetched through the memory interface employed for the parent APV (output). The 512-bit width allows the II to be as low as 2 clock cycles, which yields a maximum theoretical throughput of 111×10^6 VEUPS at 222 MHz. When the 512-bit BPU is deployed on a `f1.2×large` instance (one PCIe-attached FPGA), the effective throughput is 83.36×10^6 VEUPS (75.1% of the theoretical peak). Note that BPU computations are initiated through the OpenCL API.

Table 2 provides performance and resource utilization of the application-specific processing logic on the memory brick when 1–4 BPUs operate in parallel (as depicted in Figure 11(B)), and when a 3-BPU reduction tree is used (as depicted in Figure 12). The table also provides the

Table 2. Performance and Resource Utilization for Various BPU Configurations on the Zynq Ultrascale+ MPSoC: 1-4 BPUs in Parallel and a 3-BPU Reduction Tree (RT)

	1 BPU	2 BPUs	3 BPUs	4 BPUs	3-BPU RT
Freq. (MHz)	100/150/200	100/150/200	100/150	100/150	100/150
Power (W)	1.7/3/3.5	3.5/4.8/6.1	5.1/6.9	6.7/7.5	4.1/5.9
II (cycles)	16	16	16	16	16
BW (GB/s)	2.4/3.6/4.8	4.8/7.2/9.6	7.2/10.8	9.6/14.4	4/6
LUTs (274,080)	16.13%–17.57%	33.06%–36.05%	48.31%–51.61%	63.34%–67.69%	44.82%–48.17%
FFs (548,160)	9.79%–12.73%	26.52%–32.40%	37.83%–42.45%	48.74%–54.89%	28.56%–33.18%
DSPs (2,520)	7.18%	14.37%	21.55%	28.73%	21.55%
BRAMs (912)	3.95%	6.74%	9.05%	10.86%	9.92%

The bandwidth values (“BW”) refer to the maximum memory bandwidth that can be utilized.

Table 3. Performance and Resource Utilization for the REMAP Inter-brick Communication Controller [73] and two Memory-brick Configurations with 4 BPUs in Parallel and a 3-BPU Reduction tree (RT) on the Zynq Ultrascale+ MPSoC

	REMAP IP [73] on		Memory Brick with	
	Compute Brick	Memory Brick	4 BPUs	3-BPU RT
Frequency (MHz)	100	100	100	150
Power (W)	5.0	5.6	8.1	8.4
II (cycles)	n/a		16	16
BW (GB/s)	1.2 (between nodes)		9.5	5.9
LUTs (274,080)	6.13%	8.03%	69.71%	55.39%
FFs (548,160)	4.93%	6.68%	53.77%	38.91%
DSPs (2,520)	0.12%	0.12%	28.85%	21.67%
BRAMs (912)	21.27%	13.76%	24.62%	23.68%

The bandwidth values (“BW”) refer to the maximum memory bandwidth that can be utilized.

maximum memory bandwidth that the BPUs can collectively utilize on the memory node, as well as the peak BPU-induced dynamic power overhead, which was measured using a digital power meter. Note that Xilinx SDSoC only allows to choose from a pre-defined set of target clock frequencies for hardware generation. When 3 or 4 BPUs were placed on the memory brick, increased routing prevented the successful generation of hardware that could be clocked at 200 MHz. In this case, the next highest target clock frequency setting was used, i.e., 150 MHz. Table 3 provides performance and resource utilization for the REMAP [72] IP blocks on the compute brick and the memory brick, as well as the total amount of occupied resources on the memory brick when 4 BPUs operate in parallel and when a 3-BPU reduction tree is used. Table 4 provides performance and resource utilization of two possible configurations of the accelerator brick, one implemented on the Zynq Ultrascale+ MPSoC (ZCU102) and one implemented on the AWS-VU9P-F1 (henceforth denoted EC2-F1).

8 PERFORMANCE EVALUATION

8.1 Brick Performance Scaling and Time Breakdown

To assess datacenter-scale performance, we devise a hybrid evaluation methodology that combines native execution on our emulation platform (Section 7) and EC2-F1 instances, with simulation of the behavior of multiple accelerator/memory bricks for different inter-brick bandwidth

Table 4. Performance and Resource Utilization of Accelerator Brick Configurations on ZCU102 (Same Platform as the Compute and the Memory Bricks) and EC2-F1 (f1.2×large)

	Accelerator Brick	
	ZCU102	EC2-F1
Frequency (MHz)	150	222
Power (Watts)	8.1	6.367
II (clock c.)	16	2
APV-Entries/c	0.25	0.5
LUTs (%)	67.69	10.33
FFs (%)	54.89	10.10
DSPs (%)	21.55	21.20
BRAMs (%)	9.05	1.65

configurations. Figure 14(A) illustrates aggregate throughput performance for up to 512 bricks as a percentage of the theoretical peak of the total amount of deployed custom hardware, while Figure 15 provides a time breakdown for the same accelerator-/memory-brick configurations for a fixed bandwidth. The inter-node bandwidth configurations of 4.2 Gbps, 9.1 Gbps, 19.6 Gbps, and 40 Gbps refer to our in-house emulation platform, the REMAP disaggregated-memory controller by Theodoropoulos et al. [73], the Marlin RDMA-based communication primitives by Cheng-Chun et al. [74], and Microsoft's Configurable Cloud architecture by Caulfield et al. [33], respectively.

