NoC-Based Hardware Accelerator for Breakpoint Phylogeny

Turbo Majumder, *Student Member*, *IEEE*, Souradip Sarkar, *Member*, *IEEE*, Partha Pratim Pande, *Senior Member*, *IEEE*, and Ananth Kalyanaraman, *Member*, *IEEE*

Abstract—Maximum Parsimony phylogenetic tree reconstruction is based on finding the breakpoint median, given a set of species, and is represented by a bounded edge-weight graph model. This reduces the breakpoint median problem to one of solving multiple instances of the Traveling Salesman Problem (TSP), which is a classical NP-complete problem in graph theory. Exponential time algorithms that apply efficient runtime heuristics, such as branch-and-bound, to dynamically prune the search space are used to solve TSP. In this paper, we present the design and performance evaluation of a network-on-chip (NoC)-based implementation for solving TSP under the bounded edge-weight model, as used in the computation of breakpoint phylogeny. Our approach takes advantage of fine-grain parallelism from the multiple processing elements (PEs) and uses efficient NoC architecture for inter-PE communication. To accelerate the application on hardware, our PE design optimizes a particular lower bound calculation operation which typically tends to be the serial bottleneck in computation of a TSP solution. We also explore two representative NoC architectures—mesh and quad-tree—and show that the latter is more energy-efficient for this application domain. Experimental results show that this new implementation is able to achieve speedups of up to three orders of magnitude over state-of-the-art multithreaded software implementations.

Index Terms—Phylogenetics, breakpoint-median problem, maximum parsimony, traveling salesman problem.

1 Introduction

Phylogenetics is the study of evolutionary relationships among organisms based on their underlying genetic content. The term *genome* is a collective reference to all the DNA in the living cell of an organism and phylogenetic tree construction is the process of building an evolutionary tree based on the similarities and differences observed among the genomic DNA of a set of species. It is a fundamental problem in computational molecular biology with important applications that include drug discovery. In this tree, the leaves represent species (known) and the internal nodes represent common ancestral species (unknown). Until a decade ago, only a handful of genome sequences were available, and therefore, the knowledge regarding evolutionary trees was limited. However, with the recent advances in DNA sequencing technologies, sequence information for more than a thousand species is now available in public databases and more large-scale sequencing efforts are currently underway. Owing to this deluge in genomic information, the computational biology community has embarked on a project called the Tree of Life, which is an ambitious project to construct the evolutionary tree connecting all known species. The single largest impediment to this project is, however, the high computational costs associated with building phylogenetic trees [1].

The process of inferring the phylogeny of a set of k *taxa* (or species) entails reconstructing a phylogenetic tree based on distance or probability measures [2]. When the relative ordering of genes on a genome is known, a specific type of phylogeny called the breakpoint phylogeny can be computed, based on the *breakpoint distance*. Given a reference set of m genes $\{g_1, g_2, \ldots, g_m\}$, any genome can be represented by an ordering of the subset of genes that constitute it, as they appear from end to end of the genomic DNA. The *breakpoint distance* between any two genomes is defined as the number of gene pairs that appear adjacent in one genome but not in the other. It is a measure of how different two genomes are by their gene ordering. Blanchette et al. pioneered the work on breakpoint-based phylogeny [3]. They reduced the problem of constructing an optimal phylogenetic tree of N genomes to one of solving numerous instances of a version of the Traveling Salesman Problem (TSP) [4] where edge weights of the input graph are bounded to a fixed set of integer values. Put intuitively, each instance of TSP tries to identify the gene order of a hypothetical ancestral genome that is the closest representative to any three given genomes. This problem is called the *3-median breakpoint problem* and has been proven to be NP-Hard [5]. A software suite called Genome Rearrangements Analysis under Parsimony and other Phylogenetic Algorithms (GRAPPA) [6] computes an exhaustive search across all possible trees ($\equiv 3^*5^*7^* \ldots {}^*(2k-5)$ trees) and iteratively runs multiple instances of a TSP solver for scoring each tree. Given the large number of trees to evaluate, phylogenetic reconstruction can easily become heavily compute-intensive—taking days to weeks of compute time—for even a modest number of taxa and genes.

- *T. Majumder, P.P. Pande, and A. Kalyanaraman are with School of Electrical Engineering and Computer Science, Washington State University, EME 102, Spokane St., Pullman, WA 99164. E-mail: {tmajumde, pande, ananth}@eecs.wsu.edu.*
- *S. Sarkar is with Intel Exascience Labs (Intel Labs Europe), Kapeldreef 75, Leuven 3001, and ELIS, Universiteit Gent, Belgium. E-mail: ssarkar@elis.ugent.be.*

Manuscript received 25 June 2010; revised 24 Jan. 2011; accepted 24 Apr. 2011; published online 17 May 2011.
Recommended for acceptance by R. Marculescu.
For information on obtaining reprints of this article, please send E-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-06-0359.
Digital Object Identifier no. 10.1109/TC.2010.100.

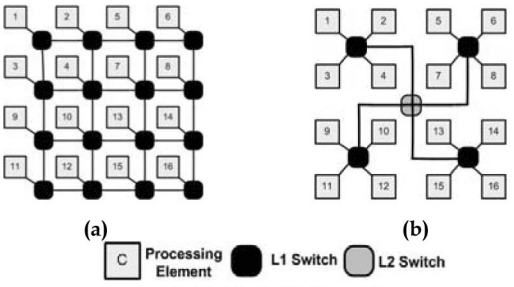

Figure 1. (a) Mesh NoC architecture
(b) Quad-tree NoC architecture

Fig. 1. (a) Mesh NoC architecture. (b) Quad-tree NoC architecture.

More importantly, over 99 percent of the total runtime gets typically spent in computing TSP instances [7].

TSP is a widely studied NP-complete problem for which several heuristics have been explored [8], [9], [10], [11], [12], [13], [14] and the *branch-and-bound*-based methods [14], [15] continue to be the most popular among accurate solvers, owing to their effectiveness in reducing the exponential search space. The runtime heuristic, which itself is computationally intensive, is an ideal candidate for parallelization. An array of processing elements (PEs) working in parallel on distinct parts of the solution would naturally enhance performance. However, these PEs cannot work completely in isolation and need to communicate among themselves. This communication needs to be efficient and synchronized with the computation operation of the PEs. To achieve this in an on-chip scenario, a platform possessing inherent fine-grained, large-scale parallelism and an efficient communication fabric needs to be chosen. A Network-on-Chip (NoC) provides the best fit to this requirement. On one hand, an NoC scales very well with increasing number of PEs; on the other hand, it offers the user the freedom to choose the communication architecture that is most apt for a target application.

The principal motivating application for this paper is accelerating the breakpoint median computation in Maximum Parsimony-based phylogenetic reconstruction. Each breakpoint median computation is an instance of the TSP. To solve the TSP for a graph with m vertices, a series of lower bound calculations is used, each of which could be implemented as a matrix reduction operation on the associated adjacency matrix. This operation has a time complexity of $O(m^2)$, and therefore, performing an overwhelming number (possibly exponential) of such reduction operations could severely limit the scalability of the computational kernel for large graphs. We have designed an application-specific PE that can perform the matrix reduction in $O(m)$ time, which in turn renders the entire computation to have a linear time complexity with graph size. Additionally, we explore two major NoC architectures—mesh, shown in Fig. 1a, and a 4-way hierarchical star or quad-tree, shown in Fig. 1b—and demonstrate the superiority of the latter for our application. This is based on a comparison of network latency and power consumption across the two frameworks.

The organization of the paper is as follows: Section 2 outlines prior work on accelerating phylogenetic applications through the use of hardware and development of heuristics for TSP. Section 3 describes the branch-and-bound algorithm for solving TSP that we have used as the basis of our hardware implementation. We describe the NoC design in detail in Section 4, separately outlining the design of the PEs, switches, network, communication protocol, and application mapping. Experimental results are presented in Section 5 for synthetic and real biological data. We conclude the paper in Section 6.

