On-Chip Network-Enabled Multicore Platforms Targeting Maximum Likelihood Phylogeny Reconstruction

Turbo Majumder, *Student Member, IEEE,* Michael Edward Borgens, Partha Pratim Pande, *Senior Member, IEEE,* and Ananth Kalyanaraman, *Member, IEEE*

***Abstract*—In phylogenetic inference, which aims at finding a phylogenetic tree that best explains the evolutionary relationship among a given set of species, statistical estimation approaches such as maximum likelihood (ML) and Bayesian inference provide more accurate estimates than other nonstatistical approaches. However, the improved quality comes at a higher computational cost, as these approaches, even though heuristic driven, involve optimization over multidimensional real continuous space. The number of possible search trees in ML is at least exponential, thereby making runtimes on even modest-sized datasets to clock up to several million CPU hours. Evaluation of these trees, involving node-level likelihood vector computation and branch-length optimization, can be partitioned into tasks (or kernels), providing the application with the potential to benefit from hardware acceleration. The range of hardware acceleration architectures tried so far offer limited degree of fine-grain parallelism. Network-on-chip (NoC) is an emerging paradigm that can efficiently support integration of massive number of cores on a chip. In this paper, we explore the design and performance evaluation of 2-D and 3-D NoC architectures for RAxML, which is one of the most widely used ML software suites. Specifically, we implement the computation kernels of the top three functions consuming more than 85% of the total software runtime. Simulations show that through appropriate choice of NoC architecture, and novel core design, allocation and placement strategies, our NoC-based implementation can achieve individual function-level speedups of 390x to 847x, speed up the targeted kernels in excess of 6500x, and provide end-to-end runtime reductions up to 5x over state-of-the-art multithreaded software.**

***Index Terms*—Hardware accelerator, multicore, network-on-chip (NoC), phylogeny reconstruction.**

I. Introduction

PHYLOGENETIC inference is one of the grand challenge problems in bioinformatics. It aims at finding a phylogenetic tree that best explains the evolutionary relationship for a set of n taxa. In a phylogenetic tree, the taxa form the leaves, and the branches indicate divergence from a common ancestor. Reconstruction of the tree is done by observing and characterizing variations at the DNA and protein level. Broadly, there are three types of approaches used for phylogeny reconstruction: distance-based hierarchical methods (e.g., neighbor joining), combinatorial optimization using maximum parsimony (MP), and statistical estimation methods [e.g., maximum likelihood (ML), Bayesian inference (BI)]. Of these, the estimation approaches such as ML and BI are statistically consistent and are therefore widely used [1]. These methods provide a statistical likelihood score for each reconstructed tree using the phylogenetic likelihood function [2], [3]. The boost in quality, however, comes at a high computation cost as the ML formulation is nondeterministic polynomial-hard [4] and suffers from the need to explore a super-exponential (in n) number of trees. For example, a run using RAxML [5], [6], which is one of the most widely used programs to compute ML-based phylogeny, on an input comprising of 1500 genes can take up to 2.25 million CPU hours [7]. With increasing availability of genomic data, as documented in public genomic data banks such as the National Center for Biotechnology Information [8], the relevance and the utility of the statistical estimation approaches are only expected to grow. However, to realize their potential, scalable methods that use novel combinations of algorithmic heuristics, hardware acceleration, and high-performance computing are needed.

In this paper, we present a novel design of a network-on-chip (NoC) based multicore platform for addressing the issue of computational complexity in ML methods. The rationale for using a NoC to address the ML application stems from the fact that there are different levels of parallelism in the ML algorithmic structure that can be exploited by the NoC to accelerate computation. Fine-grained parallelism can be exploited within a processing element (PE) to render a fast hardware implementation for each phylogenetic function kernel. While the same can also be implemented on a large field-programmable gate array (FPGA) board that supports several computation cores (e.g., similar to [9]), a NoC-based multicore system can also handle coarse-grained parallelism more efficiently [10]. The latter requirement becomes particularly important in the context of ML programs because they typically involve a large number of function invocations (see Section IV); and

Manuscript received August 22, 2011; revised December 6, 2011; accepted January 29, 2012. Date of current version June 20, 2012. This work was supported by NSF, under Grant IIS-0916463. This paper was recommended by Associate Editor R. Marculescu.

T. Majumder, P. P. Pande, and A. Kalyanaraman are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163 USA (e-mail: tmajumde@eecs.wsu.edu; pande@eecs.wsu.edu; ananth@eecs.wsu.edu).

M. E. Borgens is with Intel Corporation, Dupont, WA 98327 USA (e-mail: mborgens@eecs.wsu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2012.2188401

at any instant, there could be a variable number of instances of each function running simultaneously. Each instance of a function occupies a set of nodes, and this associativity is a time-varying property dictated by the application at runtime. As such, use of a bus or even a hierarchical bus does not satisfy the requirement. Using a multicore platform with on-chip network, these requirements can be effectively addressed by:

1) designing a homogeneous system, where application-specific PEs are able to seamlessly support execution of multiple function kernels at different times;
2) interconnecting them using a suitable network that allows concurrent execution of an arbitrary combination of function instances and provides the backbone for efficient data exchange among individual PEs.

In other words, we can build a heterogeneous application map on top of a homogeneous-core NoC. Furthermore, such a homogeneous NoC-based system can be scaled up to provide the computation bandwidth necessary for solving larger problems.

In our solution, we implemented different NoC architectures, and tested their acceleration performance using three most dominant function kernels in RAxML. Based on our experiments on multiple input datasets, these kernels collectively account for more than 85% of the total runtime, and involve several computations of sum-of-products, antilogarithm (exponential) and logarithm. The performance improvements due to the architectural advantages of NoC can be significantly enhanced if 3-D integration is adopted as the basic fabrication methodology. The amalgamation of two emerging paradigms—NoCs in a 3-D IC environment—allows for the creation of new structures that enable significant performance enhancements over traditional solutions. Consequently, we implemented and evaluated three different NoC architectures, namely 2-D torus, 3-D torus, and stacked torus.

The major contributions are as follows:

1) a homogeneous, unified PE design for parallel execution of three function kernels;
2) an efficient, fine-grained implementation of the different floating-point arithmetic operations involved in this application;
3) novel dynamic core-allocation schemes to minimize internode communication latency;
4) exploration and evaluation of the merits of different 2-D and 3-D NoC architectures.

Simulations show that through appropriate choice of NoC architecture, and novel core design, allocation and placement strategies, our NoC-based implementation can achieve function-level speedups of 390x to 847x, aggregate speedups of accelerated kernels in excess of 6500x, and end-to-end runtime reductions of over 5x with respect to state-of-the-art multithreaded software.

II. Related Work

Considerable work has been done on designing hardware accelerators for the different approaches of phylogeny reconstruction. These efforts have generally targeted MP, ML and BI because of their wide usage and large time complexities.

Acceleration of breakpoint-median phylogeny, which is based on MP, is the topic of [11] and [12]. The primary computational characteristic of breakpoint-median phylogeny is the computation of an optimal solution for the traveling salesman problem. In [11], Bakos and Elenis implemented a parallelized version of breakpoint-median phylogeny using both software and FPGA, and achieved a speedup of 417x for a whole-genome phylogeny reconstruction. In [12], we implemented a NoC-based accelerator that achieves a speedup of 774x on genome sizes comparable to those used in [11].

For BI, hardware acceleration has been proposed using cell broadband engine (CBE), GPU and general-purpose multicores [13], and FPGA [9]. These platforms achieve an order of magnitude speedup over software.

For ML phylogeny, which is the target application in this paper, a genetic algorithm using a hybrid hardware–software approach achieves an overall speedup of 30x [14]. A CBE-based implementation [15] for RAxML is shown to outperform the software by 2x. Another implementation of RAxML using FPGA boards [7] with built-in digital signal processor slices achieves a speedup of 8x.

In this paper, we explore the suitability and merits of using the NoC paradigm for ML phylogeny. We extend our analysis by evaluating the performance of 3-D NoC architectures for this application. 3-D NoCs have been shown to have higher performance and lower power consumption [16] compared to 2-D counterparts. 3-D NoCs have been proposed for improving the performance of application-specific architectures in [17]. 3-D design-space exploration for cache memories has been considered in [18]. A detailed comparison of different 3-D NoC interconnect topologies is carried out in [19]. Our results show that the NoC platform can provide two orders of magnitude speedups over other existing hardware accelerators. To the best of our knowledge, this represents the first implementation of a NoC-based accelerator for ML phylogeny.