As shown in Figure 14(A) and (B), user-level accelerator-brick performance (observed by the compute brick) is at most 43.9% and 9.4% of the theoretical peak on the ZCU102 and the EC2-F1, respectively, due to the required inter-brick data transfers (Figure 15(A) and (B)). The proposed bulk-synchronous parallel processing model and brick-aware memory layout are also employed for execution on accelerator bricks, with every accelerator brick communicating with a dedicated memory brick in full-duplex mode while amortizing the data-transfer time for two of the three APVs (one input and one output) through overlapping with processing. In every PLF call, the corresponding parts of two input APVs need to be transferred from a memory brick to an accelerator brick, and the computed part of the output APV is transferred back to a memory brick. Recall that every APV entry is 128 bytes (Section 5.2) and every accelerator brick receives, calculates, and transfers back a large number of APV entries per PLF call, depending on the sequence length and the number of accelerator bricks. This leads to inter-brick data transfers limiting the effective BPU throughput on accelerator bricks irrespective of the number accelerator bricks, as can be observed in Figure 14(A) and (B).

Figure 14(C)–(E) show near-peak BPU performance in three different memory-brick configurations, unburdened by inter-brick data transfers, and limited only by the synchronization overhead that increases with the number of bricks, as shown in Figure 15(C)–(E), respectively. Furthermore, it can be observed that memory-brick configurations with more than one BPU scale better, as they benefit the most from hardware-level synchronization (Level 0 in Figure 11(B)). For instance, a total of 512 BPUs in a 1-BPU-per-brick configuration (Figure 14(C)) achieve 53.5% of the theoretical peak, whereas the same number of BPUs deployed in pairs (Figure 14(D), 256 bricks) or in groups of four (Figure 14(E), 128 bricks) deliver 62.9% and 65.9%, respectively.

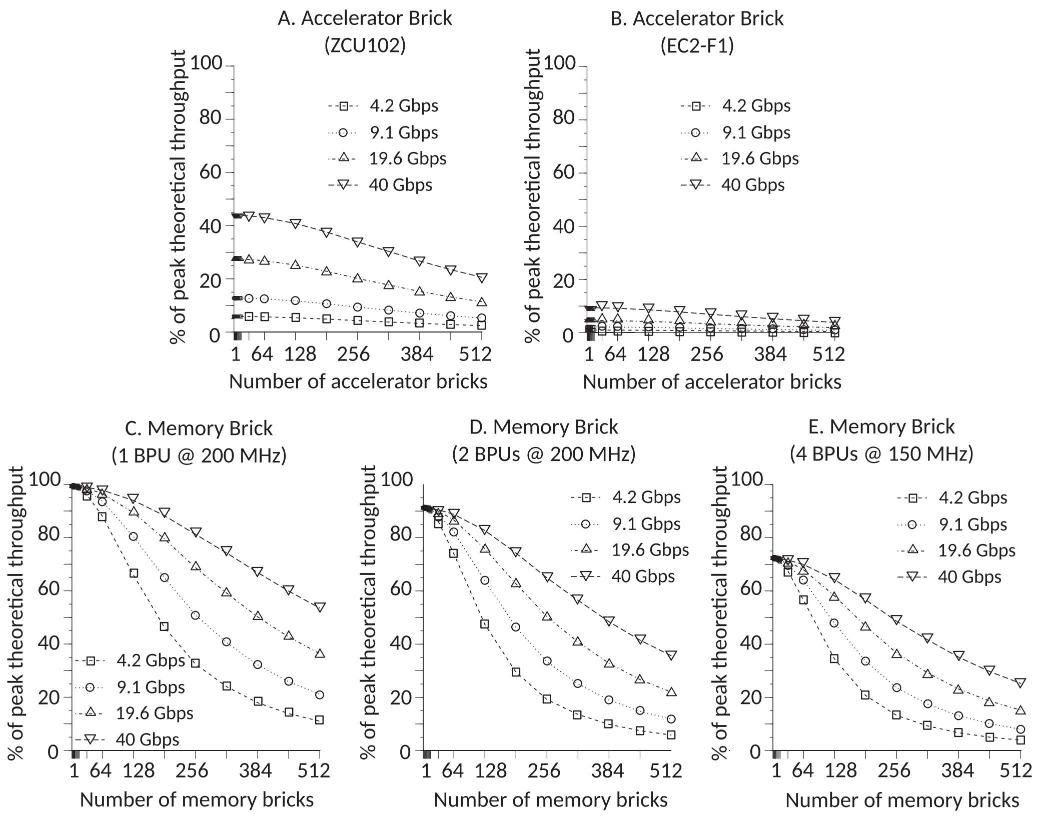

Fig. 14. Accelerator/memory brick performance as percentage of the theoretical peak (based on the total amount of deployed custom hardware) for different inter-brick bandwidth configurations.

8.2 Brick Performance and Energy Efficiency Comparison

Figure 16(A) provides a FLOPS comparison between the EC2-F1-based accelerator brick and the three ZCU102-based memory-brick configurations. When the inter-brick bandwidth is 4.2 Gbps (emulation platform), the best-performing memory-brick configuration (4BPUs, 150MHz) is up to 12.9× faster than the accelerator brick, delivering up to 792.8 GFLOPS (512 BPUs, 128 memory bricks). With an inter-brick bandwidth of 40 Gbps, BPU processing on memory bricks is up to 67.2% faster than the accelerator brick, delivering up to 2,426.4 GFLOPS (1,526 BPUs, 384 memory bricks).