2 Related Work

Substantial work has been carried out in the field of hardware acceleration targeted toward phylogenetics applications. These accelerators have been based on platforms like FPGA, Graphics Processing Unit (GPU), Cell Broadband Engine (CBE), and general-purpose multicores (traditional Intel/AMD dual-core, quad-core platforms). A hybrid hardware/software implementation proposed in [16] using a Genetic Algorithm for Maximum Likelihood (GAML) approach reports a speedup of 30 over software. The phylogenetic likelihood function (PLF) has been accelerated around eight times through the use of FPGA boards with built-in DSP slices in [17]. A whole genome phylogenetic reconstruction based on a parallelized version of the breakpoint median algorithm has been shown in [7]. Using a combination of software and FPGA, total execution has been reduced by a factor of 417 over single-thread software implementation. FPGA-based acceleration up to a factor of $10\times$ has been demonstrated over software for Bayesian inference with MrBayes 3 tool in [18]. In [19], a comparative evaluation of a program for Bayesian inference of phylogenetic trees is presented. While CBE and GPU are shown to have appreciable reduction in computation time, they introduce significant communication time penalty. The general purpose multicores have overall better performance.

There has been another body of work on efficiently solving TSP. These algorithms can be classified into two groups—1) approximation algorithms that could take polynomial time [8], [9], [10], [11] and 2) accurate algorithms that run in exponential time [14], [15]. Techniques used in approximation methods include the Kernighan-Lin heuristics, simulated annealing, and genetic algorithms [8], [9], [10], [11], [12]. Among accurate methods, dynamic programming [15] is strictly exponential in practice, whereas branch-and-bound methods [14], [15] achieve significant pruning of search space during computation without affecting the optimality of the output. Coarse-level parallelization of TSP has been explored using genetic algorithms [12] and branch-and-bound [1], [13].

Currently, there are no custom multicore NoC architectures targeting TSP or phylogenetics. Here, we present the design of a custom NoC for TSP computations typical in phylogenetic applications and evaluate its performance.

3 Algorithm

In this section, we present the core computation steps of the branch-and-bound runtime heuristic to solve TSP [15] that we used in our implementation. The input is a directed graph, $G = (V, E)$ with m vertices and a nonnegative cost associated with each edge. The m vertices of this graph

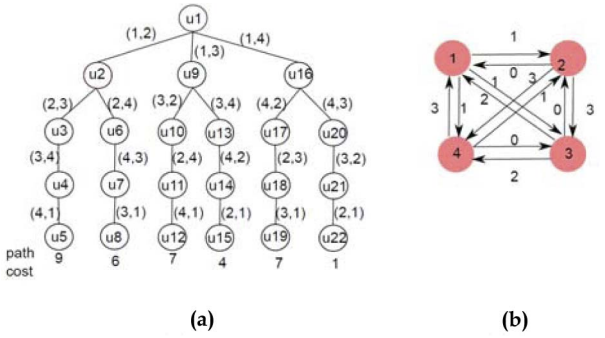

Fig. 2. An example showing (a) the exhaustive search tree corresponding to the input graph (b). If the tree is computed in the Depth-First-Search Order, then evaluation of the path that leads to a low cost (such as u1-u2-u6-u7-u8) first may help in pruning the computation of a higher cost path (such as u1-u9-u13-u14-u15). This idea is exploited in the branch-and-bound technique.

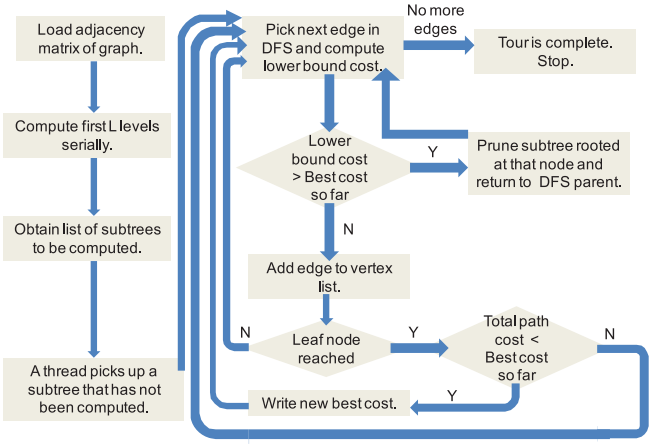

Fig. 3. Flow diagram explaining a branch-and-bound algorithm for solving the breakpoint median problem.

correspond to the m reference genes and its edges have a bounded weight—an integer cost between 0 and 3 or an edge with cost ∞ (representing nonexistent edges) [3]. The output is a least cost cyclic tour that traverses all vertices exactly once.

The overall algorithm has a worst case runtime complexity that is exponential in the number of vertices (i.e., genes). However, the use of a branch-and-bound technique reduces this exponential search space significantly for most practical inputs.

Given this input graph G, the solution space can be represented by a conceptual computation tree. An example is shown in Fig. 2. The tree has a total of *(m-1)!* potential paths to be explored before identifying the optimal TSP tour. Every tree edge (u, v) from a parent node u to a child node v corresponds to a graph edge $(i, j) \in E$, and every path from the root to a leaf node encodes a completed TSP tour with cost equal to the sum of the edge weights along its path. An optimal TSP tour represents a least cost path. Our algorithm dynamically generates and explores this conceptual search-space tree in the depth-first-search (DFS) order.

Initially, a global variable called $best\_cost$ is initialized to ∞; this variable is dynamically updated to keep track of the least cost over all TSP tours examined so far at any stage of the algorithm. At every step, the algorithm evaluates the next eligible tree edge in the DFS order as explained below and also shown in Fig. 3.

At any given step, consider the newly included tree edge to be from node u to node v, and the cost of the corresponding graph edge (i, j) to be c_{ij}. Let $c^*(v)$ denote the cost of the least cost TSP tour passing through node v. There are two possibilities for v.

If v is a leaf, then $c^*(v)$ is set equal to the net cost of the path from the root node to v. Subsequently, if $c^*(v) < best\_cost$, then *best_cost* is updated to $c^*(v)$.

If v is an internal node in the search tree, a lower bound for $c^*(v)$ is computed using a matrix reduction operation. If the computed lower bound (*lbc(v)*) is observed to be greater than or equal to *best_cost*, further exploration of the subtree under v becomes unnecessary and so the subtree is pruned and the computation returns to the parent node u; otherwise, the DFS is continued under v's subtree.

Lower bound calculation. We use the method shown in [15] for lower bound computation at each tree edge. An $m \times m$ matrix called the *reduction matrix* (R) is maintained throughout execution. Initially, the matrix at the root node is set equal to the cost matrix defined by E. At any step of the DFS, *lbc(v)* is calculated as follows:

1. All entries in row i and column j of R are set to ∞;
2. $R[j, 1]$ is also set to ∞;
3. All rows and columns that contain at least one noninfinity value are *reduced* as follows:
   a. given row i, compute $min_i = min\{R[i, j]\}$ for all $1 \leq j \leq m$;
   b. then for all $1 \leq j \leq m, R[i, j] = R[i, j] - min_i$;
   c. similarly, given column j, compute $min_j = min\{R[i, j]\}$ for all $1 \leq i \leq m$;
   d. then for all $1 \leq i \leq m, R[i, j] = R[i, j] - min_j$. As this is done, all subtracted values (i.e., the minimum values) are accumulated into another variable *adjCost*.
4. Subsequently, the lower bound is given by $lbc(v) = lbc(u) + R[i, j] + adjCost$.

4 NoC Design

The problem of MP phylogenetic reconstruction using a branch-and-bound technique naturally lends itself to parallelization using a divide-and-conquer approach by subdividing the solution-space tree into independent subtrees. A PE computes one subtree at a time and considers pruning based on the best cost available from its peers. As this requires a good combination of parallelism and intercore communication, NoC provides an ideal platform owing to its inherent parallel architecture, customizability of its core, and its efficient communication infrastructure. We designed and implemented the PEs and the on-chip communication network for this NoC. Two types of communication infrastructure were explored. One is a regular mesh network. The other is a hierarchical four-way tree or quad-tree. The remainder of this section details the design of the PE, switches, communication fabric, and application mapping.

