2011 23rd International Symposium on Computer Architecture and High Performance Computing

Accelerating Maximum Likelihood based Phylogenetic Kernels using Network-on-Chip

Turbo Majumder, Partha Pande, Ananth Kalyanaraman
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA, USA
{tmajumde, pande, ananth}@eecs.wsu.edu

***Abstract*—Probability-based approaches for phylogenetic inference, like Maximum Likelihood (ML) and Bayesian Inference, provide the most accurate estimate of evolutionary relationships among species. But they come at a high algorithmic and computational cost. Network-on-chip (NoC), being an emerging paradigm, has not been explored yet to achieve fine-grained parallelism for these applications. In this paper, we present the design and performance evaluation of an NoC architecture for RAxML, which is one of the most widely used ML software suites. Specifically, we implement the top three function kernels that account for more than 85% of the total run-time. Simulations show that through novel core design, allocation and placement strategies our NoC-based implementation can achieve function-level speedups of 388x to 786x and system-level speedups in excess of 5000x over state-of-the-art multithreaded software.**

Keywords - Network-on-Chip; phylogeny reconstruction; hardware accelerator; multi-core.

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. Most approaches for phylogenetic reconstruction are based on Neighbor Joining (NJ), Maximum Parsimony (MP), Maximum Likelihood (ML) and Bayesian Inference (BI). The latter two are generally believed to be the most accurate as they provide a statistical likelihood score for each tree using the Phylogenetic Likelihood Function (PLF) [1, 2]. The boost in quality, however, comes at a high computation cost as ML and BI formulations are NP-Hard [3] and suffer from the need to explore an exponential (in *n*) number of trees. For example, a run using RAxML [4], which is one of the most widely-used programs to compute ML-based phylogeny, on an input comprising of 1,500 genes can take up to 2.25 million CPU hours [5]. Therefore, they need to rely on algorithmic heuristics and high-performance computing for achieving practical solutions. The increasing availability of genomic data has only exacerbated this situation. In addition to algorithmic complexity, ML approaches also involve a significant amount of floating-point computations.

In this paper, we propose the use of a Network-on-Chip (NoC) based multi-core platform to address the issue of computation intensity for ML. The motivation for using an NoC to address the ML application stems from the fact that there are different levels of parallelism in the application that can be exploited to accelerate the 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 FPGA board that supports several computation cores (e.g., similar to [6]), an NoC based multi-core system can handle coarse-grained parallelism more efficiently. 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 at any instant, there could be variable number of instances of each function running simultaneously. Under the NoC framework, these requirements can be effectively addressed by (a) designing a homogeneous system, where different PEs are able to seamlessly support different functions executing at different times, and (b) interconnecting them using a suitable network that allows concurrent execution of arbitrary combinations of function instances and provides the backbone for efficient data exchange between individual PEs. In other words, we can build a heterogeneous application map on 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.

We implemented an NoC and tested its acceleration performance using three function kernels in RAxML. Based on our experiments on practical input data, these kernels collectively account for more than 85% of the total run-time, and involve several computations of logarithm, antilogarithm and sum of products. The novelty of our approach stems from an efficient, fine-grained parallel implementation of floating-point arithmetic to optimize computational run-time and novel core allocation schemes to minimize inter-core communication latency. Experimental results through simulations demonstrate

1550-6533/11 $26.00 © 2011 IEEE
DOI 10.1109/SBAC-PAD.2011.17

system-level speedups of ~5000x for system size 64 over software, on these function kernels.

II. Related Work

Of the different approaches for phylogenetic tree reconstruction, Maximum Likelihood (ML), Maximum Parsimony (MP) and Bayesian Inference (BI) are the more compute-intensive as well as accurate methods [7]. Considerable work has been done on designing hardware accelerators for these different approaches of phylogenetic computation. However, there exists no prior work on accelerating Maximum Likelihood (ML) kernels using the NoC platform, which is the subject of this paper.

Acceleration of breakpoint phylogeny, which is based on Maximum Parsimony, is the topic of [8] and [9]. The primary computational characteristic of breakpoint phylogeny is the computation of an optimal solution for the Traveling Salesman Problem. In [8], Bakos and Elenis implement a parallelized version for breakpoint-median phylogeny using both software and FPGA and achieve a speedup of 417x for a whole-genome phylogenetic reconstruction. In [9], we implemented an NoC-based accelerator that achieves a speedup of 774x on genome sizes comparable to those used in [8].

For BI, hardware acceleration has been proposed using Cell Broadband Engine (CBE), GPU and general purpose multi-cores [10], and FPGA [6]. 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 [11]. A CBE-based implementation [12] for RAxML is shown to outperform the software by 2x. Another implementation of RAxML [5] using FPGA boards with built-in DSP slices achieves a speedup of 8x.