Figure 16(B) demonstrates the effect of PTD (Partial Traversal Descriptor) size on the aggregate throughput of memory and accelerator bricks. The PTD size improves the computation-to-synchronization ratio and allows performance to scale better with the number of bricks by alleviating the effect of synchronization. Using RAxML and simulated datasets with up to 1,000 taxa, we find that, while PTD sizes vary between 1 and $n - 1$ PLF calls per run, where n is the number of taxa, over 90% of the PTDs comprise between 2 and 8 PLF calls. The average-case scenarios of PTDs with 5 and 10 PLF calls, depicted in Figure 16(B), reveal 3.3× and 4.6× higher memory-brick performance over the single-PLF-per-PTD case, respectively, when 1,024 bricks are deployed (the respective improvement for the accelerator brick is 3.1× and 4.2×).

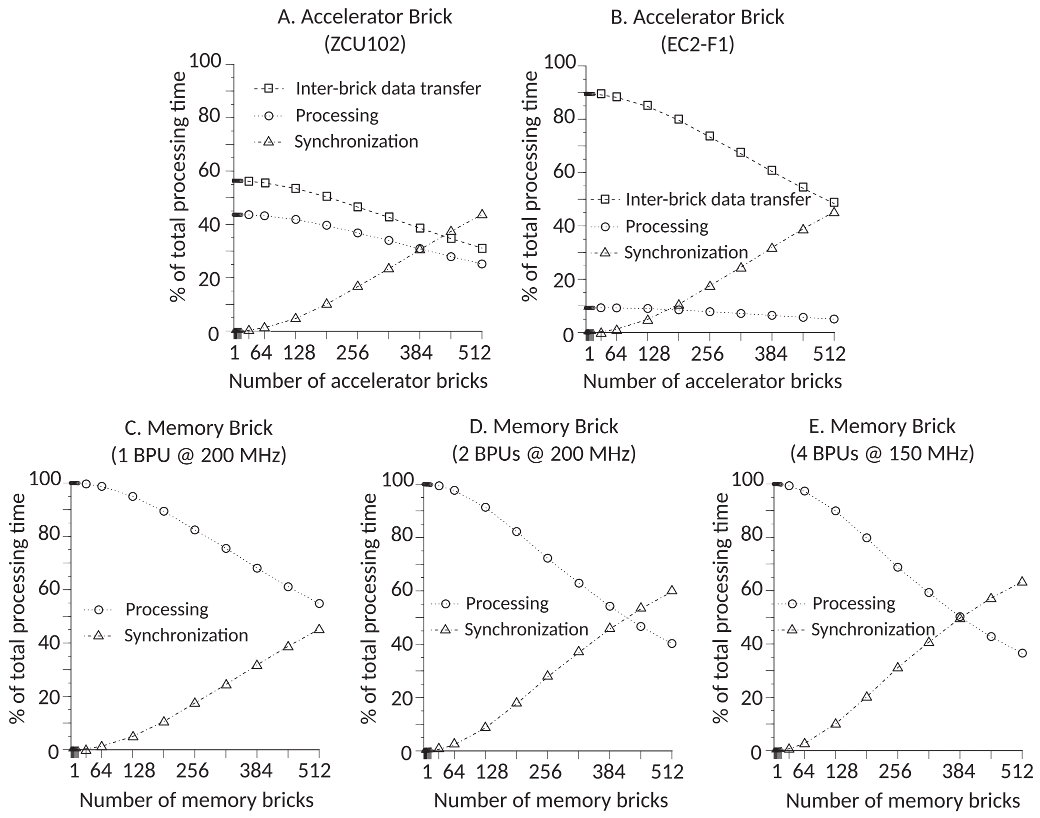

Fig. 15. Accelerator/memory brick time breakdown for the 40-Gbps bandwidth configuration. Processing includes local-data access times.

Figure 16(C) and (D) provides comparisons of energy efficiency (FLOPS/Watt) between the ZCU102-based accelerator brick and the three memory-brick configurations for 4.2 Gbps and 40 Gbps inter-brick bandwidth configurations, respectively. Remote-data transfers reduce energy efficiency by up to an order of magnitude, with the most energy-efficient memory-brick configuration (2 BPUs, 200 MHz) achieving 1.794 GFLOPS/Watt (irrespective of inter-brick bandwidth) in comparison with 116.991 MFLOPS/Watt and 878.445 MFLOPS/Watt achieved by the accelerator brick under 4.2 Gbps and 40 Gbps bandwidth configurations, respectively. The synchronization overhead increases with the number of memory/accelerator bricks, which prevents FLOPS performance from scaling, thereby diminishing the energy-efficiency gap when the number of bricks increases to 512 or higher.

Overall, Figure 16 shows that memory-brick configurations with one or more BPUs achieve higher throughput performance and energy efficiency than more powerful accelerator-brick configurations. This is because of the limited number of inter-brick data transfers that are required when BPUs are deployed on memory bricks. The memory-aware memory layout allows all BPUs in a memory brick to operate on local data on the same memory brick, thereby considerably reducing the time and energy spent on inter-brick data transfers. Accelerator-brick configurations can not avoid this time/energy overhead per PLF call, achieving overall lower throughput and energy efficiency than memory-brick configurations.