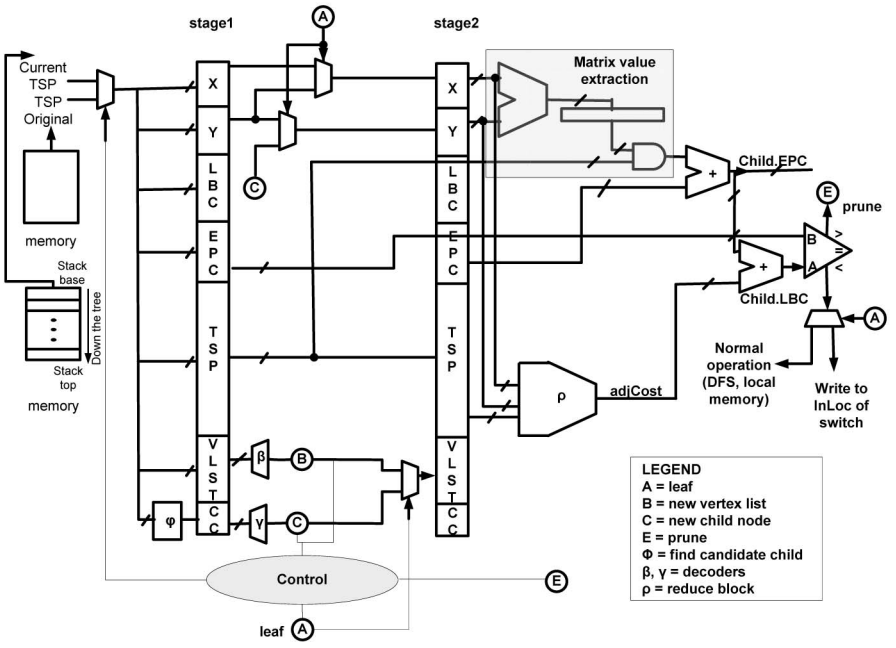

Fig. 4. Internal architecture of processing element for edge reduction.

4.1 PE Design

The PE has a pipelined architecture optimized to handle the computation along an edge as per the algorithm described in Section 3. Since the PE carries out the most computationally intensive part of the whole operation, our attempt has been to optimize its architecture to ensure that the number of clock cycles required scales nicely with increasing graph size (number of vertices, m). The primary performance parameter is timing, which aims at reducing application runtime and overall latency. To achieve this, we designed our PE for *O(m)* time complexity, as discussed further in Section 4.1.1. The problem of reducing overall latency of the system has been dealt within Section 4.5. Our PE has an integer data path because breakpoint median computation for MP phylogenetic reconstruction consists entirely of integer operations. The principal components of the PE are a *reduce* block and peripheral control logic, each of which is described in detail below. We use the short form $lg\ k$ to denote $log_2 k$. The data path consists of the following fields (m: number of vertices and w: maximum edge weight):

1. x—the parent node (u) uses $lg\ m$ bits.
2. y—the child node (v) uses $lg\ m$ bits.
3. *LBC*—the lower bound cost (*lbc(u)*) estimate at an edge; this requires $lg\ m + lg\ w + 1$ bits.
4. *EPC*—the exact path cost $(lbc(u) + R[i, j])$ determined so far; it takes $lg\ m + lg\ w + 1$ bits.
5. *TSP*—the TSP adjacency matrix (R), flattened. Its representation takes $m^{2*} lg\ w$ bits.
6. *VLST*—the current list of vertices traversed. $m*(lg\ m) + 1$ bits are required to store this field.
7. *CC*—the candidate children at every stage. It takes m bits.

As is evident, the data path complexity of the hardware is $O(m^2)$. In our approach, breakpoint distances can range from 0 to 3, which is the range of the valid weights we used. We used the weight 4 to denote a nonexistent edge or ∞. A different range of weights just changes the number of bits for w. A block diagram of the PE is shown in Fig. 4. Subsequent references to the subblocks in parentheses (e.g., ρ φ, etc.) in this section refer to this figure.

4.1.1 Reduction Block

This block (ρ) carries out the matrix reduction operation described in Section 3. Based on the algorithm, the runtime of the operation is a function of the matrix size, i.e., $O(m^2)$. This operation consumes the maximum fraction of the total time required for an edge computation. Hence, a significant amount of time is saved by suitably optimizing its design. Our implementation achieves *O(m)* cycle time by using microlevel parallelism inside the *reduce* block. This has the effect of drastically reducing the total time as well as providing better time scalability with increasing input graph size, m.

The matrix is reduced using the new values of x and y in *stage2* (see Section 4.1.2 below for details on the operations up to this stage) and the adjacency cost *adjCost* is obtained. Fig. 5 shows the architecture of the *reduce* block. The flattened TSP matrix is initially reorganized into rows and columns in the component denoted as *matrix*. There are m rows and m columns with each entry taking up $lg\ w$ bits. The register bank *minval* of width $m^*(lg\ w)$ is initialized with a bit pattern representing *infinity* (3'b100 as mentioned earlier). A *counter* is used as a state machine controller. There is an m-sized bank of comparators that compare one element from every row or column in every cycle. Minimum value calculation for all rows and the same for all columns take m cycles each. Additional three cycles are required for subtraction of the minimum values, for calculation of the final *adjCost*, and for control operations for each case (row and column blocks). The entire reduction operation takes $2^*(m + 3)$ cycles to complete under the current implementation.

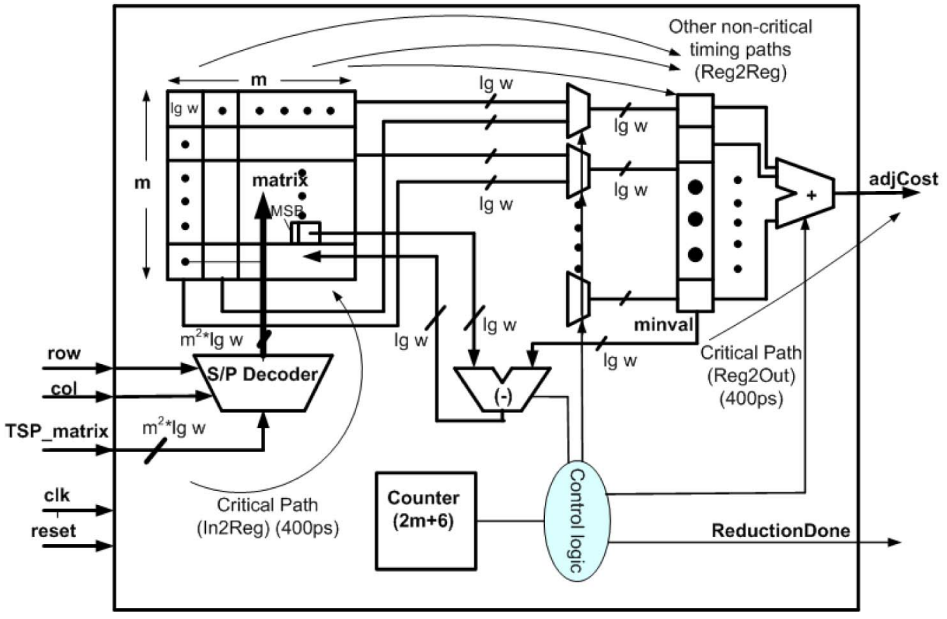

Fig. 5. Internal architecture of reduction block (ρ) for linear-time matrix reduction.

4.1.2 Peripheral Control Logic

The peripheral control logic is used for vertex selection, cost comparison, and data management. The register bank for the first stage is *stage1*, which has the same width as the data path. The input control multiplexer initially switches to select the current vertex data. The *CC* field is computed (φ) from *VLST* in m cycles in the worst case.

In the second stage, the candidate child is found by scanning (γ) *CC* of *stage1*. Again, this requires m clock cycles in the worst case. Using this candidate child, *VLST* is updated (B) for the child node in the graph. If it is not a leaf node (A), the candidate child becomes the next child node, while the current node (y of *stage1*) becomes the parent node x of *stage2*. During the same stage, the data pertaining to the best case obtained so far are fetched into *stage1*. The input multiplexer now selects the lowest cost data (*global best cost*) available to the PE at this time. At this stage, *TSP* of *stage1* gets the original TSP matrix.