In this paper, we explore the suitability and merits of using the NoC paradigm for ML phylogeny. 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, our implementation is the first NoC-based accelerator for ML phylogeny.

A key component of ML phylogeny is log-likelihood computation. Fast calculation of logarithms in hardware has been a well-researched topic. Kwon et al [13] describe a fast implementation of exponentiation in hardware targeting graphics applications. A 32-bit binary-to-binary linear approximation-based logarithm converter is described in [14]. Optimality of Chebyshev polynomials for table-based approximations of elementary functions is described in [15]. A technique for designing piecewise polynomial interpolators for implementing elementary functions such as logarithm and exponentiation in hardware is described in [16]. A unified computation architecture for calculating elementary functions is presented in [17], 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.

III. NoC Design

The design of the NoC constitutes designing the homogeneous computation core, the interconnect network,

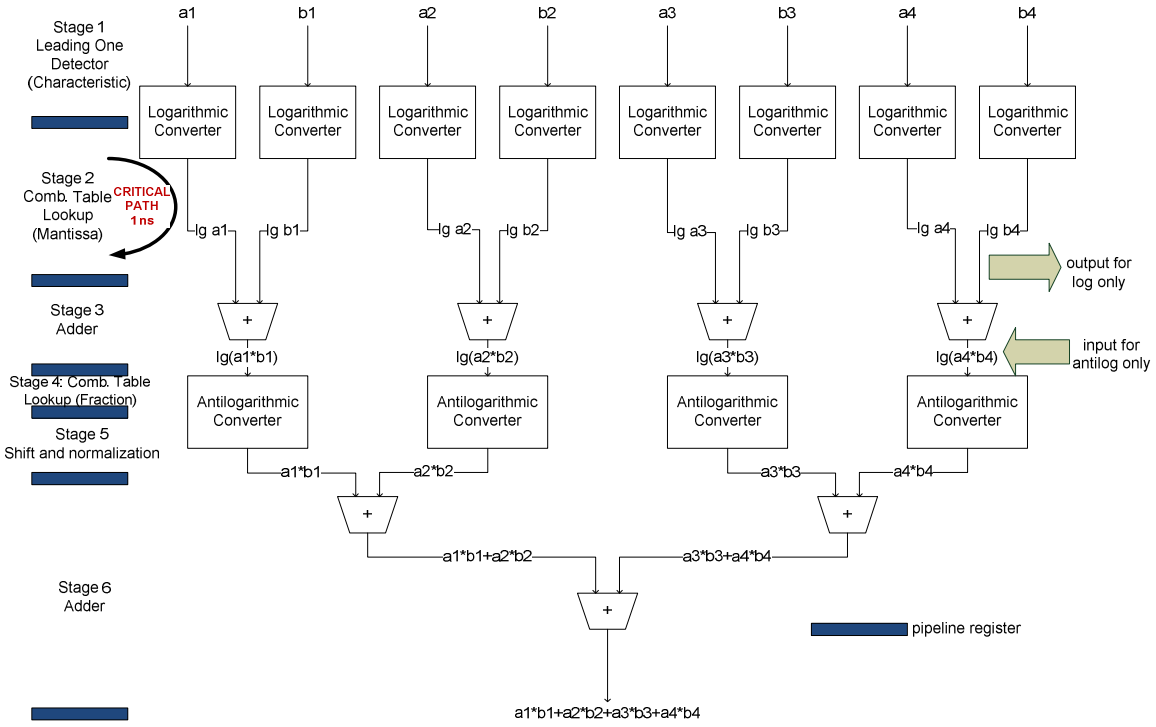

Figure 1. Architecture of computational core for sum-of-products, logarithm and antilogarithm

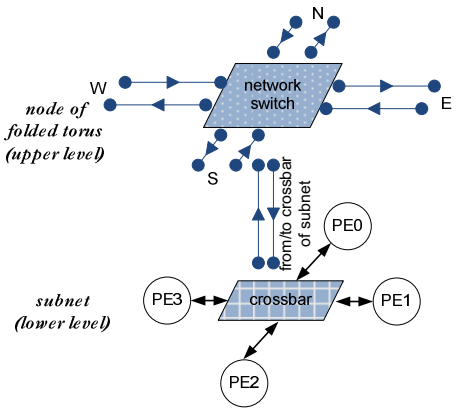

Figure 2. Network switch of NoC and the crossbar-connected subnet

application mapping, and routing and arbitration policy.