In our proposed implementation, we use logarithmic calculations in hardware. Fast calculation of logarithms in hardware has been a well-researched topic. Kwon *et al.* [20] described a fast implementation of exponentiation in hardware targeting graphics applications. A 32-bit binary-to-binary linear approximation-based logarithm converter is described in [21]. Optimality of Chebyshev polynomials for table-based approximations of elementary functions is described in [22]. A technique for designing piecewise polynomial interpolators for implementing elementary functions such as logarithm and exponentiation in hardware is described in [23]. A unified computation architecture for calculating elementary functions is presented in [24], which uses a fixed-point hybrid number system (FXP-HNS) to integrate all operations in a power and area-efficient manner with a low percentage of error.

Our approach for accelerating ML phylogeny integrates fast floating-point computation hardware within a multicore NoC framework. We exploit the advantages offered by the 3-D NoC architectures for this application domain and demonstrate tangible improvements in speedup and energy consumption over traditional 2-D NoC architectures.

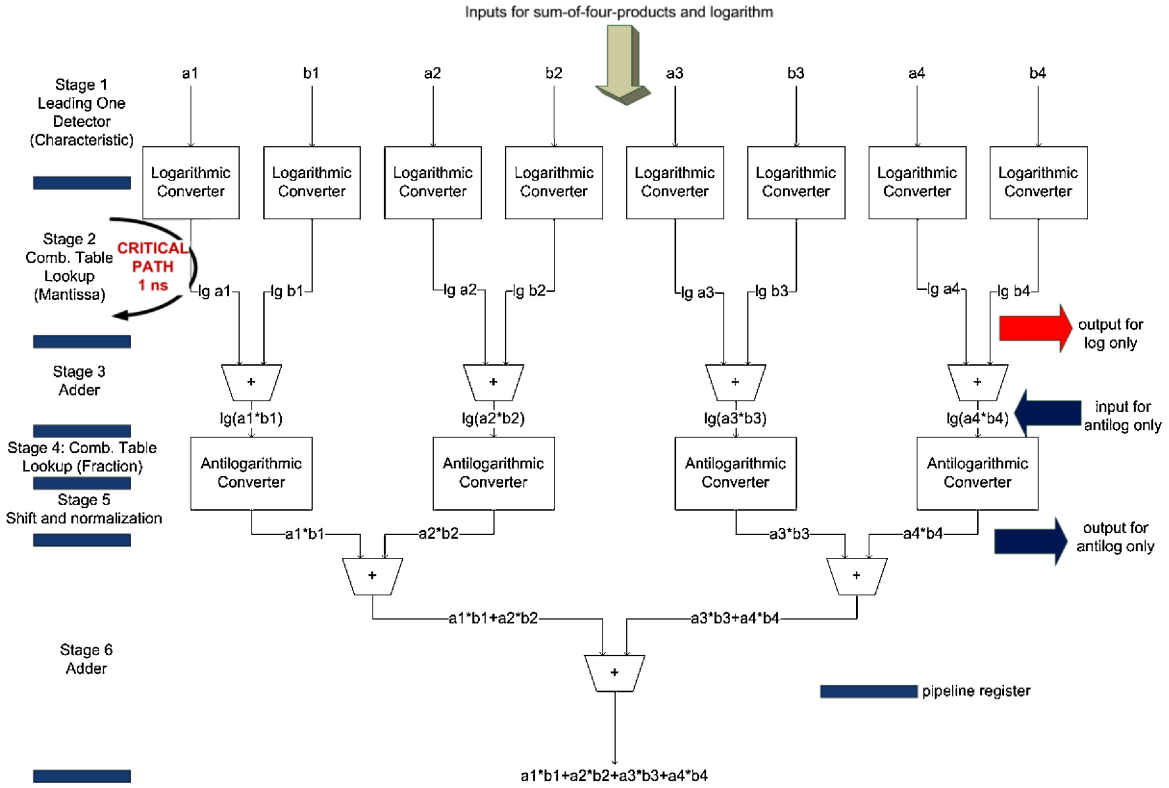

Fig. 1. Internal architecture of PE computation core for sum-of-four-products, logarithm, and antilogarithm.

III. NoC Design

The process of designing our NoC involves the designing of a homogeneous computation core, the interconnect network, application mapping, and routing and arbitration policy. The NoC-based hardware accelerator platform is designed to operate as a coprocessor with a CPU that is running RAxML code and that offloads certain computations to the processing cores of our platform. The offloading happens through a PCIe interface, details of which are described in Section IV-D2.

A. Computation Core

Each core implements the principal computation kernels of three dominant functions of the RAxML suite, namely *coreGTRCAT*, *newviewGTRCAT*, and *newviewGTRGAMMA* [6], which collectively account for more than 85% of the software runtime. These kernels comprise of the following three operations: sum-of-four-products, logarithms, and antilogarithms. We designed a unified core architecture that is capable of performing all these operations. The core has a six-stage pipelined architecture as shown in Fig. 1. For computing sum-of-four-products, all six stages are used. In stages 1 and 2, the input is transformed to the log-domain. Stage 3 adds four pairs of these numbers in the log-domain. Stages 4 and 5 are used to transform these four numbers back to the linear domain. Stage 6 adds four numbers to arrive at the final sum. The same architecture can be used to perform the logarithm operation by using stages 1 and 2. Similarly, antilogarithm (or exponentiation) can be computed using stages 4 and 5. Stages 1, 2, and 4, 5 use piecewise linear table-based approximations as described in [24]. These functions are implemented using logic gates without using a ROM.

The three representative functions of RAxML are instruction-coded to be run on the computation core. The core is instantiated within a wrapper that provides instruction decoding, data fetching, and data write-back functions. The design has been implemented with Verilog HDL and synthesized with a clock frequency of 1 GHz using 65 nm standard cell libraries from CMP [25]. The choice of the clock frequency is based on the critical path delay of 1 ns (stage 2) shown in Fig. 1.

1) *Memory Subsystem:* The computation core has the requisite memory to store the input vectors and the computation results for each step of the function computation, all in FXP-HNS format. The per-PE memory requirement is 0.5 MB. This is implemented in the form of register banks. As mentioned earlier, there are no ROM-based lookup tables for computing logarithm and antilogarithm.

B. NoC Node

The core with a wrapper is designated as a PE. Four such PEs are integrated to form one subgroup. The choice of this subgroup size comes from the fact that the three functions require different numbers of sum-of-four-product computations (8, 12, 24) for which the greatest common divisor is four. The PEs are labeled *PE0*, *PE1*, *PE2*, and *PE3*. A crossbar switch, shown in Fig. 2, connects the four PEs and coordinates communication among them. The crossbar switch has to deal with three kinds of traffic, which consists

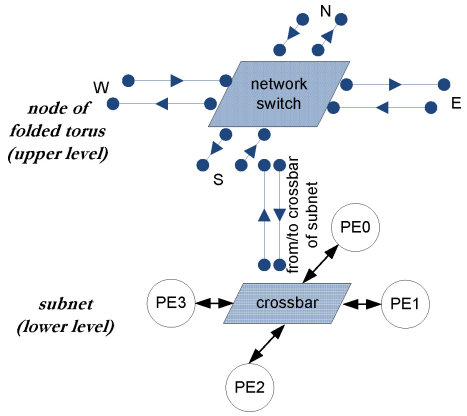

Fig. 2. Network switch of NoC and crossbar-connected subnet.

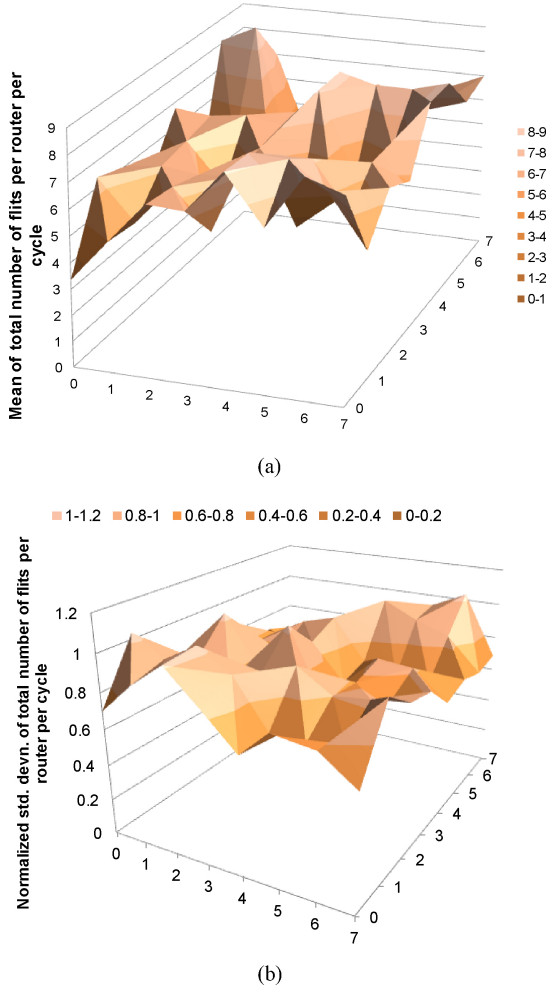

Fig. 3. (a) Mean and (b) normalized standard deviation of flits per cycle in routers in a folded torus network.

of intermediate function results. The first and simplest kind involves sending data received from *PEx* back to *PEx*. The second kind of communication involves sending data from one particular PE to all the other three PEs within the subgroup. The third kind of communication involves sending/receiving data to/from the external network through a network switch. This subgroup of four PEs along with the crossbar switch forms a *subnet* under one *NoC node*. The number of nodes in the system is denoted by N.