The current value of the exact cost of the path found so far, *EPC*, is updated by adding to it the edge cost from x to y in the original adjacency matrix. This is checked against *global best cost* and *reduce* operation is started only if *EPC* is lower. The sum of *adjCost* (obtained from *reduce* operation) and *EPC* yields the lower bound cost, which is again compared with the best cost found so far. If *EPC* or *LBC* is larger than the current *best cost*, the tree is pruned (E), the current child is aborted, and the path through another child is explored. The data on *stage2* are reloaded back to *stage1* with the old value of x and a new calculation for the candidate child. If *LBC* is smaller and we have not reached a leaf node, normal operation (DFS) continues with the new set of data. If we have hit a leaf node with an *LBC* lower than the best cost globally found so far, this value (new *global best cost*) is sent to the switch to be communicated with other PEs in the network.

4.1.3 Memory

The memory is physically distributed across all PEs, and the memory local to each PE has two logical partitions. One part of the memory stores the TSP matrix corresponding to the root of the subtree that is currently assigned to that PE. Another part of the memory stores the intermediate matrix data that result along the way of evaluating a path down that subtree.

The part of the memory that stores intermediate matrix data can be implemented as a stack. During DFS, the new vertex data (path cost, vertex list, and associated adjacency matrix) are pushed into the stack (Fig. 4). The stack is full only when the leaf node is reached. If there is pruning (before the leaf node is reached), the stack is popped. In this scheme, every PE has a stack with m levels, where each level of the stack needs to store $(m+1)^*(lgm) + (m^2+1)^*(lg\ w) + 2$ bits. Since $lg\ w$ is a constant, the total memory requirement is $O(m^2)^*O(m)$ or $O(m^3)$. The total memory required per PE for different values of m is shown in Table 1.

An improved scheme is explored, where the memory requirement is reduced to $O(m^2)$. In this scheme, the entire adjacency matrix at each level is not stored in the stack. Instead, we store only the original values in the row i and column j that are made ∞, the row-wise minima and the column-wise minima obtained during reduction at each level. These data require $4m^*(lg\ w) + 2m$ bits at each level. In addition, the adjacency matrix only for the current child level is stored. While going back to the parent, these data at each level are used to backtrack and reconstitute the adjacency matrix at the parent level. The reconstitution step leads to a negligible runtime penalty of 1.8 percent. The total

TABLE 1
Per PE Memory Requirement for Different Input Genome Sizes

Number of genes per input genome, m	Per PE memory requirement (MB)	
	Basic scheme	Improved scheme
128	0.514	0.024
256	4.063	0.094
512	32.282	0.375
1024	257.252	1.5

TABLE 2
Worst Case Write Latency in Clock Cycles

N	Mesh	Quad-tree
4	6	6
8	9	10
16	12	10
64	14	12
256	30	14
1024	62	16

reduction operation in the improved scheme takes $0.1832\ \mu s$ against $0.18\ \mu s$ in the older scheme. However, the overall memory requirement improves to $(m+1)^*(lg\ m)+(5m^2+1)^*(lg\ w)+2m^2+2$ bits. This improves the memory scalability of the design and enables implementation for higher values of m for the same per-PE memory as can be seen from Table 1. We use this memory implementation for our experiments.

A list of all subtrees to be computed is maintained in memory. Once each PE completes one subtree reduction, it picks up the next available subtree and removes it from the list. This is achieved by maintaining a global array of flags and a mutually exclusive semaphore.

4.2 Network Design

The choice of the network architecture is affected by application modeling and traffic pattern analysis, as is explained in the seminal paper on an NoC design methodology [20]. Our application is mapped on a set of homogeneous cores, each of which carries out reduction of a subtree. The need for communication arises when a PE needs to update the network with the best score it has obtained. This is explained in detail in Sections 4.3 and 4.4. The mode of communication involved in this case is a *conditional broadcast*. We explored two different kinds of network architecture—a mesh, shown in Fig. 1a, and a quad-tree, shown in Fig. 1b. A mesh is the most appropriate scalable topology for broadcast traffic. Its regularity provides for easier timing closure and reduces dependence on interconnect scalability [20]. The hierarchical nature of a quad-tree minimizes the diameter of the network for the same number of nodes, thereby amortizing the router (switch) overhead and reducing latency [20]. Other common network architectures like point-to-point, full crossbar, or ring do not scale well with increasing system size [21], [22] and far exceed latency/area budgets. With increasing system size (N), the number of interswitch links in a mesh increases faster than that in a quad-tree. The expected volume of inter-PE communication in our application is relatively low. Hence, having fewer links in our network can lead to potential savings in area and power without incurring a risk of network congestion.

The diameter of a mesh architecture increases as $O(\sqrt{N})$ where N is the system size or the number of PEs. The same for a quad-tree increases as $O(log_4 N)$. In the worst case, the mode of communication for our application involves some form of broadcast as the *best cost* is written to all the PEs except for the originating PE. Hence, the worst case hop count is a linear function of the diameter. It should be remembered that all links are not of the same length in a quad-tree, where links higher up the tree are longer and have a greater delay. Table 2 shows an estimate of the number of clock cycles required per write in the worst case in 65 nm CMOS technology with a clock period of 400 ps. Quad-tree has an advantage over mesh in terms of communication latency for $N > 16$. However, the key advantage of a quad-tree comes from power savings because the number of links and switches is drastically reduced. These comparisons are provided in Section 5.

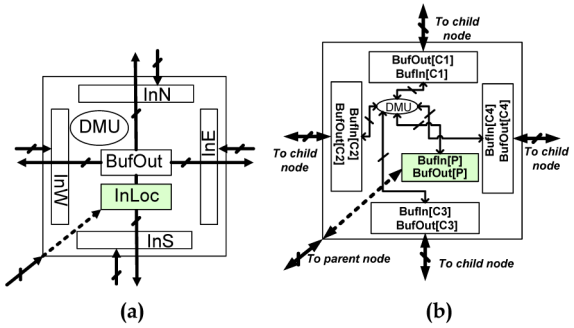

Fig. 6. Internal architecture of switch for (a) mesh and (b) quad-tree.

The problem of mapping the application on the nodes of the NoC is also important in optimizing overall latency. This is discussed in detail in Section 4.5.

4.3 Switch Design

Different switches are designed for each of the two network architectures explored.

4.3.1 Mesh

A typical switch that is used on a mesh is shown in Fig. 6a. Input buffers *InN*, *InE*, *InS*, and *InW* receive data from four neighboring switches and an input buffer *InLoc* receives data from the associated PE. There is a dedicated buffer (*BufOut*) that provides data to the network as well as to the associated PE.

Each set of input/output data consists of the fields 1) path cost, 2) vertex list, and 3) transmission control bits. At every cycle, one of four transmission decisions is taken by the Decision Making Unit (DMU) and the data are written into an internal buffer (*local*). The same is transmitted out in the next cycle through *BufOut*. The transmission control bits are as follows:

- *NOTX*: No valid transmission.
- *NORETX*: No retransmission.
- *DOTX*: New best cost from local PE; transmit.
- *TRWL*: New best cost from other PE; transmit and update local PE.

Fig. 7 shows the state diagram for the control states and Fig. 8 shows a timing diagram for a typical situation. It is to be noted that a switch receives data from each of its neighboring switches in every cycle but the transmission control bits determine whether the data are valid for consideration or not. The data are considered if the control bits are *DOTX* or *TRWL* but not if they are *NOTX* or *NORETX*.