A. Computation Core

The core aims to capture the crux of the computation involved in phylogenetic kernels. We address this by combining fast and efficient computation strategies in hardware with extensive fine-grained parallelism. The architecture has six pipeline stages and can compute a sum of four products during one traversal across the stages, in addition to regular logarithm and exponential. Logarithm and exponential are directly computed using linear table-based approximations as in [17]. These functions are entirely implemented using logic gates, without using a ROM. For sum of products, an indirect approach is used. Logarithms of four pairs of numbers are taken, each log value in a pair is added to the other, four antilogarithms are taken and the four results are added together. In other words, multiplication is done by addition in the log-domain and addition is done in the linear domain. Figure 1 shows a schematic diagram of the computation architecture. Note that if we are interested in only computing the logarithm, the adder stage is not required and the core provides the result of the stage 2 as output depending on the instruction being executed. For computing exponential, the input goes directly to stage 4 with modifications in the number representation format and the output is available at the end of stage 5. The three representative functions of the RAxML suite that we used, namely *coreGTRCAT*, *newviewGTRCAT* and *newviewGTRGAMMA* [18], 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. We implemented the design with Verilog HDL, and synthesized it with a clock frequency of 1 GHz using 65 nm standard cell libraries from CMP [19]. The critical path delay is 1 ns as shown in Figure 1.

B. NoC Node

The core with a wrapper is designated a processing element (PE). Four such PEs are integrated to form one subgroup. The PEs are designated *PE0*, *PE1*, *PE2* and *PE3*. A crossbar switch, shown in Figure 2, connects the four PEs and coordinates communication among them based on the instruction type. The crossbar switch has to deal with three kinds of traffic. The first and simplest kind involves sending data received from *PEx* back to *PEx*. The second kind of communication involves sending data from *PEx* to *PEx*, *PEy*, *PEz* and *PEw*, where *x*, *y*, *z* and *w* are all distinct and each is one of 0, 1, 2 and 3. The third kind of communication involves sending/receiving traffic to/from the external network through a network switch. This cluster of four PEs is considered a *subnet* under one *NoC node*.

C. Network

A mesh is a highly scalable network architecture whose regularity provides for easier timing closure and reduces dependence on interconnect scalability [20]. A folded torus further reduces the separation between nodes at the boundaries of a mesh by cutting down the diameter of the network by half without compromising on the regularity or scalability of the entire network. 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, for which a mesh/folded torus is more suitable than other network topologies, such as a ring or star. We use folded torus NoCs of size 4×4 (N=16) and 8×8 (N=64). Since each subnet associated with a node has 4 PEs, our system has 64 and 256 PEs respectively. A network switch handles traffic emanating from or destined to each network node. We use the switch described in [21] for our design. Each switch has four ports to neighboring switches and one port to the crossbar switch of the subnet (Figure 2).

The primary data contained in the messages exchanged among nodes are intermediate function results. These are 64-bit numbers using FXP-HNS format [17]. As such, the message size is very small, and we split the message into 3 flits (header, body and tail). Since very deep buffers may slow down clock frequency and do not appreciably improve performance for short messages [22], we use buffer depth of 2 flits. We adopt the routing and arbitration mechanism from [21].

D. Node Allocation

As mentioned earlier, the three representative functions (phylogenetic kernels) from RAxML [19] we used are *newviewGTRCAT*, *coreGTRCAT* and *newviewGTRGAMMA*. We parallelize each function by breaking down larger computation arrays into smaller units and by utilizing loop-level parallelism, such that each function requires 2, 3 and 6 nodes respectively. Taking *newviewGTRCAT* as an example, we can see from Figure 3 that computation of the *x3* array in each iteration requires computation of eight sums of four products each, using arrays *left*, *x1*, *x2* and the eigenvalue vector *EV[15:0]*. Since each PE can compute the sum of four products, 8 PEs (or equivalently, two NoC nodes) are required. The other functions have a similar computation footprint and