C. Network

The choice of the network is determined by the traffic patterns [26] generated by the application. In our case, a single RAxML run typically generates millions of invocations of a few functions at different time-points, and each of these functions can benefit from fine-grain parallelism by an assignment to multiple PEs. This leads to a high volume of arbitrary point-to-point communication. In addition, we observed dynamically changing traffic patterns and a clear absence of steady-state localized traffic or clustering, all of which indicate the desirability of a distributed interconnection topology. A statistical analysis of the traffic patterns under the assumption of an underlying folded torus network reveals this fact. The mean and normalized standard deviation of the number of flits per cycle contained in the buffers of each router in a 8×8 folded torus for a typical application scenario is shown in Fig. 3. The clear lack of clustering can be observed from the absence of prominent peaks in the mean traffic plot in Fig. 3(a). The dynamically varying nature of the traffic can be gleaned from Fig. 3(b) that shows substantial standard deviation of traffic (typically above 50% of the mean) across simulation cycles. Hence, topologies like star or quad-tree that cater to regular or localized traffic patterns would not benefit this application scenario.

From the VLSI implementation perspective, a mesh is a scalable network architecture whose regularity provides for easier timing closure and reduces dependence on interconnect scalability [27]. A folded torus further reduces the *point-to-point separation* (Manhattan distance) between nodes by cutting down the diameter of the network by half without compromising on the regularity or scalability of the entire network. Hence, we decided to explore folded torus in our 2-D NoC design.

As mentioned earlier, 3-D NoCs provide enhanced performance due to the additional degree of freedom in the vertical dimension. For larger system sizes, this enables better integration and reduced internode hop-count [19]. We explored the design of two different 3-D NoC architectures: 3-D folded torus and 3-D stacked torus; and used a system size N of 64 ($= 4 \times 4 \times 4$) in our application study. A 3-D folded torus NoC has a folded torus along each dimension (x, y, and z). There are one-hop vertical links (in the z dimension) between adjacent layers. On the other hand, a stacked torus [18] is a hybrid between a 2-D folded torus, which is a packet-switched network, and a bus, which takes advantage of the short interlayer distances. It integrates multiple layers of folded tori by connecting them with buses spanning the entire vertical height of the chip. Hence, any interlayer communication (for the same $<x, y>$ coordinates) is one-hop.

Since each subnet associated with a node has four PEs, our system has 64 PEs for $N = 16$ and 256 PEs for $N = 64$. A network switch handles traffic emanating from or destined to each network node. We use the switches described in [19] and [28] for our design. Each switch in the 2-D architecture

has four bidirectional ports to neighboring switches and one bidirectional port to the crossbar switch of the subnet (Fig. 2). In the 3-D folded torus, the switch has two additional ports (total seven ports) to the layers above and below. Alternatively, in the 3-D stacked torus, the switch has just one additional port (total six ports) to the vertical bus connecting all the layers.

We adopted wormhole routing-based data exchange among the NoC nodes. The primary data contained in the messages exchanged among nodes are intermediate function results, which are 64-bit numbers using FXP-HNS format [24]. Given this small message size, we split each message into 3 flits (header, body, and tail), each of width 64 bits. Since deeper buffers may slow down clock frequency and do not appreciably improve performance for short messages [29], we use buffer depth of 2 flits. We adopt the routing and arbitration mechanism from [19] and [28].

D. Function-Level Parallelization

The three target phylogenetic function kernels from RAxML are *newviewGTRCAT* (*f2*), *coreGTRCAT* (*f3*), and *newviewGTRGAMMA* (*f6*). We parallelize each function by breaking down larger computation arrays into smaller units as follows. Taking *newviewGTRCAT* (*f2*) as an example, we can see from Fig. 4 that computation of the *x3* array in each iteration requires computation of eight sums-of-four-products, using arrays *left*, *x1*, *x2* and the eigenvalue vector *EV[15:0]*. Since each PE can compute one sum-of-four-products, eight PEs (or equivalently two NoC nodes) are required. We refer to this function as *f2*, indicating that its computation requires two NoC nodes. Similarly, each iteration within the function *coreGTRCAT* (*newviewGTRGAMMA*) involves computation of up to 12 (24) sums-of-four-products, thereby requiring three (six) NoC nodes. Hence, we refer to it as *f3* (*f6*). Other operations, such as carrying out the exponentiation operation involved in computing the *left* vector in *newviewGTRCAT*, are also computed in the PEs within each node. Also, in *coreGTRCAT* for example, sum-of-products, exponentiation, and cumulative addition are involved. All these operations are executed in parallel over multiple PEs. Intermediate results are redistributed among PEs within the same node using the intranode crossbar switch and among other nodes using the network switch/router.

E. Dynamic Node Allocation

A node is *busy* when the PEs within its subnet are collectively executing a function; otherwise it is *available*. Nodes continually keep sending their busy/available status to a centralized controller (*MasterController*) that dynamically allocates a subset of nodes from the set of available NoC nodes to a function. If the number of available nodes at any point of time is less than the number of nodes requested by that function (2, 3, or 6), the function waits till the requisite number of nodes is available. The nodes allocated for executing one function instance are said to belong to one *partition*. Nodes can be reused after execution of the function has completed in the partition.

Since the nodes belonging to a given dynamically allocated partition need to communicate with one another, it is desired that they be colocated on the network in order to reduce the number of hops required for data exchange and thereby reduce the communication latency associated with the function. A good allocation strategy needs to ensure this colocality, as well as execute fast enough to not introduce any significant allocation overhead.

One approach for allocating a partition is to use breadth-first search (BFS) on the network. Although this appears to be a reasonable strategy, there are certain drawbacks. First, BFS does not guarantee the colocality of nonroot nodes. The dispersion could become greater if the root node for BFS lies in a neighborhood containing a majority of busy nodes. Higher dispersion among allocated nodes in a partition results in a higher average message hop-count, resulting in higher communication latency. Second, the allocation overhead becomes dependent on the choice of the BFS root node, and growing a partition around a root node surrounded by a majority of busy nodes has the risk of increasing the allocation overhead. This is because the *MasterController* handling the node allocation has to traverse each node in the neighborhood (in the adjacency list of the parent node). Scanning the adjacency list of each node takes one clock cycle, and therefore, growing the full partition requires a number of cycles equal to the number of nodes requested by the function in the best case, and N clock cycles in the worst case, where N is the system size.

In this paper, we have developed a novel approach that uses the Hilbert curve [30] for the dynamic allocation problem. A Hilbert curve is a locality-preserving space-filling curve widely used in scientific computing [31]. In addition, this approach results in consistently lower allocation times, as described below. In the following subsections, we describe different approaches for dynamic node allocation that make use of the Hilbert curve superposed on different 2-D and 3-D NoC architectures.

1) *2-D Hilbert Curve With Serial Scan and First Fit (2D_serial):* In our first approach, we use the Hilbert curve on a 2-D folded torus as follows. The *MasterController* serially scans the nodes along the Hilbert curve and chooses the required number of available nodes and allocates them as a partition to the requesting function.

Using the Hilbert curve offers a couple of key advantages. A Hilbert curve has the property that when mapped onto a regular mesh or a folded torus, nodes adjacent along the Hilbert curve traversal are also adjacent on the network. Furthermore, there could be nodes which are not adjacent along a Hilbert curve but are adjacent on the folded torus. Also, a Hilbert curve is essentially converting a 2-D allocation problem into a 1-D problem. Taking advantage of this property, we use a fixed Hilbert curve embedded on a folded torus as shown in Fig. 5 (for $N = 16$), where there is a one-to-one correspondence between the node ids on the torus and those on the Hilbert curve. This allows us to effectively predetermine the set of possible nodes for allocation. Since this information can be hard-wired in the design, the allocation of an entire partition can be achieved in one clock cycle (for $N = 16$) or four clock cycles (for $N = 64$) in our design.