4.3.2 Quad-Tree

There are different levels of switches for this network architecture. The leaf level switches (refer to Fig. 1b) are denoted L1, the next higher level L2, and so on. An L1 switch consists of five buffered input/output ports (*BufIn/BufOut*): four catering to the four leaf PEs and the fifth to the parent

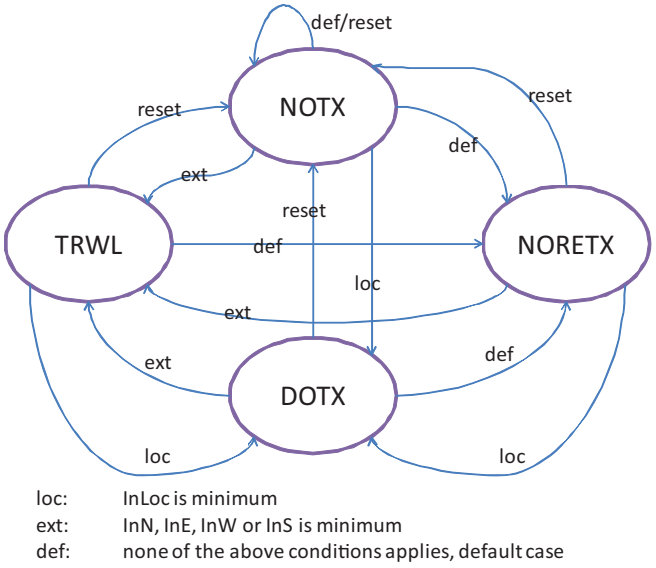

Fig. 7. State diagram of control states in a mesh switch.

switch. For an L2 switch and upward, four children ports cater to lower level switches and the parent port caters to the higher level switch. The top level switch has only four downlinks but no uplink. Each set of input/output data consists of the fields 1) Path Cost, 2) Vertex List, and 3) Update control bit (*UCB*). The switch architecture is shown in Fig. 6b.

UCB is a flag to indicate whether the status of the data is valid (*UPDT*) or invalid (*NOUP*). The receiving parent or child switch infers "no transmission" if *UCB* is set to *NOUP*. In every cycle, the switch takes a decision based on the following algorithm.

Let $C1, C2, C3$, and $C4$ be the four (children) downlinks and P be the (parent) uplink and let us define the set $L = \{C1, C2, C3, C4, P\}$. Let us suppose the best (lowest) cost, PC_i, for a decision cycle comes from $i \in L$, i.e., $PC_i < PC_j \forall j \neq i, j \in L$. Then, we have

$$BufOut[k] \leftarrow PC_i \forall k \in L,$$
$$UCB[i] \leftarrow NOUP,$$
$$UCB[j] \leftarrow UPDT \forall j \neq i, j \in L.$$

4.4 Communication Protocol

In the mesh architecture, every switch communicates with its immediate neighbor and gets data in every cycle from at most four neighboring switches. Based on the decision mechanism described in Section 4.3, the switch places data on *BufOut* with appropriate control bits. The neighboring switches get this value in their input buffers in the next cycle. Hence, at every cycle, data are sent in all four directions.

In the quad-tree, every switch communicates with its four children and one parent in every clock cycle. It receives data from its parent and/or one or more of its children and takes a decision on the lowest cost available to it thus far. Once found, these data are placed on four output buffers, except the direction they came from along with appropriate *UCB*. For the best cost data to propagate to the entire network, they have to go through a maximum of H hops where H is given by

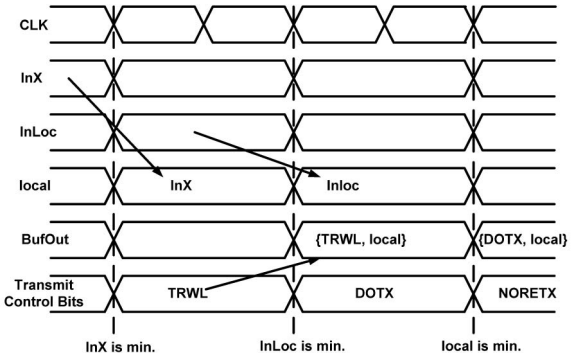

Fig. 8. Timing diagram showing typical scenarios encountered in a mesh switch.

$$H = 2^{*}\lceil \log_4 N \rceil. \quad (1)$$

Note that $H/2$ is the *height* of the tree. One important fact to keep in mind is that each hop does not consume the same number of clock cycles as the wire length varies at different levels.

The need for inter-PE communication arises when a particular PE checks against the *global best cost* obtained so far and finds out that its local best cost is lower than the global best cost. At this stage, the PE should broadcast its newly obtained value to the whole network. One way to implement this is to use flooding. However, this could lead to an unnecessary network congestion thereby affecting scalability. Therefore, we devised an improved alternative strategy where a PE *conditionally broadcasts* valid data only if

1. its local best cost is worse than the global best cost but it has *not yet participated* in the broadcast of this global cost or
2. its *local best cost* is better than the *global best cost* (currently available to the rest of the network) *and* it has *not been previously transmitted*.

The above scheme ensures elimination of redundant communication, thus reducing communication overhead and power consumption without compromising on the correctness of the answer.

4.5 Application Mapping and Trade-Off

Application mapping significantly impacts the overall latency and total energy consumption of the NoC. The PE that finishes its share of reduction computations last limits the performance of the entire system. The determining factor for this is the load distribution among PEs, which is dependent on input data. In our scheme, each PE picks a subtree dynamically from a common pool of available uncomputed subtrees, once it has finished computing its own subtree. This can happen either when the PE has finished computing the subtree exhaustively or when it has pruned it. This could result in each subtree contributing to a different number of reductions and each PE computing a different number of subtrees. It is evident that we need to ensure that PEs are evenly utilized to minimize the impact of a "bottleneck" PE and achieve the best overall latency. Hence, application mapping on the NoC needs to be optimized such that the load distribution among PEs is even. We use the following definitions to formulate the problem:

- f_i: utilization factor of PE i,
- t_i: application latency of PE i,

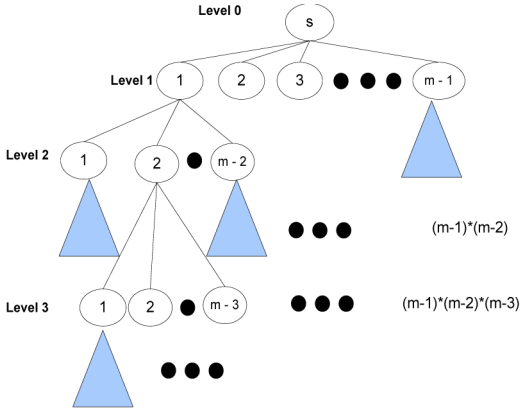

Fig. 9. Diagram showing the number of subtrees generated at different levels after partitioning the search space tree.

- T_{PCIe}: latency overhead of the system for loading data through PCIe, and
- T_{pick}: cumulative latency overhead of each PE picking subtrees from the pool of uncomputed subtrees.

Overall application latency, $T_{overall}$, is given by

$$T_{overall} = T_{PCIe} + T_{pick} + \max\{t_i\}, \quad (2)$$

where $max\{t_i\}$ is over all i. Note that the last term is the latency of the "bottleneck" PE. t_i is proportional to f_i. T_{PCIe} and T_{pick} are related to f_i as explained below.

Experiments showed that the load distribution among PEs (f_i) becomes more balanced when the number of subtrees in the common pool is much higher than the number of PEs. For a graph with m vertices, the solution-space tree with the starting node as root (level 0) has $(m-1)$ nodes at level 1, $(m-1)^*(m-2)$ nodes at level 2, $(m-1)^*(m-2)^*(m-3)$ nodes at level 3, and so on (Fig. 9). So partitioning the solution space by choosing subtrees rooted at a deeper level generates more subtrees, helping to balance load and thereby ensure maximum achievable parallel speedup. Now, T_{PCIe}, the overhead involved in loading the entire set of subtrees to the system using PCIe increases with the amount of data that needs to be transferred, which increases with the number of subtrees. T_{pick} also increases with the total number of subtrees handled by each PE. Hence, it is clear that the dependence of t_i on f_i is opposite to that of T_{PCIe} and T_{pick} on f_i. We need to optimize $T_{overall}$ in (2) with these constraints. As explained in Section 5, we have considered $m = 110$ in our experiments. In this case, we resolved this trade-off by choosing to work on subtrees rooted at level 2, which generated 109^*108 subtrees and yet kept the overhead to a manageable amount. Note that 109^*108 is much larger than the largest system size (number of PEs, $N = 64$) we experimented with, which led to a balanced load distribution.