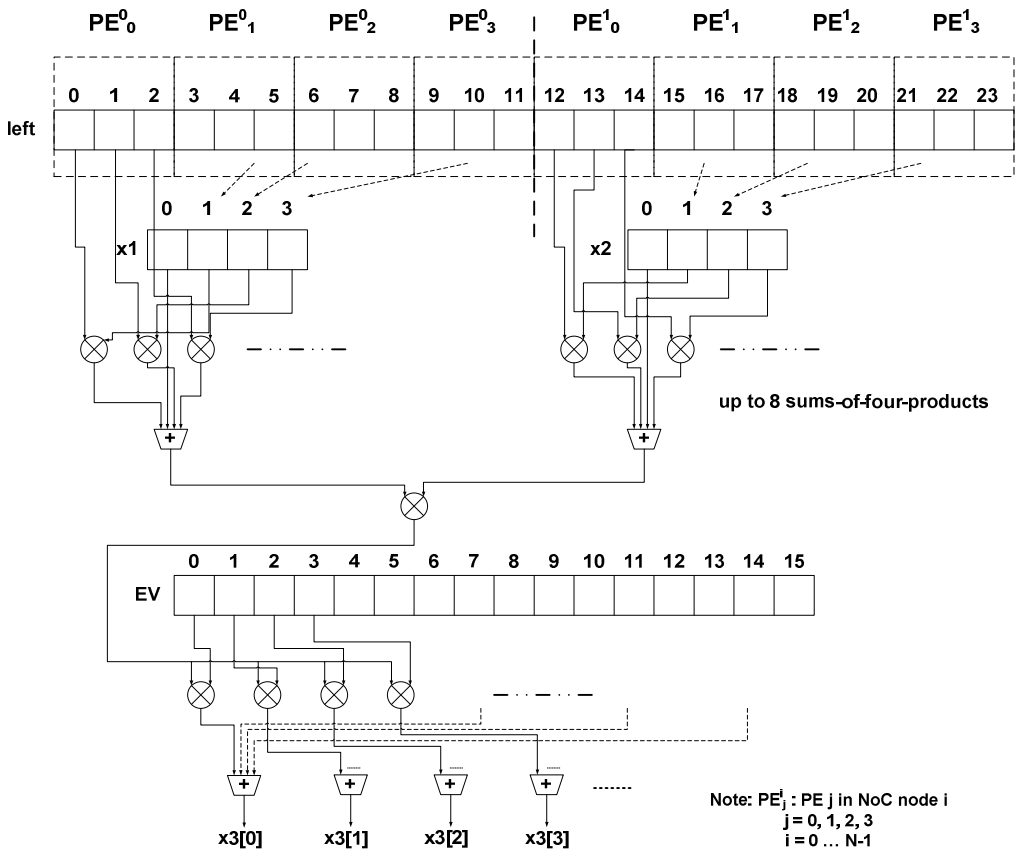

Figure 3. Part of the computation tree of *newviewGTRCAT*

parallelization details are omitted due to space constraints. A node is *busy* when the subnet under it is performing computations pertaining to a function; otherwise it is *available*. There is a centralized controller (*MasterController*) that allocates a set of nodes from the set of available nodes on the NoC to a requesting 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 one function are said to belong to one *partition*. Nodes are dynamically allocated and hence can be reused in a new partition after the function executing in the current partition terminates.

Since the nodes are connected with a folded torus and they need to communicate with one another within the partition, it is desired that they be close together to reduce the number of hops required for data exchange and thereby reduce the communication latency associated with the function. Hence, an intelligent node allocation policy is required. One of the ways of achieving good clustering of allocated nodes is to have breadth-first search (BFS) among *available* nodes, where adjacency between nodes is defined on the basis of their location on the folded torus. Although this appears to be an optimum allocation strategy, the drawbacks are as follows: First, this requires the first node chosen to be within a neighborhood of a sufficient number of *available* nodes when such number is actually available in the system. More importantly, since a *MasterController* handles node allocation, it has to traverse each node in the neighborhood (in the adjacency list of the parent node) in turn to grow the partition. Scanning the entire adjacency list of each node takes one cycle and growing the partition fully requires a number of cycles equal to the number of nodes requested by the function (2, 3 or 6). This latency comes without any major benefit as we shall see shortly and also in the experimental results presented in Section 4.

Our approach uses the Hilbert curve [23] for node allocation. A Hilbert curve is a locality-preserving space-filling curve popular in scientific computing. It has an added property that when mapped to a regular mesh or folded torus, nodes adjacent along a 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. This converts a two-dimensional allocation problem to a one-dimensional problem. We use a fixed Hilbert curve embedded on a folded torus as shown in Figure 4 (for N=16), where there is a fixed correspondence between the node numbers on the torus and those on the Hilbert curve to effectively predetermine the set of possible nodes for allocation. Since this information is hard-wired in the design and no run-time decision needs to be taken, a set of *available* nodes can be

allocated to a function in one cycle (for N=16) or four cycles (for N=64). Note that this may lead to three situations, shown in Figure 4. First, allocated nodes are all adjacent to each other or *contiguous* on Hilbert and hence the *partition* is *contiguous* on the torus. Second, the nodes are *non-contiguous* on Hilbert but form a *contiguous partition* on the torus. Third, the nodes are *non-contiguous* both on Hilbert and on the torus.

E. Routing and Arbitration

The routing policy we use is based on XY routing on folded torus. Nodes belonging to the same partition need to communicate with one another during the execution of the function. In our system, we distinguish messages originating from contiguous and non-contiguous partitions. We designate the corresponding flits 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 non-contiguous 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 non-contiguous partition is free to move in any direction 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 while a B-type message faces no such restriction. Since B-types 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 have to switch between XY and YX routing based on the availability of routing channels at each switch. Figure 5 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 outside the partition and makes use of the torus loopback.

It is clear 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 non-contiguous partition(s). On the other hand, switches internal to a non-contiguous partition will face only B-type traffic from non-contiguous partition(s). When there is more than one message competing for the same port, the following arbitration policy is used. The remaining hop count of the message is determined by calculating the 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. This policy ensures that traffic with a higher potential latency is routed earlier, thereby reducing worst-case latency.