Note that our Hilbert curve-based approach may lead to three scenarios, as shown in Fig. 5. First, allocated nodes are

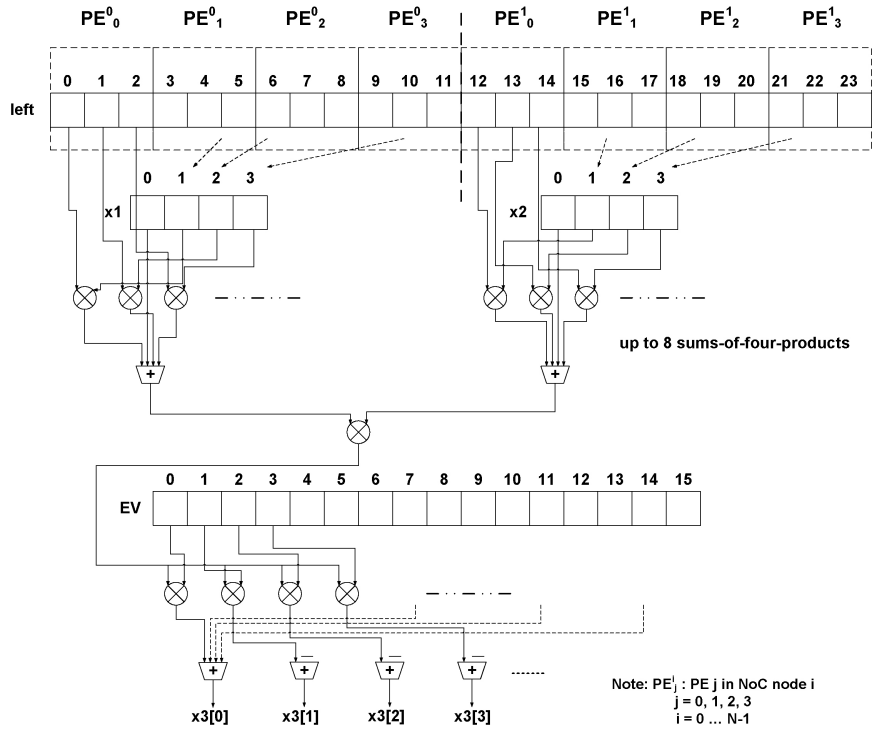

Fig. 4. Schematic representation of computation tree of *newviewGTRCAT* (*f2*).

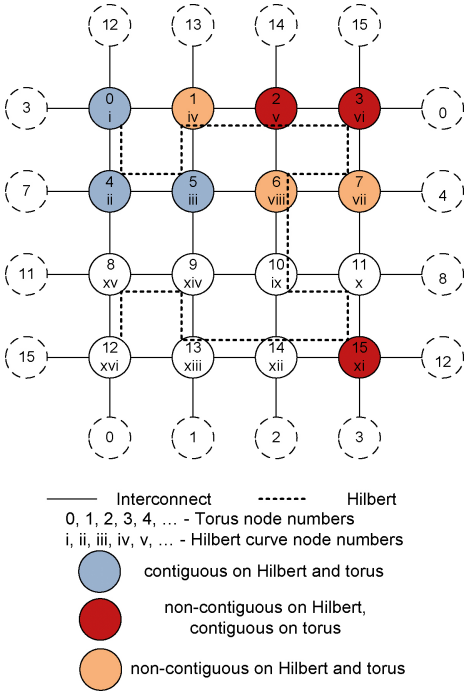

Fig. 5. Hilbert curve embedded in folded torus network architecture (for $N = 16$) showing contiguous and noncontiguous partitions.

all adjacent to each other or *contiguous* on Hilbert and hence the *partition* is *contiguous* on the torus. Second, the nodes are *noncontiguous* on Hilbert but form a *contiguous partition* on the torus. Third, the nodes are *noncontiguous* both on Hilbert and on the torus.

2) *Multiple 2-D Hilbert Curves With Parallel Scan and Best Fit (2D_parallel):* Despite the ease of implementing the *2D_serial* approach, there are two main drawbacks. First, there is a constant allocation penalty of four cycles per partition for a system size of 64. In addition, the allocation policy in *2D_serial* is first-fit. Hence, it does not guarantee allocation of a contiguous partition even if one is available. Therefore, we developed an alternative approach, *2D_parallel*, where we make the following changes to the allocation policy. This policy is particularly suited for larger system sizes; so we will use $N = 64$ in the following description of the underlying algorithm.

1) First, we use four Hilbert curves on a square folded torus in *2D_parallel* (instead of one as in *2D_serial*). These four curves are obtained by using right-angle rotation operations of a single Hilbert curve.
2) We further divide each of the four Hilbert curves into four *segments*, one from each quadrant—thereby resulting in a total of 16 segments. The *MasterController* module now has 16 heads, each of which is responsible for scanning a segment. All 16 heads act in parallel.
3) Each head now preferentially looks for a contiguous partition starting from any of the nodes in its segment. The first head to find a contiguous partition returns it to the requesting function and interrupts all the other scanning heads.
4) In case, no contiguous partition is found after each head has finished scanning its segment, we fall back to execute *2D_serial*.

We have experimentally verified that step 4) has a low probability (<0.2) of occurring and a contiguous partition can be found in most cases. Although there is an additional allocation penalty due to the best-fit strategy we use [step 3)], it provides a higher percentage of contiguous partitions

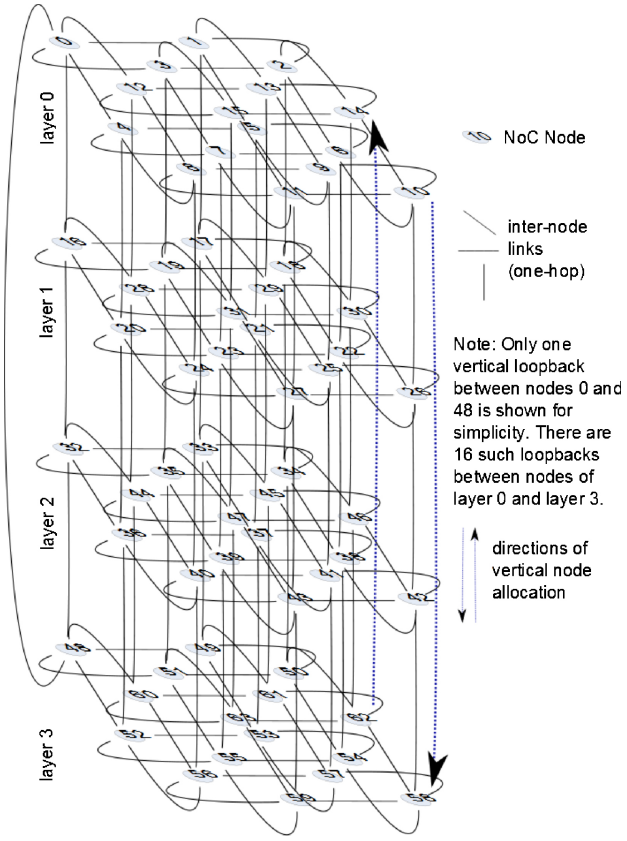

Fig. 6. 3-D folded torus NoC architecture for $N = 64$; also shown are the alternating vertical node allocation directions.

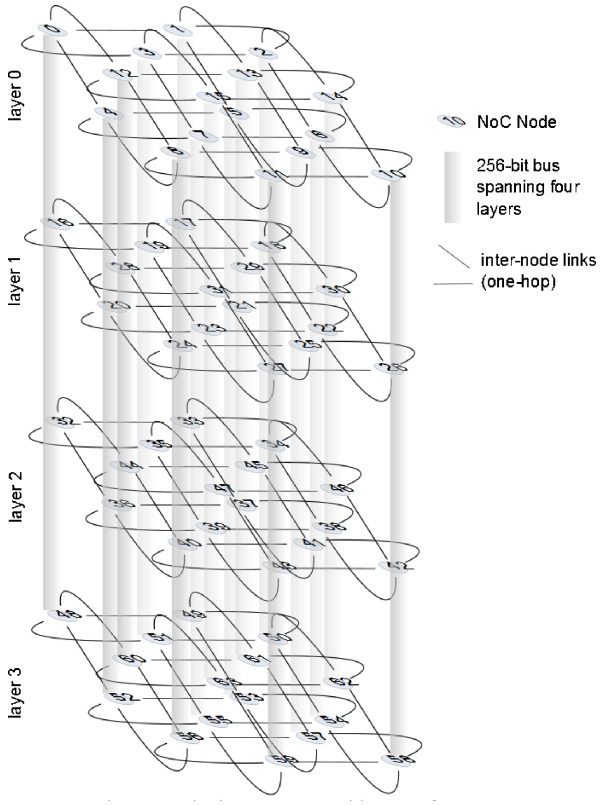

Fig. 7. Stacked torus NoC architecture for $N = 64$.

than is obtained using *2D_serial*. The average number of cycles spent per allocation comes down from 4 to 3.22. More importantly, greater contiguity of allocated partitions reduces internode communication latency and provides better speedup, as will be shown in Section IV.