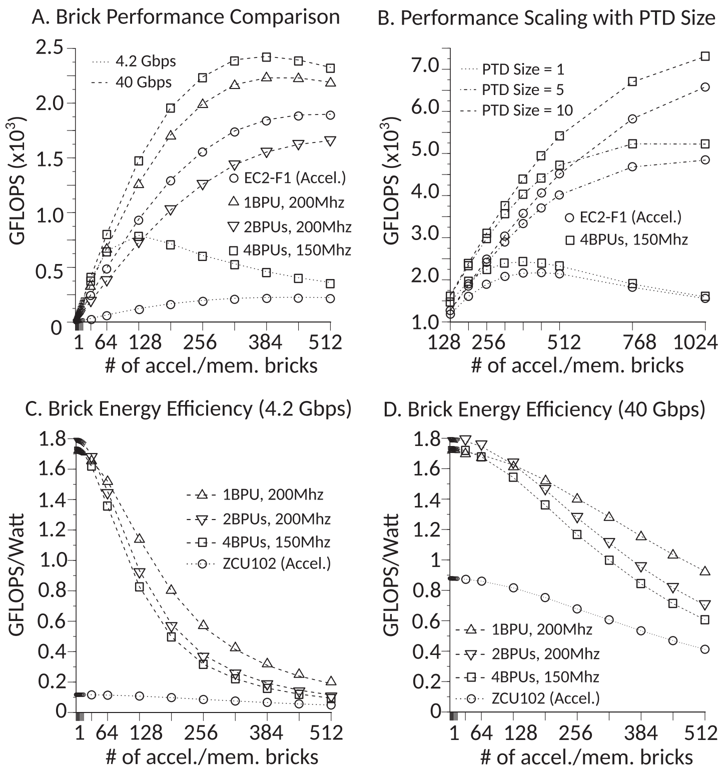

Fig. 16. (A) FLOPS performance comparison. (B) Performance scaling with the PTD size (number of PLF calls). (C) Energy efficiency under 4.2 Gbps inter-brick bandwidth. (D) Energy efficiency under 40 Gbps inter-brick bandwidth.

8.3 Memory-side Compression/Decompression

To evaluate the proposed compression/decompression scheme, we execute the standard RAxML version 8.2.12 [17] on the compute brick, while the memory brick hosts the 3-BPU reduction tree with topology-aware functionality (Figure 13). RAxML was used to perform full phylogenetic analyses of two simulated datasets with a different number of DNA sequences (50 and 100) and the same alignment length (1,000 sites). Figure 17 shows the total execution time required per dataset to compute the PLF using the 3-BPU datapath on the memory brick when the compression ratio increases up to 2.5 (with respect to the RAxML full memory requirements per dataset). The figure also provides the respective execution time when the same compression scheme is implemented in a software-only environment on a Dell PowerEdge R530 rack server with two 10-core Intel Xeon E5-2630v4 CPUs (20 threads per CPU) running at 2.2 GHz (base), and 128 GB of DDR4 main memory. We employ the fastest parallel implementation of RAxML, which uses Posix threads and Advanced Vector Extensions 2 (AVX2), for these measurements. In addition, the figure illustrates the reduction in energy consumption (note the secondary vertical axis) that is achieved by recomputing ancestral probability vectors on an as-needed basis on the memory node, instead of using 1 and 2 CPU cores. Power consumption of the Dell server was monitored using the **Dell Remote Access Controller** (**iDRAC**), i.e., a controller card embedded in the motherboard.

As can be observed in Figure 17(A) for the 50-taxon phylogenetic analysis, trading computation time for memory introduces a relatively low recomputation overhead to the CPU-only

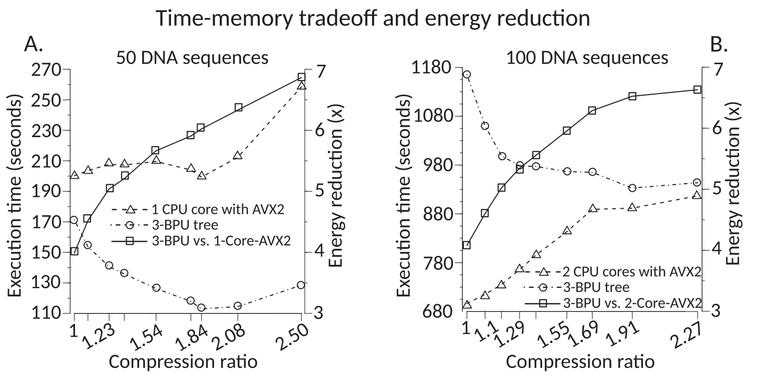

Fig. 17. Time-memory trade-off for the phylogenetic analysis of simulated datasets with 50 (A) and 100 (B) sequences (length 1,000 sites), as the compression ratio increases to 2.5. The 50-taxon software-only analysis employed 1 CPU core, whereas the respective 100-taxon one utilized 2 CPU cores.

implementation (less than 5%) when the compression ratio remains under 1.84, whereas for larger compression ratios, the recomputation overhead approaches 30%. When the ancestral probability vectors are recomputed on the memory node; however, we achieve improved performance, with up to 34% shorter execution times, due to the effective utilization of the deeper application-specific pipeline that the interconnected BPUs form on the memory node. Based on the power overhead that the 3-BPU tree adds to the disaggregated memory node itself (4.1 W, operating at 100 MHz),[4] and the respective power overhead that a CPU core using AVX2 instructions adds to the rack server (14 W),[5] we estimate that the proposed memory-side compression/decompression scheme achieves up to 6.9 times lower energy consumption for the computation of the PLF, with the CPU core and the 3-BPU tree consuming a total of 3.619 kJ and 0.527 kJ when the compression ratio is 2.5, respectively. Similar energy savings are achieved when the phylogeny size increases, as can be observed in Figure 17(B) for the 100-taxon analysis. Furthermore, the figure reveals that, when the compression ratio is 2.27, a disaggregated memory node with a 3-BPU tree exhibits comparable performance with 2 CPU cores using AVX2 (the memory node requires less than 3% longer execution time), while achieving 6.6 times lower energy consumption, with the 2 CPU cores and the 3-BPU tree now consuming 25.674 kJ and 3.869 kJ, respectively.