Since B-type messages follow XY routing (on torus), any non-contiguous partition never experiences deadlock. For contiguous partitions of sizes 2 and 3, there is no possibility of a cycle in the channel dependency graph because the message is always contained within the partition and 2 or 3 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 Figure 5 or a partition

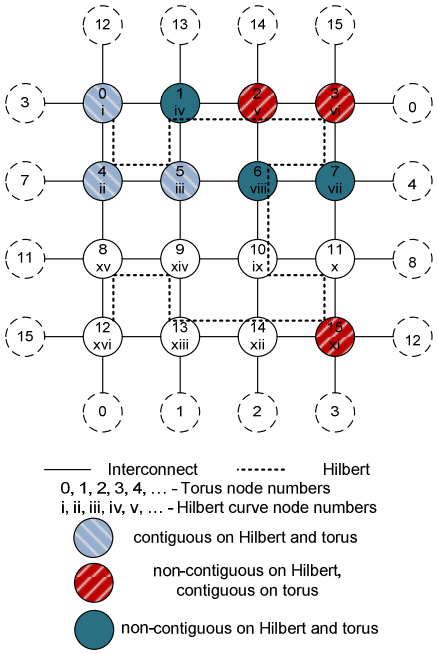

Figure 4. Hilbert curve embedded in the folded torus network architecture and different kinds of contiguous and non-contiguous partitions.

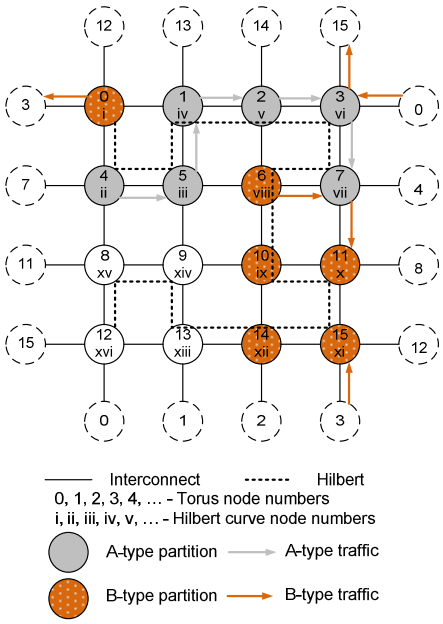

Figure 5. Examples of different routing methods for A-type (contiguous) and B-type (non-contiguous) partitions

comprising of nodes 8, 9, 10, 12, 13 and 14 in Figure 4. In the former case, we do not have a cycle. In the latter case, we follow strict XY routing. Hence, no deadlock occurs in this scenario as well. Therefore, our routing and arbitration policy 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 switches and *MasterController* with 65 nm standard cell libraries from CMP [19]. We simulated NoCs with system sizes N=16 and N=64 using the NoC simulator used in [21].

We ran RAxML-VI-HPC (version 7.0.4) [18] on DNA sequences supplied with the suite (originally derived from a 2,177-taxon 68-gene mammalian dataset described in [24]) in single and multi-threaded modes on a Pentium IV 3.2 GHz dual-core CPU and used the best software run-times 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 showed that the functions *coreGTRCAT* (48%), *newviewGTRGAMMA* (21%) and *newviewGTRCAT* (17%) collectively account for more than 85% of the total run-time. The average time spent in each invocation of a function is also noted.

B. Test Case Design

Allocation of nodes to each function is based on the policy in Section III D. We simulate a range of test case configurations for both system sizes (16 and 64), where different nodes of the NoC are allocated for several functions resulting in a mix of contiguous and non-contiguous partitions. Test cases have been designed to capture 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. Communication of the result from the previous step occurs simultaneously with the current step computation taking place in the PEs of the subnet under the node. This allows masking of communication latency by computation delay. It is observed that *newviewGTRCAT* requiring 2 nodes per invocation is generally computation-intensive, while *coreGTRCAT* and *newviewGTRGAMMA* requiring 3 and 6 nodes respectively are generally communication-intensive.