3) *3-D Folded Torus NoC (3D_torus):* To further improve contiguity of the allocated partitions while spending fewer allocation cycles, we map our application to a 3-D folded torus architecture. The NoC is a 4×4×4-folded torus as shown in Fig. 6. A 2-D 16-point Hilbert curve is embedded on the top layer (layer 0) and is used to allocate partitions. For each allocation request, *MasterController* allocates all *available* nodes in the column (consisting of four layers) corresponding to the current head position. The next head position follows from the 16-point Hilbert curve. This is done till all requested nodes are allocated. In addition, we ensure vertical contiguity by flipping the vertical direction of allocation. For instance, if the most recent node allocated in the current column is from layer 3, the next node to be allocated comes from layer 3 in the column corresponding to the next head position. In other words, we alternately move up and down the columns during node allocation. We are able to handle allocation of nodes in one vertical column in one cycle; hence the average allocation time for *3D_torus* goes down to 1.56 cycles. As shown in Section IV, *3D_torus* provides the highest speedup and greatest energy efficiency.

4) *3-D Stacked Torus (3D_sttorus):* Another popular 3-D NoC architecture is the stacked torus. For our application, we have four 4×4-folded tori vertically stacked using 16 buses as shown in Fig. 7. Bus width is a determinant of the performance of a stacked torus. As shown in [19], a stacked torus with a bus width of 4 flits achieves the same throughput performance as of a 3-D torus. Hence, we use a bus width of 4 flits, i.e., 256 bits in our design. Note that these are very short buses spanning four layers. The allocation policy in *3D_sttorus* is exactly the same as *3D_torus*. However, as a consequence of the allocation method, our application generates significant amount of traffic between nodes in the same column, which in turn leads to bus contention and destination contention, as will be shown in Section IV.

F. Routing and Arbitration

Our routing policy is based on dimension-order: XY routing on folded torus for *2D_serial* and *2D_parallel*, and XYZ routing for *3D_torus* and *3D_sttorus*.

In *2D_serial* and *2D_parallel* systems, we distinguish between messages originating from contiguous partitions and noncontiguous partitions, and the corresponding flits are designated as *A-type* or *B-type*, respectively. Each node has a set of allowed directions depending on the partition it is situated in. For a contiguous partition, any network switch on the partition boundary has channels leading out of the partition marked as *disallowed*. For noncontiguous partitions, all network switches have all directions marked as *allowed*. In other words, traffic emanating from a contiguous partition always remains within the partition boundary and traffic emanating from a noncontiguous partition is free to move in any direction

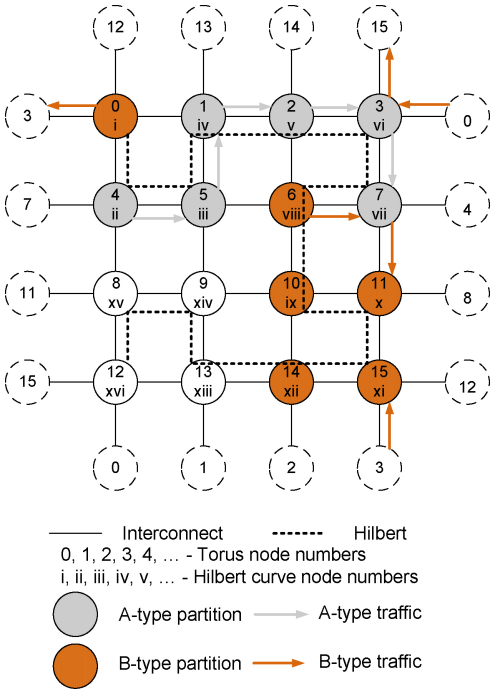

Fig. 8. Examples of different paths taken while routing A-type and B-type traffic.

dictated by the routing policy. At each network switch, an A-type message is restricted to make the next hop in one of the allowed directions. However, since B-type messages have unrestricted access, they follow torus routing, which is similar to XY routing but includes the torus loopback information to determine the shortest path. A-type messages take the X direction if that direction is allowed, and Y direction otherwise. Fig. 8 shows an example. A message in an A-type partition going from node 4 to node 7 follows the path indicated. In the B-type partition, there is one message going from node 6 to node 11 via node 7 outside the partition. Another message from node 0 to node 15 goes through node 3, which is outside the partition, and makes use of the torus loopback.

It can be noted from the above routing mechanism that switches internal to a contiguous partition will face A-type traffic from that partition and may face B-type traffic from any noncontiguous partition(s). On the other hand, switches internal to a noncontiguous partition will face only B-type traffic from noncontiguous partition(s). When there is more than one message competing for the same port at any switch, the following arbitration policy is used. The remaining hop-count of the message is determined by looking up the pre-calculated Manhattan distance from the current node to the destination. The message with the maximum remaining hop-count, i.e., the one farthest from its destination, is granted the channel. In case of a tie, B-type is given preference. The remaining hop-count is also used as the arbitration parameter while routing in *3D_torus*. This policy ensures that traffic with a higher potential latency is routed earlier, thereby reducing worst-case latency.

Since B-type messages in *2D_serial* and *2D_parallel* follow XY routing on torus (as described above), any noncontiguous partition is automatically deadlock-free. For contiguous partitions of sizes 2 and 3, there is no possibility of a cycle in the channel dependence graph because the message is always contained within the partition and two or three nodes cannot form a cycle on a torus. Hence, deadlock is avoided in this case. For contiguous partitions of size 6, we can have a partition like the A-type partition (nodes 1, 2, 3, 4, 5, and 7) in Fig. 8 or a partition comprising of nodes 8, 9, 10, 12, 13, and 14 in Fig. 5. In the former case, we do not have a cycle and hence deadlock cannot arise. In the latter case, because we follow XY routing, deadlocks are avoided. For routing in *3D_torus* and *3D_sttorus*, we follow XYZ (dimension-order) routing. Therefore, our routing and arbitration policy for each architecture is deadlock-free.

IV. Experimental Results

A. Experimental Setup

The computation core has a datapath width of 64 bits and provides a number representation accuracy of 2^{-52}. We synthesized Verilog RTLs for the computation core, the instruction-decoding wrapper, the routers and *MasterController* with 65 nm standard cell libraries from CMP [25]. The NoC interconnects are laid out and their physical parameters (power dissipation, delay) are determined using the extracted parasitics (resistances and capacitances). Use of folded torus topology prevents occurrence of long warp-back wires. The critical path occurs in the PE datapath as mentioned in Section III-A, following which we used a clock with 1 ns period. We simulated *2D_serial* NoCs with system sizes $N = 16$, and *2D_serial*, *2D_parallel*, *3D_torus* and *3D_sttorus* NoCs with $N = 64$ using the NoC simulator used in [28]. Recall that there are four PEs per NoC node in the system.

The NoC-based multicore platform is modeled as a co-processor connected using a PCIe interface. We modeled a PCI Express 2.0 interface using Synopsys Designware IP PCI Express 2.0 PHY. This IP has been implemented on 65 nm process and operates at 5.0 Gb/s. We use a 32-lane PCIe 2.0 for our simulation.

We ran RAxML-VI-HPC (version 7.0.4) [6] on three inputs that are provided with the suite. These inputs comprised of DNA sequences originally derived from a 2177-taxon 68-gene mammalian dataset described in [32]. We ran RAxML in single and multithreaded modes on a Pentium IV 3.2 GHz dual-core CPU, and used the best software runtimes (four threads or 4T) as our baseline. Furthermore, to measure the relative computation intensities of each function kernel, we profiled RAxML on all inputs using the GNU *gprof* utility. The results consistently showed that the functions *coreGTRCAT* (*f3*) (48%), *newviewGTRGAMMA* (*f6*) (21%), and *newviewGTRCAT* (*f2*) (17%) collectively account for more than 85% of the total runtime. The average CPU times spent in the invocation of each of the three functions were also noted; these times are labeled T_{f2}, T_{f3}, and T_{f6}. We generated 100 bootstrap trees using RAxML for each input and used them for subsequent likelihood calculation.