8.4 Comparison with Software and other Accelerators

Table 5 compares the proposed BPU architecture on the ZCU102 and the EC2-F1 with RAxML [17] and previous FPGA accelerators [24, 45, 60] in terms of VEUPS and VEUPS/Watt. We employ a Dell PowerEdge R530 rack server with two 10-core Intel Xeon E5-2630v4 CPUs running at 2.2 GHz as the test platform. We deploy the fastest RAxML implementation, which uses Posix threads and AVX2 extensions, while setting CPU affinity per thread (RAxML-provided option for performance). Power consumption of the Dell server was monitored using the iDRAC, a controller card embedded in the motherboard.[6]

As can be observed in the table, two BPUs on a ZCU102 achieve approximately the same throughput performance as the fastest RAxML execution on a ×86 core, and more than two times higher

[4]The static power consumption of the ZCU102 board before programming is 22.6 W.
[5]The iDRAC reports power consumption of 112 W for the rack server when unloaded.
[6]The iDRAC reports 112 W power consumption when the server is unloaded. The ZCU102 static power consumption before programming is 22.6 W.

Table 5. Comparison of Throughput (VEUPS) and Energy Efficiency (VEUPS/Watt)

Implementation	Information	VEUPS×10^6	VEUPS×10^6/Watt
RAxML-AVX2 (PowerEdge R530)	1 thread	24.054	1.718
	4 threads	89.132	2.122
	20 threads	329.582	2.746
Alachiotis et al. [24]	284 MHz	17.750	n/a
Alachiotis et al. [45]	101 MHz	25.250	n/a
Berger et al. [60]	167 MHz	18.385	n/a
1 BPU (ZCU102)	200 MHz	12.414	3.547
2 BPUs (ZCU102)	200 MHz	24.765	4.059
4 BPUs (ZCU102)	150 MHz	27.124	3.617
1 BPU (EC2-F1)	222 MHz	83.362	13.093

The BPUs operate on local data for this comparison. Designs [24, 45], [60] are mapped on Virtex5.

energy efficiency. While throughput performance of the PowerEdge server increases with the number of threads, energy efficiency does not improve significantly. The fastest BPU configuration in this study, on an EC2-F1 instance, achieves nearly the same throughput performance as four ×86 cores and 6.2× higher energy efficiency, delivering up to 13×10^6 VEUPS per Watt. Therefore, a major advantage of the BPU architecture over multi-core processors, irrespective of whether it is deployed in a near-memory configuration or as a dedicated accelerator, is the higher energy efficiency.

Throughput comparisons with other hardware accelerators show that one BPU does not outperform previous architectures. However, such comparisons should take into account the practical limitations introduced in each design. The first accelerator architecture by Alachiotis et al. [24], for instance, could only process fully balanced phylogenetic trees, thereby limiting its applicability in real-world studies. Alachiotis et al. [45] addressed that limitation but did not support scaling, thereby restricting its application to phylogenetic trees with a small number of sequences. Berger et al. [60] presented an optimized accelerator for any tree topology and number of sequences, but, similarly to the previous accelerator architectures [24, 45], the architecture does not accommodate statistical models for rate heterogeneity. All aforementioned limitations are addressed in the BPU architecture, and the support of rate heterogeneity with four discrete Γ rates, as required by RAxML, leads to the BPU performing at least four times more operations per alignment site than the previous accelerator architectures in this comparison.

9 CONCLUSIONS

This work presented a custom architecture for a fundamental computational kernel in phylogenetics, and described an efficient memory layout that eliminates remote-data transfers on datacenters with disaggregated memory. We devised a bulk-synchronous parallel model of execution with a favorable computation-to-synchronization ratio, observing an order of magnitude better performance and energy efficiency when computing on local data instead of conducting explicit data transfers between remote accelerator and memory resources. We found that performance and power efficiency improves by an order of magnitude when computing on local data instead of conducting explicit data transfers between disaggregated compute and memory resources. Moreover, we described an effective way to deploy interconnected BPUs that operate similarly to a data interpolation engine transparently to the user application, leading to further performance and power efficiency improvements, as well as up to 2.5 times lower memory requirements, thereby paving the way for reconstructing even larger phylogenies.

As future work, we intend to explore the potential of deploying 3D-stacked DRAMs on disaggregated memory nodes, and assess performance and energy gains from deploying the proposed application-specific accelerator architecture directly on the logic-in-memory layer. Furthermore, we intend to adapt the proposed custom logic for protein and RNA data. In addition, we intend to devise near-memory processing solutions for more scientific-computing kernels that are bound by memory access.

REFERENCES

[1] C. S. Baker and S. R. Palumbi. 1994. Which whales are hunted? A molecular genetic approach to monitoring whaling. *Science* 265, 5178 (1994), 1538–1540.

[2] Logan Volkmann, Iain Martyn, Vincent Moulton, Andreas Spillner, and Arne O. Mooers. 2014. Prioritizing populations for conservation using phylogenetic networks. *PLoS One* 9, 2 (2014), e88945.