C. Communication Latency

The histograms of the communication latencies involved in each function (in terms of number of cycles) for N=16 are shown in Figure 6. As expected, the best-case latency is predominantly achieved for *coreGTRCAT* and *newviewGTRCAT*; whereas for *newviewGTRGAMMA*, the number of partitioning possibilities is relatively low in a 16-node system. Hence it appears that all latencies are evenly distributed. While it can be expected that non-contiguous partitioning leads to latency dilation, it is not readily apparent that latency of a particular function partition is also dependent on the overlapping traffic from neighboring B-type partitions. Referring to Figure 7, each B-type partition has to use the link between nodes 13 and 14. Each of these partitions has a total communication latency of 144 cycles and is marked "α" on the histogram in Figure 6(a). Note that most *newviewGTRCAT* partitions experience a communication latency of 96 cycles. Another example is

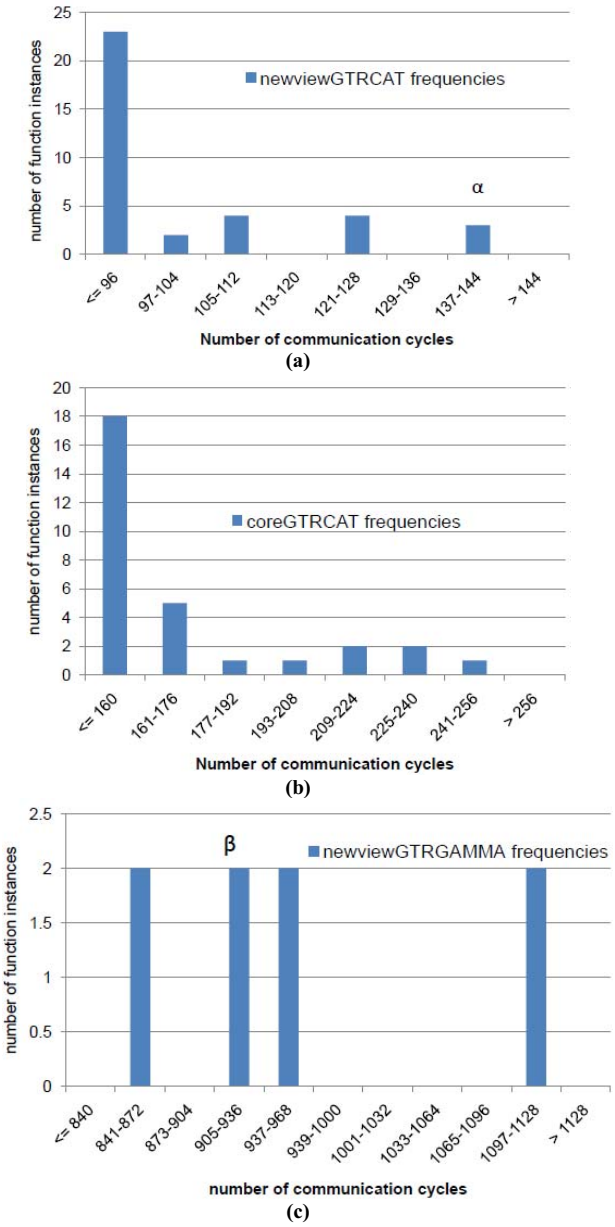

Figure 6. Histograms of communication latencies for (a) *newviewGTRCAT*, (b) *coreGTRCAT* and (c) *newviewGTRGAMMA* on NoC of system size, N=16

shown in Figure 8, where a non-contiguous partition does

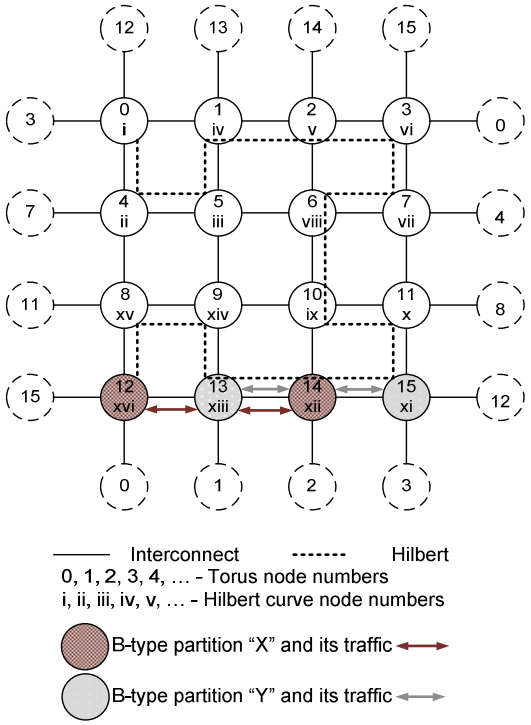

Figure 7. Overlapping traffic from non-contiguous partitions severely affecting latency