We compared the numerical results produced in our PEs with the ones produced while running RAxML on the above-mentioned CPU using eight decimal places of precision and

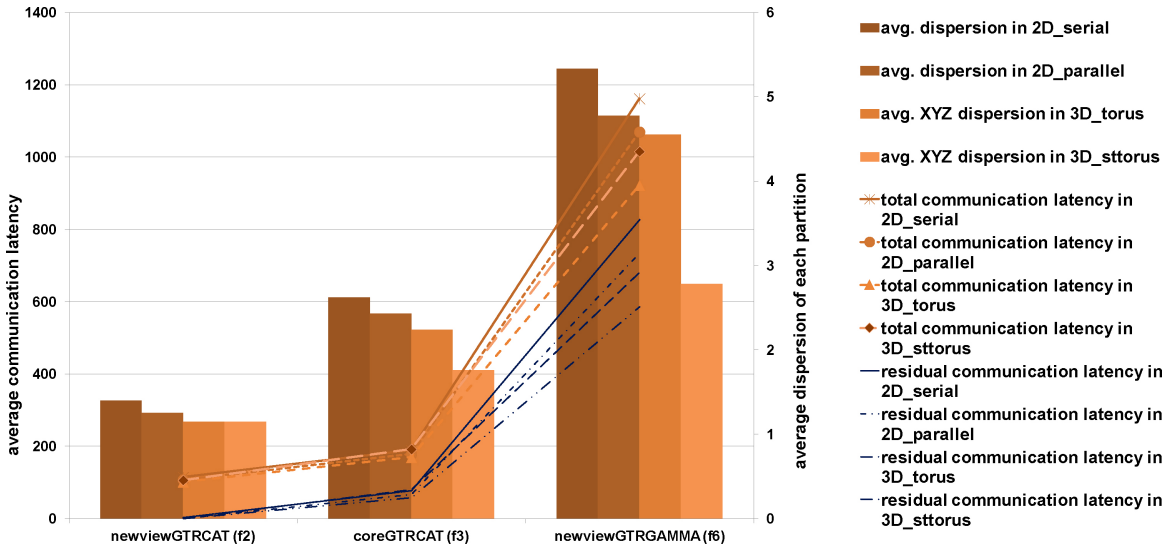

Fig. 9. Variation of partition dispersion and function communication latency across different NoC architectures.

verified that the average percentage of deviation was below 0.1%, which was within tolerable limits and did not hamper the stability of RAxML or the likelihood computation.

B. Test Case Design

The function kernels whose acceleration we target are invoked during generation of bootstrap trees and computation of likelihood of the generated trees to find out the best tree. Working on each tree in parallel helps us work around the sequential dependence among functions within one execution thread. Target function kernels originate from different parallel execution threads and hence can be allocated to different nodes of the NoC-based platform. Allocation of nodes to each function is based on the policy in Section III-E. We designed test cases for *2D_serial* (N = 16 and N = 64), *2D_parallel* (N = 64), *3D_torus* (N = 64), and *3D_sttorus* (N = 64). Each test case represents a combination of the three target functions (*f2*, *f3*, *f6*). In order to compare the different NoC architectures, we use the same test cases on each. However, the allocation of a test case can result in different mixtures of contiguous and noncontiguous partitions depending on the underlying architecture and system size. Test cases have been captured from a wide range of real-world scenarios, including the best and the worst case. The mean execution time of each function is estimated by averaging over all the test case scenarios. During the execution of a function, internode exchange of intermediate results occurs simultaneously with intranode computation (in the PEs within the subnet). This allows masking of communication latency by computation delay. We observed that *newviewGTRCAT* (*f2*) requiring two nodes per invocation is generally computation-intensive, while *coreGTRCAT* (*f3*) and *newviewGTRGAMMA* (*f6*) requiring three and six nodes, respectively, are generally communication-intensive.

C. Communication Latency

Total communication latency indicates the amount of time spent in executing the function. Since computation and communication are pipelined, we define *residual communication latency* as the number of clock cycles spent in performing only internode communication. The average lifetime of each partition is closely related to the total communication latency. The contiguity (or noncontiguity) of an allocated partition has a direct bearing on the communication latency (total or residual) for executing the function. The effect is most pronounced in the case of *newviewGTRGAMMA* (*f6*) and also affects *coreGTRCAT* (*f3*). On the other hand, *newviewGTRCAT* (*f2*) has a net zero residual communication latency. This is because this function is spread across only two nodes and computation and communication cycles complement each other. We use average partition dispersion (diameter) as a measure of the noncontiguity of the allocated partition. We observe (Fig. 9) a gradual decline in average partition dispersion moving from *2D_serial* to *2D_parallel* to *3D_torus*. The average communication latency involved in function execution displays a similar trend across architectures. The role of the interconnection topology here is to reduce the average partition dispersion and hence the residual communication latency. B-type messages originating from noncontiguous partitions as a percentage of the total number of messages reduce from 34% in *2D_serial* to 24% in *2D_parallel*, further demonstrating the impact of contiguity of partitions on latency performance. Partitions on *3D_sttorus* have much lower dispersion than the other architectures owing to the presence of a bus in the vertical dimension, which provides one-hop transit between any two layers. However, this does not translate to lower communication latency because of bus and destination contention. In fact, latencies for *3D_sttorus* are observed to be slightly higher than those in *3D_torus* (most pronounced for *f6* as shown in Fig. 9).

D. Speedup

We used two different measures to evaluate the acceleration performance of our design. The first measure is function-level speedup, which assesses the level of fine-grained parallelism achieved by our PE design. The second measure is aggregate speedup of the accelerated kernels, which measures the degree of acceleration achieved by integrating the PEs in the NoC framework.

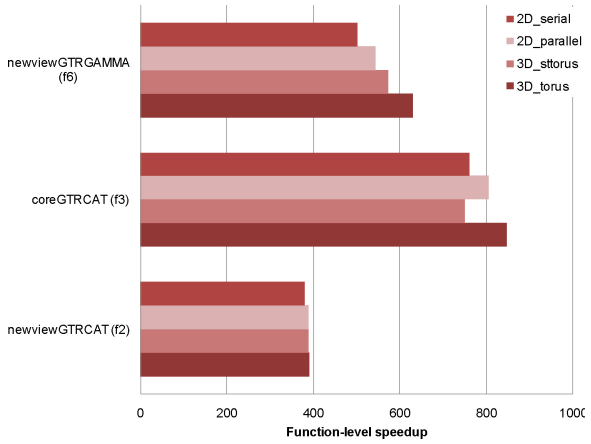

Fig. 10. Function-level speedup across different NoC architectures.

1) *Function-Level Speedup:* In order to determine function-level speedup, the total execution time for each function, consisting of computation and communication components, was averaged over all test cases on each architecture (*2D_serial*, *2D_parallel*, *3D_torus*, and *3D_sttorus*) and compared with the baseline CPU times consumed by the function while running the software (T_{f2}, T_{f3}, T_{f6}). The speedup obtained for the functions on each architecture is shown in Fig. 10. *3D_torus* consistently provides the best function-level speedup for all three functions. Note that the best speedup (on *3D_torus*) of 847x is obtained for *coreGTRCAT* (*f3*), which accounts for 48% of the total software runtime. The least speedup (on *3D_torus*) of 390x is obtained for *newviewGTRCAT* (*f2*), because it is the smallest function kernel and requires only two NoC nodes (or eight PEs) by design. As expected, function-level speedup has an inverse relationship with communication latency (Fig. 9).

2) *Aggregate Speedup of the Target Function Kernels:* This is a measure of the acceleration achieved on the targeted function kernels, and is the ratio of the CPU runtimes of the test cases consisting of these kernels to the runtimes of these test cases on our NoC-based platform of a given system size (N) and architecture (*2D_serial*, *2D_parallel*, *3D_torus*, and *3D_sttorus*). Each test-case configuration represents a typical snapshot of the system during the course of execution of parallel RAxML threads, with our NoC-based platform handling the three phylogenetic kernels. Several instances of *newviewGTRGAMMA* (*f6*), *coreGTRCAT* (*f3*), and *newviewGTRCAT* (*f2*) occupying contiguous and noncontiguous partitions are present in each such test-case. The total time spent in one test case also includes the time required to allocate all partitions (*allocation time*) and to load the input vectors to the function in 64-bit FXP-HNS format [24] (*interface time*) on the NoC using the PCIe interface described earlier.