[3] Jonathan Rolland, Marc W. Cadotte, Jonathan Davies, Vincent Devictor, Sebastien Lavergne, Nicolas Mouquet, Sandrine Pavoine, Ana Rodrigues, Wilfried Thuiller, Laure Turcati, et al. 2011. Using phylogenies in conservation: New perspectives. *The Royal Society* (2011).

[4] Robin M. Bush, Catherine A. Bender, Kanta Subbarao, Nancy J. Cox, and Walter M. Fitch. 1999. Predicting the evolution of human influenza A. *Science* 286, 5446 (1999), 1921–1925.

[5] Tommy Tsan-Yuk Lam, Chung-Chau Hon, and Julian W. Tang. 2010. Use of phylogenetics in the molecular epidemiology and evolutionary studies of viral infections. *Critical Reviews in Clinical Laboratory Sciences* 47, 1 (2010), 5–49.

[6] Ana B. Abecasis, Marta Pingarilho, and Anne-Mieke Vandamme. 2018. Phylogenetic analysis as a forensic tool in HIV transmission investigations. *Aids* 32, 5 (2018), 543–554.

[7] Nina Rønsted, Vincent Savolainen, Per Mølgaard, and Anna K. Jäger. 2008. Phylogenetic selection of narcissus species for drug discovery. *Biochemical Systematics and Ecology* 36, 5–6 (2008), 417–422.

[8] Alexander Dimitri Yermanos, Andreas Kevin Dounas, Tanja Stadler, Annette Oxenius, and Sai T. Reddy. 2018. Tracing antibody repertoire evolution by systems phylogeny. *Frontiers in Immunology* 9 (2018), 2149. https://doi.org/10.3389/fimmu.2018.02149

[9] Alexandros Stamatakis. 2005. Phylogenetics: Applications, software and challenges. *Cancer Genomics-Proteomics* 2, 5 (2005), 301–305.

[10] David A. Bader, Bernard M. E. Moret, and Lisa Vawter. 2001. Industrial applications of high-performance computing for phylogeny reconstruction. In *Proceedings of the Commercial Applications for High-Performance Computing*, Vol. 4528. International Society for Optics and Photonics, 159–169.

[11] Naruya Saitou and Masatoshi Nei. 1987. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution* 4, 4 (1987), 406–425.

[12] Robert R. Sokal. 1958. A statistical method for evaluating systematic relationship. *University of Kansas Science Bulletin* 38 (1958), 1409–1438.

[13] Marketa J. Zvelebil and Jeremy O. Baum. 2007. *Understanding Bioinformatics*. Garland Science.

[14] Joseph Felsenstein. 1981. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution* 17, 6 (1981), 368–376.

[15] Alexei J. Drummond and Andrew Rambaut. 2007. BEAST: Bayesian evolutionary analysis by sampling trees. *BMC Evolutionary Biology* 7, 1 (2007), 214.

[16] Morgan N. Price, Paramvir S. Dehal, and Adam P. Arkin. 2010. FastTree 2–approximately maximum-likelihood trees for large alignments. *PloS One* 5, 3 (2010), e9490.

[17] Alexandros Stamatakis. 2014. RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics* 30, 9 (2014), 1312–1313.

[18] Fredrik Ronquist, Maxim Teslenko, Paul Van Der Mark, Daniel L. Ayres, Aaron Darling, Sebastian Höhna, Bret Larget, Liang Liu, Marc A. Suchard, and John P. Huelsenbeck. 2012. MrBayes 3.2: Efficient bayesian phylogenetic inference and model choice across a large model space. *Systematic Biology* 61, 3 (2012), 539–542.

[19] Fernando Izquierdo-Carrasco, Nikolaos Alachiotis, Simon Berger, Tomas Flouri, Solon P. Pissis, and Alexandros Stamatakis. 2013. A generic vectorization scheme and a GPU kernel for the phylogenetic likelihood library. In *Proceedings of the 2013 IEEE Int. Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 530–538.

[20] Alexandros Stamatakis and Michael Ott. 2008. Efficient computation of the phylogenetic likelihood function on multigene alignments and multi-core architectures. *Philosophical Transactions of the Royal Society B: Biological Sciences* 363, 1512 (2008), 3977–3984.

[21] Frederico Pratas, Pedro Trancoso, Alexandros Stamatakis, and Leonel Sousa. 2009. Fine-grain parallelism using multi-core, cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function. In *Proceedings of the 2009 International Conference on Parallel Processing*. IEEE, 9–17.

[22] Michael Ott, Jaroslaw Zola, Alexandros Stamatakis, and Srinivas Aluru. 2007. Large-scale maximum likelihood-based phylogenetic analysis on the IBM BlueGene/L. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. ACM, 4.

[23] Jun Chai, Huayou Su, Mei Wen, Xing Cai, Nan Wu, and Chunyuan Zhang. 2013. Resource-efficient utilization of CPU/GPU-based heterogeneous supercomputers for bayesian phylogenetic inference. *The Journal of Supercomputing* 66, 1 (2013), 364–380.

[24] Nikolaos Alachiotis, Euripides Sotiriades, Apostolos Dollas, and Alexandros Stamatakis. 2009. Exploring FPGAs for accelerating the phylogenetic likelihood function. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[25] Stephanie Zierke and Jason D. Bakos. 2010. FPGA acceleration of the phylogenetic likelihood function for bayesian MCMC inference methods. *BMC Bioinformatic* 11, 1 (2010), 184.