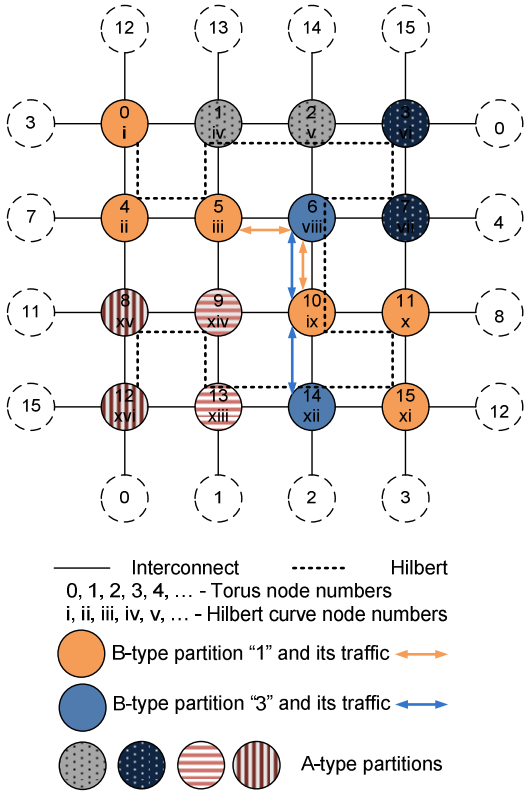

Figure 8. Orthogonal traffic between non-contiguous partitions cause no major dilation of latency

not suffer appreciable latency dilation due to orthogonal traffic patterns from partitions "1" and "3". This is marked by the point "β" in Figure 6(c). Note that this communication latency lies in the lower half of the range of latencies observed for *newviewGTRGAMMA*.

D. Speedup

We used two different kinds of metrics to evaluate the acceleration performance of our design. The first metric is function-level speedup, which is a measure of the strength of our PE architecture because it provides the level of fine-grained parallelism we are able to achieve. The second metric is system-level speedup, which is a measure of the acceleration that is achieved by using coarse-grained parallelism by embedding PEs in the NoC framework.

Function-level speedup. In order to estimate function-level speedup, the total function execution time, consisting of computation and communication components, was averaged over all scenarios and compared with the baseline function execution time mentioned earlier. The speedup obtained for each function is shown in Figure 9. Note that the best speedup of 786x is obtained for *coreGTRCAT*, which accounts for 48% of the total software run-time. The least speedup of 388x is obtained for *newviewGTRCAT*, because it is the smallest function kernel and requires two NoC nodes (or 8 PEs) by design. Since the function-level speedup is more dependent on the PE architecture, it is less affected by system size, *N*.

System-level speedup. System-level speedup is calculated

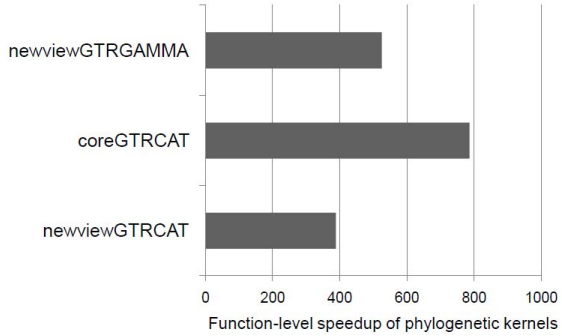

Figure 9. Function-level speedup of phylogenetic kernels

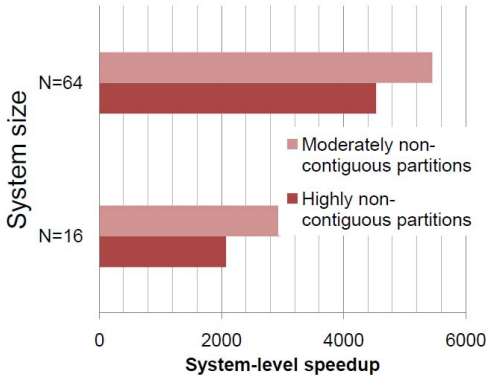

Figure 10. System-level speedup for different test-case configurations on NoCs with *N*=16 and *N*=64

as the ratio of run-times for test-case configurations containing a mix of all three functions. Each test-case configuration represents a typical snapshot of the system during the course of execution of RAxML, with our NoC-based system handling the phylogenetic kernels. Several instances of *newviewGTRGAMMA*, *coreGTRCAT* and *newviewGTRCAT* occupying contiguous and non-contiguous partitions are present in each such test-case. Allocation overhead in *MasterController* is accounted for. System-level speedup is indicative of the amount of speedup that can be observed on different runs on RAxML. The test-cases used for evaluating system-level speedup were binned into two categories – one comprising of partitions having moderate fragmentation, i.e. fewer non-contiguous partitions, and the other comprising of a large number of non-contiguous partitions. These two categories represent *optimistic* (average speedup: 5455x for N=64 and 2963x for N=16) and *pessimistic* (average speedup: 4534x for N=64 and 2075x for N=16) fragmentation scenarios as shown in Figure 10. Any typical realistic scenario is likely to have a speedup within these bounds. Also as Figure 10 shows, system-level speedup increases for higher system size. This is due to the fact that a larger system size can accommodate more instances of each function running in parallel. Furthermore, it can be observed that the speedup increases by ~2x from N=16 to N=64 instead of ~4x. This is due to higher levels of scattering of the allocated nodes and the increased node-allocation time for N=64. The latter is due to higher partition allocation time per function and larger number of functions in one test-case configuration.