5 Experimental Results

5.1 Experimental Setup

The performance evaluation of the NoC was carried out from the timing and power perspectives during phylogenetic reconstruction with varying data sets. Different parameters associated with the NoC are as follows: the system size, N, is the number of PEs in the NoC. N was set to 4, 16, and 64 for evaluating the performance of the NoC with scaling of system size. The number of vertices in the input graph is denoted by m, which determines the width of the data path. In practice, this value should be set to the number of genes shared by the input genomes. For example, chloroplast genomes of potato, tomato, and wheat share 110 genes; hence, $m = 110$ in this case. In our experiments, we used two types of input data: 1) multiple sets of synthetic genomes with $m = 110$ used for exhaustive system-wide parametric study; and 2) two sets of real input genomes (as explained in Section 5.3). Note that the value of m affects the size of the data path and the memory requirements in the PE as per the discussion in Section 4. Since we have dealt with three-median breakpoints, breakpoint distance can vary between zero and three. Without loss of generality, the maximum weight w has been taken to be four to indicate ∞ or a nonexistent edge. As with m, this choice affects the data path size but to a lesser degree.

Each PE with its corresponding switch constitutes one node in the NoC. They were implemented by synthesizing Verilog RTL using Synopsys Design Compiler followed by place-and-route with Cadence SoC Encounter using standard cell library of 65 nm process [23]. Extracted parasitics were used in Synopsys PrimeTime to determine postlayout timing performance. The pipelined design could sustain a clock frequency of 2.5 GHz in each PE and switch. This was verified with $m = 110$ and higher. The critical path delay using 65 nm timing library is within 400 ps, as shown in Fig. 5. In order to estimate the total power dissipation, it becomes necessary to record the total communication events involving all the PEs. For modeling the event statistics, we implemented a multithreaded program to act as the *software driver*, which recorded the number of reduction operations performed by each thread, and the number of successful write operations by that thread. Each individual thread of the software driver functionally simulated a processing element of the NoC. Thereafter, these statistics were used in conjunction with Synopsys Power Compiler using the library [23] for estimating the total computation power of all the PEs. The switch power (also obtained from Synopsys Power Compiler) was separately added to this component. Logic gate count for one PE and associated network switch with $m = 128$ is 1.267 million.

Interconnect characteristics (delay, power) were determined using Cadence Spectre. Wire capacitance information extracted from layout was used to determine delay and energy dissipation of interconnects. Multiple clock cycle delay in longer interconnects was accounted for.

PCI Express 2.0 is used as the interface for initially loading the graph data into the NoC. For modeling this interface, Synopsys Designware IP PCI Express 2.0 PHY was used. It has been implemented on 65 nm process and operates at 5.0 Gbps. We use a 32-lane PCIe 2.0 for our simulation. Both mesh and quad-tree architectures were considered for performance evaluation.

GRAPPA [6] was used as the software benchmark. It is a standard and widely used program for MP phylogenetic analysis. To achieve its best performance, GRAPPA was run in its *multithreaded mode* on a quad-core 2.40 GHz Intel Xeon E5530 processor with 16 GB of RAM. The runtime measured

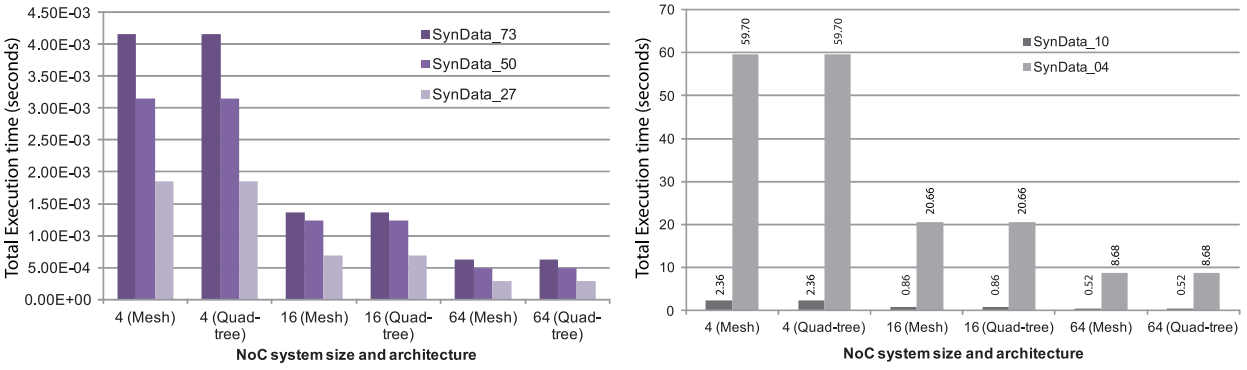

Fig. 10. Total execution time in hardware for (a) $SynData\_73$, $SynData\_50$, and $SynData\_27$ and (b) $SynData\_10$ and $SynData\_04$.

through GRAPPA served as the basis in our speedup calculations. Specifically, speedups reported are calculated as the ratio of GRAPPA runtime over the total execution time on an N-PE NoC. Note that different multithreaded GRAPPA runs were found to yield different output sequences with the same optimum score. Our NoC simulation also outputs a sequence that matches this optimum score.

5.2 Performance on Synthetic Data

Five synthetic data sets were generated and used as input. Each input consisted of three genomes with 110 genes each such that $m = 110$. Each data set was generated to have a different common subsequence length and, hence, different divergence. Pairwise divergence (δ) is given by subtracting the length of the longest common subsequence from m. We have three values of δ for each input. The standard deviation of the pairwise divergences (σ_δ) was normalized by dividing it by the mean (μ_δ) and used as the divergence metric, $\Delta = (\sigma_\delta/\mu_\delta)$. This metric serves as a measure of the skew among the three genomes and is made to vary across the entire range of possible values, thereby covering the entire range of the possible input spectrum. Low values of Δ indicate that the genomes are equally far apart irrespective of the actual magnitude of the breakpoint distance. A high value of Δ indicates that two genomes are closer to each other than they are to the third. Five synthetic sets of three genomes each were generated such that the values of Δ in these inputs are 0.731, 0.498, 0.274, 0.103, and 0.039, respectively; these inputs were labeled $SynData\_73$, $SynData\_50$, $SynData\_27$, $SynData\_10$, and $SynData\_04$, respectively. It is also to be noted that the δ values and μ_δ increase as we move from $SynData\_73$ to $SynData\_04$.

5.2.1 Timing Performance

Figs. 10a and 10b show the total execution times for NoCs with system sizes (N) 4, 16, and 64 for all the synthetic inputs. The total execution time includes the total computation and communication cycles spent in the NoC and the time required to load the data on the NoC using PCIe. It is interesting to note that the absolute runtimes are heavily dependent on the input data and the absolute divergences. Since the execution times are a function of the bottleneck number of reductions carried out by the PEs (see Section 4.5), the execution times for $SynData\_10$ and $SynData\_04$ are orders of magnitude higher than those for the other three inputs. This is because of their larger absolute divergences and, hence, a larger number of reductions performed by each PE. There is not much difference in the overall runtimes on mesh and quad-tree. This is because quad-tree helps reduce only the write latency, which contributes a small fraction to the total execution time in this case. Fig. 11 shows the communication-only (write) latency for mesh and quad-tree NoCs with different system sizes. The advantage offered by the quad-tree topology in reducing communication-only latency is evident from this plot. This is also similar to what is expected for $N = 4, 16$, and 64 from Table 2.