[26] Lidia Kuan, Joao Neves, Frederico Pratas, Pedro Tomás, and Leonel Sousa. 2014. Accelerating phylogenetic inference on GPUs: An OpenACC and CUDA comparison.. In *Proceedings of the IWBBIO*. 589–600.

[27] Pei Liu, Fatemeh O. Ebrahim, Ahmed Hemani, and Kolin Paul. 2011. A coarse-grained reconfigurable processor for sequencing and phylogenetic algorithms in bioinformatics. In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 190–197.

[28] Pei Liu, Ahmed Hemani, and Kolin Paul. 2011. A reconfigurable processor for phylogenetic inference. In *Proceedings of the 2011 24th International Conference on VLSI Design*. IEEE, 226–231.

[29] Turbo Majumder, Michael Edward Borgens, Partha Pratim Pande, and Ananth Kalyanaraman. 2012. On-chip network-enabled multicore platforms targeting maximum likelihood phylogeny reconstruction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 7 (2012), 1061–1073.

[30] Turbo Majumder, Partha Pande, and Ananth Kalyanaraman. 2011. Accelerating maximum likelihood based phylogenetic kernels using network-on-chip. In *Proceedings of the 2011 23rd International Symposium on Computer Architecture and High Performance Computing*. IEEE, 17–24.

[31] Amy C. Driskell, Cécile Ané, J. Gordon Burleigh, Michelle M. McMahon, Brian C. O'meara, and Michael J. Sanderson. 2004. Prospects for building the tree of life from large sequence databases. *Science* 306, 5699 (2004), 1172–1174.

[32] Jason A. Reuter, Damek V. Spacek, and Michael P. Snyder. 2015. High-throughput sequencing technologies. *Molecular Cell* 58, 4 (2015), 586–597.

[33] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[34] Maciej Bielski, Ilias Syrigos, Kostas Katrinis, Dimitris Syrivelis, Andrea Reale, Dimitris Theodoropoulos, Nikolaos Alachiotis, Dionisios Pnevmatikatos, EH Pap, George Zervas, et al. 2018. dReDBox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1093–1098.

[35] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. 2020. ThymesisFlow: A software-defined, HW/SW co-designed interconnect stack for rack-scale memory disaggregation. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 868–880.

[36] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8 (1990), 103–111.

[37] Nikolaos Alachiotis, Andreas Andronikakis, Orion Papadakis, Dimitris Theodoropoulos, Dionisios Pnevmatikatos, Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, George Zervas, Vaibhawa Mishra, et al. 2019. dReDBox: A disaggregated architectural perspective for data centers. In *Proceedings of the Hardware Accelerators in Data Centers*. Springer, 35–56.

[38] Quan Zou, Shixiang Wan, Xiangxiang Zeng, and Zhanshan Sam Ma. 2017. Reconstructing evolutionary trees in parallel for massive sequences. *BMC Systems Biology* 11, 6 (2017), 100.

[39] Robert C. Edgar. 2004. MUSCLE: Multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research* 32, 5 (2004), 1792–1797.

[40] Kazutaka Katoh and Daron M. Standley. 2013. MAFFT multiple sequence alignment software version 7: Improvements in performance and usability. *Molecular Biology and Evolution* 30, 4 (2013), 772–780.

[41] Mark A. Larkin, Gordon Blackshields, Nigel P. Brown, R. Chenna, Paul A. McGettigan, Hamish McWilliam, Franck Valentin, Iain M. Wallace, Andreas Wilm, Rodrigo Lopez, et al. 2007. Clustal w and clustal x version 2.0. *Bioinformatics* 23, 21 (2007), 2947–2948.

[42] Ming Li and Louxin Zhang. 1999. Twist–rotation transformations of binary trees and arithmetic expressions. *Journal of Algorithms* 32, 2 (1999), 155–166.

[43] Magnus Bordewich and Charles Semple. 2005. On the computational complexity of the rooted subtree prune and regraft distance. *Annals of Combinatorics* 8, 4 (2005), 409–423.

[44] Benjamin L. Allen and Mike Steel. 2001. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics* 5, 1 (2001), 1–15.

[45] Nikolaos Alachiotis, Alexandros Stamatakis, Euripides Sotiriades, and Apostolos Dollas. 2009. A reconfigurable architecture for the phylogenetic likelihood function. In *Proceedings of the 2009 International Conference on Field Programmable Logic and Applications*. IEEE, 674–678.

[46] Simon Tavaré. 1986. Some probabilistic and statistical problems in the analysis of DNA sequences. *Lectures on Mathematics in the Life Sciences* 17, 2 (1986), 57–86.

[47] Ziheng Yang. 1994. Maximum likelihood phylogenetic estimation from DNA sequences with variable rates over sites: Approximate methods. *Journal of Molecular Evolution* 39, 3 (1994), 306–314.

[48] Kostas Katrinis, Dimitris Syrivelis, Dionisios Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K. Hasharoni, Daniel Raho, Christian Pinto, F. Espina, et al. 2016. Rack-scale disaggregated cloud data centers: The dReDBox project vision. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 690–695.

[49] Dimitris Syrivelis, Andrea Reale, Kostas Katrinis, Ilias Syrigos, Maciej Bielski, Dimitris Theodoropoulos, Dionisios N. Pnevmatikatos, and Georgios Zervas. 2017. A software-defined architecture and prototype for disaggregated memory rack scale systems. In *Proceedings of the 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. IEEE, 300–307.