On average, *2D_serial* with $N = 16$ provides a speedup of ∼2200x, whereas a larger system size ($N = 64$) provides ∼4300x speedup. The ideal increase (4x) in speedup with system size was not obtained because of higher penalties incurred in *allocation time* and *interface time*, and higher noncontiguity of partitions leading to increased communication latency. This

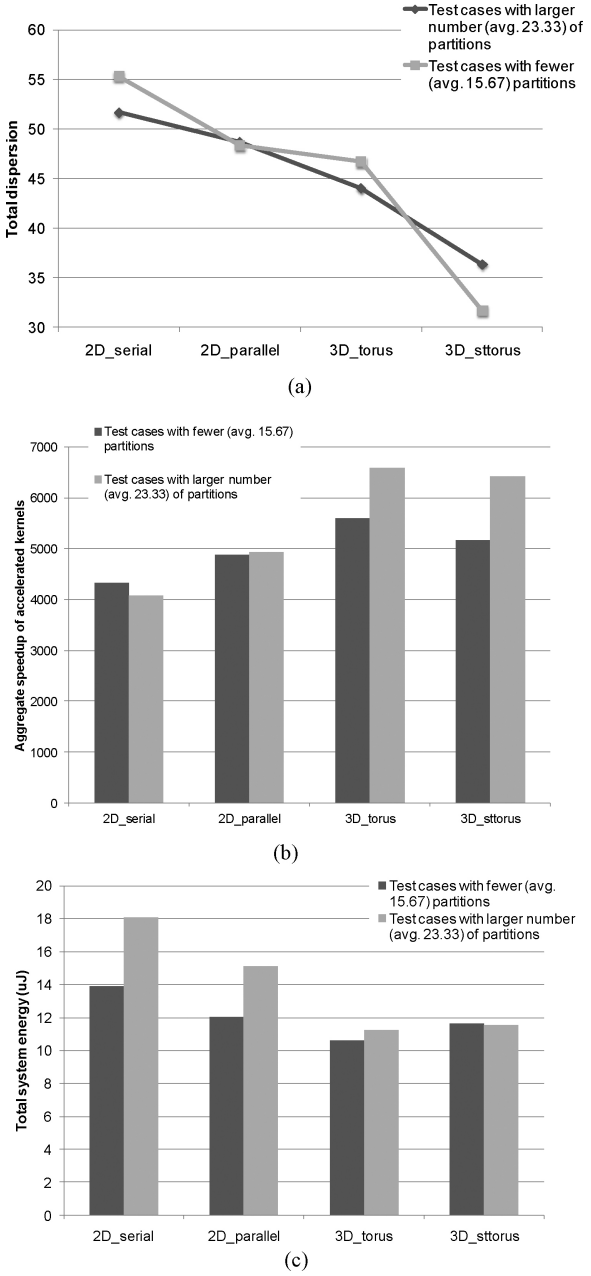

Fig. 11. (a) Total dispersion across different NoC architectures. (b) Average aggregate speedup of the accelerated kernels across different NoC architectures. (c) Total system energy consumption across different NoC architectures.

is where the benefits provided by *2D_parallel*, *3D_torus*, and *3D_sttorus* become evident.

We classified test cases on systems with $N = 64$ on the basis of the number of constituent functions (or partitions). Test cases with a lower number of partitions (average 15.67) have more instances of *f6*. Such instances occur mainly during the likelihood evaluation phase. Test cases with higher number of partitions (average 23.33) have significantly more instances of *f2* and *f3*. These scenarios are prominent during generation of bootstrap trees. Fig. 11(a) shows the observed dispersion as a function of the underlying architecture and the number of partitions. A test case with fewer partitions is expected to result in a higher degree of dispersion because there are

TABLE I
TOTAL RUNTIMES FOR DIFFERENT INPUTS USING DIFFERENT NOC-BASED PLATFORMS VIS-A-VIS ONLY SOFTWARE

Input data (DNA)		Unaccelerated software run-time (s)	Time spent in accelerated kernels (s)	Allocation time (s)	PCIe interface time (s)	Total run-time using NoC platform as hardware accelerator (s)	Total 4T software run-time (s)
50_5000	2D_serial	292.000444	0.515478	0.130387	0.145065	292.791374	924.052039
	2D_parallel	292.000444	0.481303	0.104805	0.145065	292.731617	924.052039
	3D_torus	292.000444	0.433625	0.050889	0.145065	292.630024	924.052039
	3D_sttorus	292.000444	0.474657	0.050889	0.145065	292.671056	924.052039
500_5000	2D_serial	7038.847538	19.1142	8.467062	8.273363	7074.702162	37124.7233
	2D_parallel	7038.847538	18.04733	6.805803	8.273363	7071.974034	37124.7233
	3D_torus	7038.847538	16.766102	3.304655	8.273363	7067.191658	37124.7233
	3D_sttorus	7038.847538	18.102936	3.304655	8.273363	7068.528491	37124.7233

more instances of *f6*. This is generally true for all the architectures except for *3D_sttorus* because of the bus. Fig. 11(b) shows the aggregate speedups of the accelerated kernels as a function of the underlying architecture and the number of partitions. *2D_parallel*, *3D_torus*, and *3D_sttorus* NoCs provide higher speedup than *2D_serial* because they reduce *allocation time* while improving partition contiguity. For all test cases, *3D_torus* provides the best aggregate speedup (6594x) followed by *3D_sttorus* (6428x), *2D_parallel* (4937x), and *2D_serial* (4326x). *3D_torus* outperforms *3D_sttorus* because bus and destination contention in *3D_sttorus* leads to higher communication latency (as described earlier in Section IV-C).

E. Total Execution Time

In order to determine the overall reduction in runtime, the runtime of the nonaccelerated portion of the software is considered along with the accelerated portion running on the NoC-based platform. The total execution time takes into account all overheads involved in offloading a part of the computation to the NoC-based platform. Table I shows the total runtimes for two representative input datasets, 50_5000 containing 50 DNA sequences with 5000 columns each and 500_5000 containing 500 DNA sequences with 5000 columns each. Table I shows the total runtime using our *2D_serial*, *2D_parallel*, *3D_torus*, and *3D_sttorus* architectures vis-a-vis software. The best runtime reduction is obtained using *3D_torus* NoC-based platform and is highlighted in the table. It can be observed that most of the runtime that results from the use of the NoC-based platform comes from the unaccelerated portion. Even so, the overall runtime is reduced by more than 3x for 50_5000 and more than 5x for 500_5000. This proves the immense potential of such hardware accelerator platforms in the field of phylogeny reconstruction applications.

F. Energy Consumption

Fig. 11(c) shows the total energy consumed across different test cases and architectures. Test cases with larger number of partitions consume more energy than those with fewer partitions on the same architecture. However, there is a significant reduction (up to 37.7%) of energy going from *2D_serial* to *3D_torus*. This follows a trend similar to that observed for total test case dispersion [Fig. 11(a)]. Lower dispersion leads to lower average hop-count of internode messages and hence lower energy consumed in communication. Also, in the case of 3-D (both *3D_torus* and *3D_sttorus*), substitution of longer horizontal links with much shorter vertical links leads to lower energy consumption. *3D_sttorus* has a slightly higher overall energy consumption over *3D_torus* because of the higher capacitance of the buses and higher application runtimes.

V. CONCLUSION

In this paper, we presented a novel design and implementation of a NoC-based multicore platform for accelerating ML-based phylogeny reconstruction, which is an important, compute-intensive application in bioinformatics. The NoC-based accelerator targeted the three most time-consuming function kernels that collectively account for the bulk of the runtime in the widely used RAxML software suite. Our implementation achieved parallelization at different levels—both within a function kernel and across several invocations of these function kernels in parallel execution threads. Consequently, our contributions include: 1) the design of a fine-grained parallel PE architecture; 2) a novel algorithm to dynamically allocate nodes to tasks based on Hilbert space-filling curves; and 3) the design and extensive evaluation of different NoC architectures, both in 2-D and 3-D, in the context of this application. The overarching purpose of our experimental study was to evaluate the feasibility and merits of a NoC-based hardware accelerator for ML-based phylogenetic kernels. To this end, our experimental results showed that our NoC-based accelerators are capable of achieving a function-level speedup of ~847x, aggregate speedup of the accelerated portion up to ~6500x, and overall runtime reduction of more than 5x over multithreaded software. Comparative evaluation across NoC architectures showed that the best performances in terms of speedup and energy consumption were obtained from 3-D NoC platforms. Our speedup performance represented considerable improvement over existing hardware accelerators for this application.

Although this paper targeted the RAxML implementation of ML phylogeny, the design methodology and ideas for node allocation and routing are generic enough to be carried forward to other scientific applications which have a similar computational footprint, i.e., the need to execute a large volume of a fixed number of function kernels, for example, other statistical estimation methods in phylogenetic inference such as BI.

ACKNOWLEDGMENT

The authors would like to thank Prof. E. Roalson for the insightful discussions and comments that helped us better our understanding of the problem from a biological perspective.

REFERENCES

[1] C. R. Linder and T. Warnow, "Chapter 19: An overview of phylogeny reconstruction," in *Handbook of Computational Molecular Biology* (Computer and Information Science Series), S. Aluru, Ed. Boca Raton, FL: Chapman and Hall/CRC, 2005.

[2] J. Felsenstein, "Evolutionary trees from DNA sequences: A maximum likelihood approach," *J. Molecular Evol.*, vol. 17, no. 6, pp. 368–376, 1981.

[3] J. Felsenstein, *Inferring Phylogenies*. Sunderland, MA: Sinauer, 2004.