Fig. 12 shows the speedup over GRAPPA using a quad-tree for these inputs. Since speedup is the ratio of GRAPPA's runtime to the execution time on our design, the trends in speedup and execution time are not identical across different inputs. For example, even though execution time increases from $SynData\_10$ to $SynData\_04$ for all system sizes, speedup is also observed to increase because GRAPPA's runtime increases by a larger factor. Speedup is also dependent on Δ, which indicates that our design is able to accelerate median computation of genomes that are almost equally far apart (e.g., $SynData\_04$) significantly more compared to the case where two of the genomes are very close to each other (e.g., $SynData\_73$). This observation

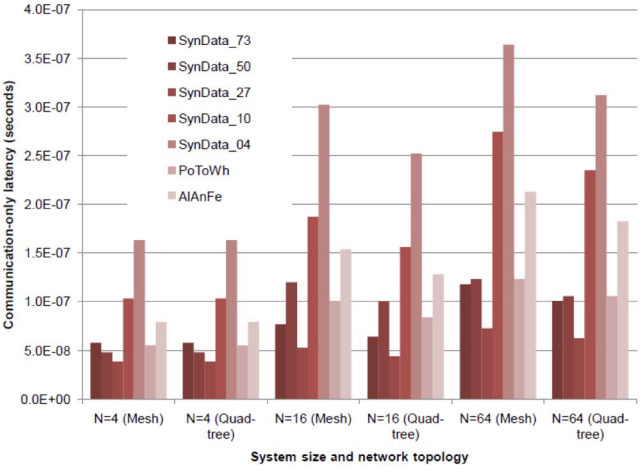

Fig. 11. Variation on write latency with network topology and system size.

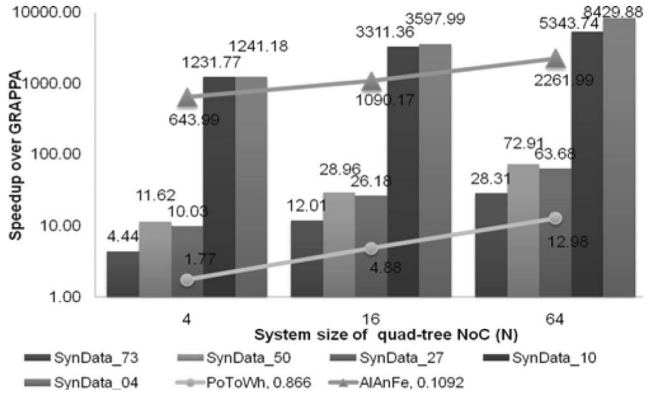

Fig. 12. Absolute speedup over GRAPPA.

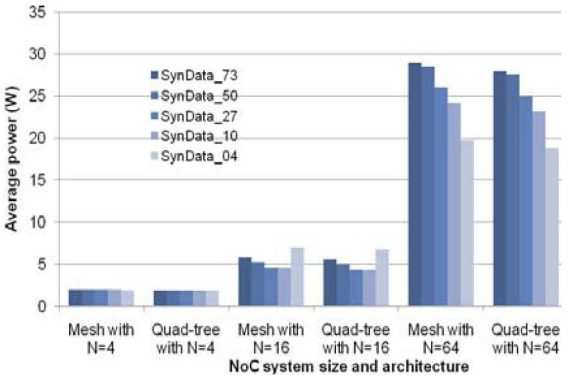

Fig. 14. Power consumption across various inputs, network architectures, and system sizes.

is more clearly demonstrated in Fig. 13, where the speedup on a quad-tree NoC with $N = 16$ is plotted against values of Δ. The best speedups of 1,241 ($N = 4$), 3,598 ($N = 16$), and 8,430 ($N = 64$) are consistently obtained with $SynData\_04$. Our results compare favorably with the overall speedup of 417 or the application speedup of 1,005 achieved by FPGA-based acceleration of GRAPPA in [7].

Note that the synthetic data encompass almost the full range of possible inputs, with Δ varying from 0.039 to 0.731. Biological inputs can lie on either end of the spectrum or anywhere in between. In particular, as we mention again in Section 5.3, the two real genomic inputs that we use have Δ values of 0.866 and 0.1092. It is also interesting to note that we achieve significantly higher speedups in the cases of genomes displaying greater absolute divergence ($SynData\_10$ and $SynData\_04$). These are also the cases where even highly optimized software implementations such as GRAPPA take very long times to complete. Our design provides better speedup when there is a greater requirement and, hence, will be of more practical value.

5.2.2 Energy Performance

Several measures were used to evaluate the energy performance of the NoC. The average power consumption for mesh and quad-tree NoCs for $N = 4$, 16, and 64 is shown in Fig. 14. It will again be noticed that power consumption is a function of the input data, especially for $N = 64$. There is a slight advantage of quad-tree over mesh in terms of power efficiency. For example, a quad-tree NoC consumes up to five percent less power than that based on a mesh NoC. Note that the PEs in both configurations have the same power consumption and the savings come entirely from the communication architecture. Higher levels of network activity would lead to greater power savings in the quad-tree. However, since the execution time varies widely across inputs, only power consumption provides a partial picture. A more accurate rubric is the total energy consumption, shown in Figs. 15a and 15b. Although these figures show the advantage of quad-tree over mesh in terms of energy performance, comparing only the communication energy consumptions in Figs. 16a and 16b further highlights this. Quad-tree consistently outperforms mesh by consuming around 75 percent less communication energy. Both average power and total energy are input dependent and generally show a marked increase with increase in system size (N). The most interesting observation on energy efficiency, however, can be seen from Figs. 17a and 17b that show the variation of the energy-delay product (EDP)

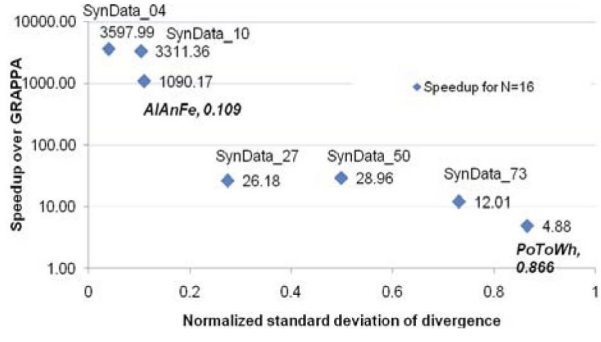

Fig. 13. Variation of speedup with skew of input data on quad-tree NoC with $N = 16$.

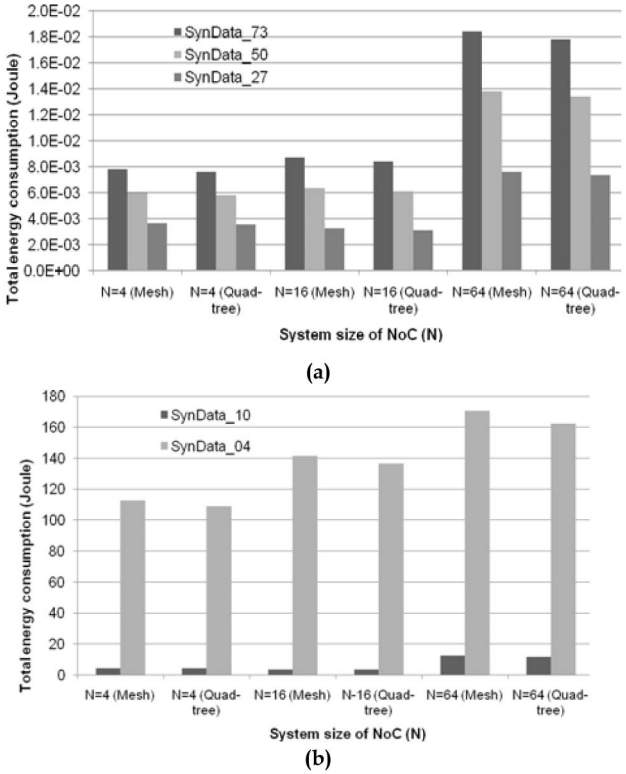

Fig. 15. Energy consumption across different inputs.