These simulation-based studies show the high potential of NoC to speed up compute-intensive phylogenetic kernels by orders of magnitude, considerably higher than other existing accelerator platforms, paving the way for future prototyping.

V. Conclusion and Future Work

We demonstrate function-level speedup of up to 786x and system-level speedup of over 5000x for phylogenetic kernels through the design of an NoC-based multi-core platform and hierarchical parallelism. Each core is intrinsically able to exploit the fine-grained parallelism inherent in the computations related to these applications. We propose a fast node allocation scheme that optimizes communication latency. Although we have chosen three functions from RAxML, the approach is general and can be extended to other similar applications in most probability-based phylogenetic inference methods.

References

[1] Felsenstein, J. 1981. Evolutionary trees from DNA sequences: A maximum likelihood approach. In *J. Molecular Evolution 17*, pp. 368-376.

[2] Felsenstein, J. 2004. *Inferring Phylogenies.* Sinauer Associates, Inc.

[3] Chor, B. and Tuller, T. 2005. Maximum Likelihood of Evolutionary Trees: Hardness and Approximation. *Bioinformatics*, vol. 21(1), pp. 97-106.

[4] Stamatakis, A. 2006. RAxML-VI-HPC: Maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*.

[5] Alachiotis, N. et al. 2009. Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function. In *Proc. IEEE Intl. Sym. on Parallel and Distributed Processing*, pp 1-8.

[6] Zierke, S. and Bakos, B. 2010. FPGA Acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. In *BMC Bioinformatics*,11:184

[7] Bader, D. A. and Yan, M. 2005. High-Performance Phylogeny Reconstructio. In *Handbook of Computational Molecular Biology*, ed. S. Aluru. Chapman & Hall/CRC Computer and Information Science Series.

[8] Bakos, J. and Elenis, P. 2008. A Special-Purpose Architecture for Solving the Breakpoint Median Problem. In *IEEE Trans.VLSI Systems* vol. 16, pp. 1666-1676.

[9] Majumder, T. et al. 2010. An Optimized NoC Architecture for Accelerating TSP Kernels in Breakpoint Median Problem. In *Proc. IEEE Intl. Conf. Application-specific Systems, Architectures and Processors*, pp. 89-96.

[10] Pratas, F. et al. 2009. Fine-grain Parallelism using Multi-core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function. In *Proc. IEEE Int. Conf. Parallel Processing*, pp. 9-17.

[11] Mak, T. S. T. and Lam, K. P. 2003. High Speed GAML-based Phylogenetic Tree Reconstruction Using HW/SW Codesign. In *Proc. Comp. Systems Bioinformatics*, pp. 470.

[12] Blagojevic, F. et al. 2007. RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Emgine. In *Proc. IEEE International Symposium on Parallel and Distributed Processing*, pp. 1-10.

[13] Kwon, Y. S. et al. 2000. A Hardware Accelerator for the Specular Intensity of Phong Illumination Model in 3-Dimensional Graphics. In *Proc. Asia and South Pacific Design Automation Conf.*, pp. 559-564.

[14] Abed, K. H. and Siferd, R. E. 2003. CMOS VLSI Implementation of a Low-Power Logarithmic Converter. In *IEEE Trans. Computers*, vol. 52, no. 11, pp. 1421-1433.

[15] Li, R.C. 2004. Near Optimality of Chebyshev Interpolation for Elementary Function Computations. In *IEEE Trans. Computers*, vol. 53, no. 6, pp. 678-687.

[16] Strollo, A. et al. 2010. Elementary Functions Hardware Implementation Using Constrained Piecewise Polynomial Approximations. In *IEEE Trans. Computers*.

[17] Nam, B. G. et al. 2008. Power and Area-Efficient Unified Computation of Vector and Elementary Functions for Handheld 3D Graphics Systems. In *IEEE Trans. Computers*, vol. 57, no. 4, pp. 490-504.

[18] http://www.exelixis-lab.org (Last accessed 2 July 2011).

[19] http://cmp.imag.fr (Last accessed 2 July 2011).

[20] Marculescu, R. et al. 2009. Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. In *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(1): 3-21.

[21] Pande, P. P. et al. 2005. Performance Evaluation and Design Trade-Offs for Network-on-Chip Interconnect Architectures. In *IEEE Trans. Computers*, vol. 54, no. 8, pp. 1025-1040.

[22] Duato, J., Yalamanchili, S. and Ni, L. 2003. *Interconnection Networks. An Engineering Approach*. Ch. 9. Morgan Kaufmann Publishers.

[23] Hilbert, D. 1891. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen* 38, pp. 459-460.

[24] Bininda-Emonds, O. R. P. et al. 2007. The delayed rise of present-day mammals. In *Nature* 446: 507-512.