[4] B. Chor and T. Tuller, "Maximum likelihood of evolutionary trees: Hardness and approximation," *Bioinformatics*, vol. 21, no. 1, pp. 97–106, 2005.

[5] A. Stamatakis, "RAxML-VI-HPC: Maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models," *Bioinformatics*, vol. 22, no. 21, pp. 2688–2690, Nov. 2006.

[6] The Exelixis Laboratory, Heidelberg Institute for Theoretical Studies, Heidelberg, Germany [Online]. Available: http://sco.h-its.org/exelixis/software.html

[7] N. Alachiotis, E. Sotiriades, A. Dollas, and A. Stamatakis, "Exploring FPGAs for accelerating the phylogenetic likelihood function," in *Proc. IEEE Int. Symp. Parallel Distributed Process.*, May 2009, pp. 1–8.

[8] National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD [Online]. Available: http://www.ncbi.nlm.nih.gov

[9] S. Zierke and B. Bakos, "FPGA acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods," *BMC Bioinformat.*, vol. 11, p. 184, Apr. 2010.

[10] L. Benini and G. De Micheli, "Networks on chip: A new SoC paradigm," *IEEE Trans. Comput.*, vol. 49, nos. 2–3, pp. 70–71, Jan. 2002.

[11] J. Bakos and P. Elenis, "A special-purpose architecture for solving the breakpoint median problem," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 12, pp. 1666–1676, Dec. 2008.

[12] T. Majumder, S. Sarkar, P. Pande, and A. Kalyanaraman, "An optimized NoC architecture for accelerating TSP kernels in breakpoint median problem," in *Proc. IEEE Int. Conf. Applicat.-Specific Syst. Architectures Processors*, Jul. 2010, pp. 89–96.

[13] F. Pratas, P. Trancoso, A. Stamatakis, and L. Sousa, "Fine-grain parallelism using multi-core, cell/BE, and GPU systems: Accelerating the phylogenetic likelihood function," in *Proc. IEEE Int. Conf. Parallel Process.*, Sep. 2009, pp. 9–17.

[14] T. S. T. Mak and K. P. Lam, "High speed GAML-based phylogenetic tree reconstruction using HW/SW codesign," in *Proc. Comp. Syst. Bioinformat.*, 2003, p. 470.

[15] F. Blagojevic, A. Stamatakis, C. D. Antonopoulos, and D. S. Nikolopoulos, "RAxML-cell: Parallel phylogenetic tree inference on the cell broadband engine," in *Proc. IEEE Int. Symp. Parallel Distributed Process.*, Mar. 2007, pp. 1–10.

[16] V. F. Pavlidis and E. G. Friedman, "3-D topologies for networks-on-chip," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 10, pp. 1081–1090, Oct. 2007.

[17] S. Yan and B. Lin, "Design of application-specific 3-D networks-on-chip architectures," in *Proc. Int. Conf. Comput. Des.*, 2008, pp. 142–149.

[18] Y.-F. Tsai, F. Wang, Y. Xie, N. Vijaykrishnan, and M. J. Irwin, "Design space exploration for 3-D cache," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 4, pp. 444–455, Apr. 2008.

[19] B. S. Feero and P. P. Pande, "Networks-on-chip in a three-dimensional environment: A performance evaluation," *IEEE Trans. Comput.*, vol. 58, no. 1, pp. 32–45, Jan. 2009.

[20] Y.-S. Kwon, I.-C. Park, and C.-M. Kyung, "A hardware accelerator for the specular intensity of Phong illumination model in 3-dimensional graphics," in *Proc. Asia South Pacific Des. Autom. Conf.*, Jun. 2000, pp. 559–564.

[21] K. H. Abed and R. E. Siferd, "CMOS VLSI implementation of a low-power logarithmic converter," *IEEE Trans. Comput.*, vol. 52, no. 11, pp. 1421–1433, Nov. 2003.

[22] R. C. Li, "Near optimality of Chebyshev interpolation for elementary function computations," *IEEE Trans. Comput.*, vol. 53, no. 6, pp. 678–687, Jun. 2004.

[23] A. G. M. Strollo, D. De Caro, and N. Petra, "Elementary functions hardware implementation using constrained piecewise polynomial approximations," *IEEE Trans. Comput.*, vol. 60, no. 3, pp. 418–432, Mar. 2011.

[24] B.-G. Nam, H. Kim, and H.-J. Yoo, "Power and area-efficient unified computation of vector and elementary functions for handheld 3-D graphics systems," *IEEE Trans. Comput.*, vol. 57, no. 4, pp. 490–504, Apr. 2008.

[25] Circuits Multi-Projects, Grenoble Cedex, France [Online]. Available: http://cmp.imag.fr

[26] P. Bogdan and R. Marculescu, "Non-stationary traffic analysis and its implications on multicore platform design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 508–519, Apr. 2011.

[27] R. Marculescu, U. Y. Ogras, L.-S. Peh, N. E. Jerger, and Y. Hoskote, "Outstanding research problems in NoC design: System, microarchitecture, and circuit perspectives," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 1, pp. 3–21, Jan. 2009.

[28] P. P. Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. Comput.*, vol. 54, no. 8, pp. 1025–1040, Aug. 2005.

[29] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks. An Engineering Approach*. San Francisco, CA: Morgan Kaufmann, 2003, ch. 9.

[30] D. Hilbert, "Über die stetige abbildung einer linie auf ein Flächenstück," *Math. Annal.*, vol. 38, no. 3, pp. 459–460, 1891.

[31] S. Seal and S. Aluru, "Chapter 44: Spatial domain decomposition methods for parallel scientific computing," in *Handbook of Parallel Computing: Models, Algorithms and Applications* (Computer and Information Science Series), S. Rajasekaran and J. Reif, Eds. Boca Raton, FL: Chapman and Hall/CRC, 2007.

[32] O. R. P. Bininda-Emonds, M. Cardillo, K. E. Jones, R. D. E. MacPhee, R. M. D. Beck, R. Grenyer, S. A. Price, R. A. Vos, J. L. Gittleman, and A. Purvis, "The delayed rise of present-day mammals," *Nature*, vol. 446, pp. 507–512, Mar. 2007.

Turbo Majumder (S'11) received the B.Tech. (hons.) degree in electronics and electrical communication engineering and the M.Tech. degree in automation and computer vision, both from the Indian Institute of Technology Kharagpur, Kharagpur, India, in 2005. He is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science, Washington State University, Pullman.

Prior to joining the Ph.D. program, he was with Freescale Semiconductor, Bangalore, India, and Nvidia Graphics, Bangalore. His current research interests include networks-on-chip and multicore systems-on-chip design for biocomputing applications, very large scale integration design, and parallel and high-performance computer architectures.

Michael Edward Borgens received the B.S. degree in computer engineering from Washington State University, Pullman, in 2011.

After graduation, he became a Component Design Engineer with Intel Corporation, Dupont, WA.

Partha Pratim Pande (SM'11) received the M.S. degree in computer science from the National University of Singapore, Singapore, and the Ph.D. degree in electrical and computer engineering from the University of British Columbia, Vancouver, BC, Canada.

He is currently an Associate Professor with the School of Electrical Engineering and Computer Science, Washington State University, Pullman. His current research interests include novel interconnect architectures for multicore chips, on-chip wireless communication networks, and hardware accelerators for biocomputing. He has around 50 publications on this topic in reputed journals and conferences.

Dr. Pande currently serves in the editorial boards of IEEE DESIGN AND TEST OF COMPUTERS and *Sustainable Computing: Informatics and Systems*. He is a Guest Editor of a special issue on sustainable and green computing systems for the *ACM Journal on Emerging Technologies in Computing Systems*. He serves in the program committees of many reputed international conferences.

Ananth Kalyanaraman (M'06) received the Bachelors degree from the Visvesvaraya National Institute of Technology, Nagpur, India, in 1998, and the M.S. and Ph.D. degrees from Iowa State University, Ames, in 2002 and 2006, respectively.

He is currently an Assistant Professor with the School of Electrical Engineering and Computer Science, Washington State University (WSU), Pullman. He is an Affiliate Faculty Member with the WSU Molecular Plant Sciences Graduate Program and with the Center for Integrated Biotechnology, WSU. The primary focus of his work has been on developing algorithms that use high-performance computing for data-intensive problems originating from the areas of computational genomics and metagenomics. His current research interests include high-performance computational biology.

Dr. Kalyanaraman received the 2011 DOE Early Career Award and two conference Best Paper Awards. He was the Program Chair for the IEEE HiCOMB 2011 Workshop and regularly serves on a number of conference program committees. He has been a member of the Association for Computing Machinery, since 2002, the IEEE Computer Society, since 2011, and the International Society for Computational Biology, since 2006.