[50] Seth H. Pugsley, Jeffrey Jestes, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro* 34, 4 (2014), 44–52. DOI: http://dx.doi.org/10.1109/MM.2014.54

[51] Panagiotis Skrimponis, Emmanouil Pissadakis, Nikolaos Alachiotis, and Dionisios Pnevmatikatos. 2020. Accelerating binarized convolutional neural networks with dynamic partial reconfiguration on disaggregated FPGAs. In *Proceedings of the Parallel Computing: Technology Trends*. IOS Press, 691–700.

[52] Emmanouil Pissadakis, Nikolaos Alachiotis, Panagiotis Skrimponis, Dimitris Theodoropoulos, Thanasis Korakis, and Dionisios Pnevmatikatos. 2018. ReFiRe: Efficient deployment of remote fine-grained reconfigurable accelerators. In *Proceedings of the 2018 International Conference on Field-Programmable Technology*. 322–325. DOI: http://dx.doi.org/10.1109/FPT.2018.00064

[53] Nikolaos Alachiotis, Panagiotis Skrimponis, Manolis Pissadakis, Sundeep Rangan, and Dionisios Pnevmatikatos. 2020. Near-memory acceleration for scalable phylogenetic inference. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, 324. DOI: http://dx.doi.org/10.1145/3373087.3375364

[54] Terrence S. T. Mak and Kai-Pui Lam. 2003. High speed GAML-based phylogenetic tree reconstruction using HW/SW codesign. In *Proceedings of the 2003 IEEE Bioinformatics Conference on Computational Systems Bioinformatics*. IEEE, 470–473.

[55] Paul O. Lewis. 1998. A genetic algorithm for maximum-likelihood phylogeny inference using nucleotide sequence data. *Molecular Biology and Evolution* 15, 3 (1998), 277–283.

[56] Thomas H. Jukes and Charles R. Cantor. 1969. Evolution of protein molecules. *Mammalian Protein Metabolism* 3, 21 (1969), 132.

[57] Jennifer Ripplinger and Jack Sullivan. 2008. Does choice in model selection affect maximum likelihood analysis? *Systematic Biology* 57, 1 (2008), 76–85.

[58] Terrence S. T. Mak and Kai-Pui Lam. 2004. Embedded computation of maximum-likelihood phylogeny inference using platform FPGA. In *Proceedings. 2004 IEEE Computational Systems Bioinformatics Conference, 2004.* IEEE, 512–514.

[59] Fredrik Ronquist and John P. Huelsenbeck. 2003. MrBayes 3: Bayesian phylogenetic inference under mixed models. *Bioinformatics* 19, 12 (2003), 1572–1574.

[60] Simon A. Berger, Nikolaos Alachiotis, and Alexandros Stamatakis. 2012. An optimized reconfigurable system for computing the phylogenetic likelihood function on dna data. In *Proceedings of the IEEE 26th Int. Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 352–359.

[61] Zheming Jin and Jason D. Bakos. 2013. Extending the BEAGLE library to a multi-FPGA platform. *BMC Bioinformatics* 14, 1 (2013), 25.

[62] Daniel L. Ayres, Aaron Darling, Derrick J. Zwickl, Peter Beerli, Mark T. Holder, Paul O. Lewis, John P. Huelsenbeck, Fredrik Ronquist, David L. Swofford, Michael P. Cummings, et al. 2011. BEAGLE: An application programming interface and high-performance computing library for statistical phylogenetics. *Systematic Biology* 61, 1 (2011), 170–173.

[63] Alexandros Stamatakis and Nikolaos Alachiotis. 2010. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics* 26, 12 (2010), i132–i139.

[64] Fernando Izquierdo-Carrasco and Alexandros Stamatakis. 2011. Computing the phylogenetic likelihood function out-of-core. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 444–451.

[65] Fernando Izquierdo-Carrasco, Julien Gagneur, and Alexandros Stamatakis. 2011. Trading memory for running time in phylogenetic likelihood computations. *Heidelberg Institute for Theoretical Studies* (2011).

[66] Douglas J. Eernisse. 1998. A brief guide to phylogenetic software. *Trends in Genetics* 14, 11 (1998), 473–475.

[67] Alexandros Stamatakis. 2019. A review of approaches for optimizing phylogenetic likelihood calculations. In *Proceedings of the Bioinformatics and Phylogenetics*. Springer, 1–19.

[68] James E. Smith. 1984. Decoupled access/execute computer architectures. *ACM Transactions on Computer Systems* 2, 4 (1984), 289–308.

[69] Tao Chen and G. Edward Suh. 2016. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 46.

[70] John Levesque and Gene Wagenbreth. 2010. *High Performance Computing: Programming and Applications*. Chapman and Hall/CRC.

[71] Maciej Bielski and Andrea Reale. *Hotplug for Arm64*. Retrieved from https://lkml.org/lkml/2016/11/17/49 [Online; LKML.org].

[72] Dimitris Theodoropoulos, Nikolaos Alachiotis, and Dionisios Pnevmatikatos. 2017. Multi-FPGA evaluation platform for disaggregated computing. In *Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 193–193.

[73] Dimitris Theodoropoulos, Andrea Reale, Dimitris Syrivelis, Maciej Bielski, Nikolaos Alachiotis, and Dionisios Pnevmatikatos. 2018. REMAP: Remote mEmory manager for disaggregated platforms. In *Proceedings of the 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors*. IEEE, 1–8.

[74] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. 2014. Marlin: A memory-based rack area network. In *Proceedings of the 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. IEEE, 125–135.

Received March 2021; revised July 2021; accepted August 2021