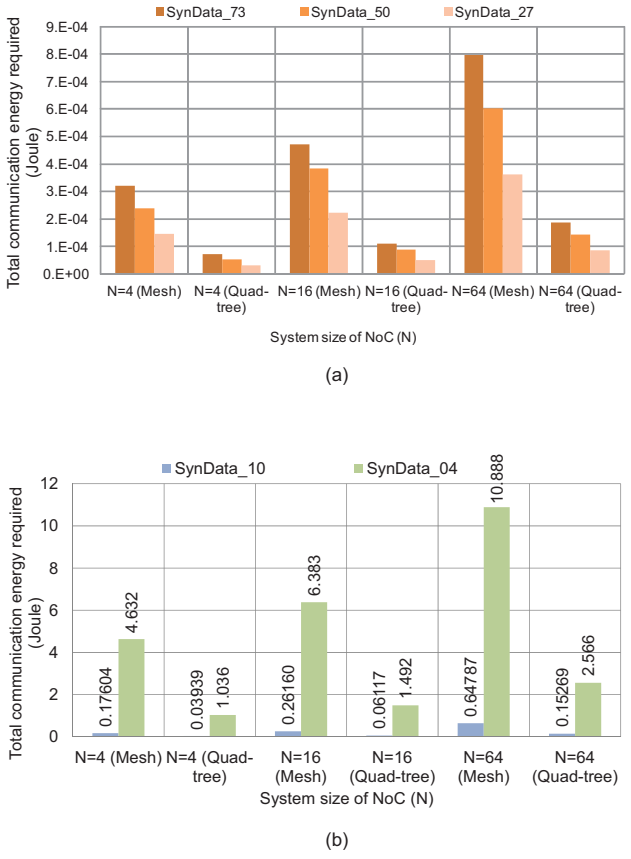

Fig. 16. Communication energy expended across different inputs.

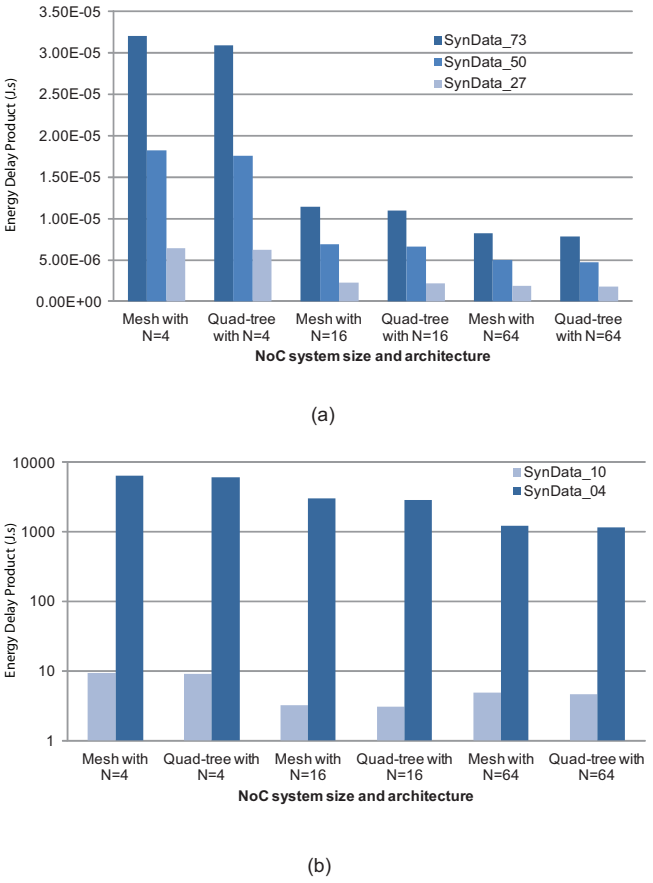

Fig. 17. Variation of energy-delay product across inputs.

with system size (N) across all inputs. EDP is observed to *decrease* with increasing system size for most inputs. This is because the increase in energy consumption is compensated by the runtime reduction, thereby showing that parallelization is indeed energy efficient in this case.

5.3 Performance on Real Genomic Data

Two real genomic inputs were used to evaluate the performance on biological data. Genomic data were downloaded from the National Center for Biotechnology Information's organellar genome repository [24]. One input (*PoToWh*) consisted of the chloroplast genomes of *Solanum tuberosum* (potato, 141 genes), *Solanum lycopersicum* (tomato, 130 genes), and *Triticum aestivum* (bread wheat, 137 genes). The other input (*AlAnFe*) consisted of chloroplast genomes of *Chlamydomonas reinhardtii* (a unicellular green alga, 109 genes), *Brachypodium distachyon* (purple false brome grass, an angiosperm, 133 genes), and *Adiantum capillus-veneris* (black maidenhair fern, 130 genes). These genomes were preprocessed with Mauve [25] in order to determine the common genes. The values of Δ for the inputs are 0.866 for *PoToWh* and 0.1092 for *AlAnFe*. This is indicative of the fact that *PoToWh* represents a skewed data set, with potato and tomato being much closer to one another than they are to wheat. This is expected, as evolutionarily potato and tomato are closely related and belong to the same genus. On the other hand, *AlAnFe* represents a uniformly divergent scenario. The speedups obtained with these inputs for $N = 4$, 16, and 64 are shown in Fig. 12. Fig. 13 shows the speedup correlation with synthetic data having similar values of Δ.

As mentioned in Section 5.1, speedup is calculated as multithreaded GRAPPA runtime divided by the total execution time on the NoC. As explained earlier, the total execution time on NoC is proportional to the bottleneck number of reductions. For example with $N = 16$, the bottleneck number of reductions for *PoToWh* is 6,286 and that for *AlAnFe* is 46,958. The total execution times on a quad-tree NoC are 1.14 and 8.46 ms, respectively, and have the same ratio. In comparison, the GRAPPA runtimes are 5.55 ms and 9.22 s, respectively. Next, we turn our attention to the variation of speedup with increasing N. It can be seen from Fig. 12 that the speedup on *PoToWh* increases from 1.77 to 12.98 as we increase N from 4 to 64. For *AlAnFe*, the speedup increases from 643.99 to 2,261.99. Table 3 shows the mean, standard deviation, and the maximum (bottleneck) number of reductions per PE for *PoToWh* and *AlAnFe*. It is evident that speedup is inversely proportional to the maximum number of reductions per PE. Speedup also varies inversely as the average number of reductions when load is balanced among PEs. Finally, in order to investigate the reason behind the widely different speedups obtained with *PoToWh* and *AlAnFe*, we plot histograms (Figs. 18a and 18b) of the number of reductions per subtree for each of the inputs. The larger skew (Δ) for *PoToWh* is evident from a comparison of the two histograms. Due to the higher skew in *PoToWh*, the best cost is obtained quickly and most subtrees are pruned at the initial stage of the operation, leading to few (<10) reductions per subtree. The lower skew in *AlAnFe* leads to a more gradual update of the best cost and subtrees are pruned to a lesser degree. Since the reduction load is shared by several subtrees in the latter case, parallelization provides greater speedup.

TABLE 3
Reduction Statistics for *PoToWh* and *AlAnFe*

	N = 4			N = 64		
	Average reductions per PE	Standard deviation of reductions per PE	Max reductions per PE	Average reductions per PE	Standard deviation of reductions per PE	Max reductions per PE
PoToWh	15672.75	1847.57	17430	1942.73	194.73	2342
AlAnFe	69222	8558.69	79516	19496.84	1700.02	22614

6 Conclusion

In this paper, we have undertaken the design, implementation, and performance evaluation of an NoC-based multicore architecture for accelerating the breakpoint median problem in phylogeny. Our evaluation encompassed a wide spectrum of inputs, including both synthetic and real genomes. We show that the proposed NoC architecture provides a speedup of up to 8,430 with respect to multithreaded GRAPPA software. We also show how the relationship among the input genomes affects the timing performance of our design and we are able to provide greater speedup when software methods incur a huge runtime penalty. On the network architecture front, we demonstrate the superiority of a quad-tree over a mesh in terms of energy efficiency for this application class.

We believe that our current implementation provides appreciable performance enhancement over comparable hardware accelerators targeting breakpoint phylogeny, and can serve as a basis for more NoC-based platforms with applications to life sciences. In addition, our design provides a paradigm for accelerating similar vector or matrix-based applications like image processing.

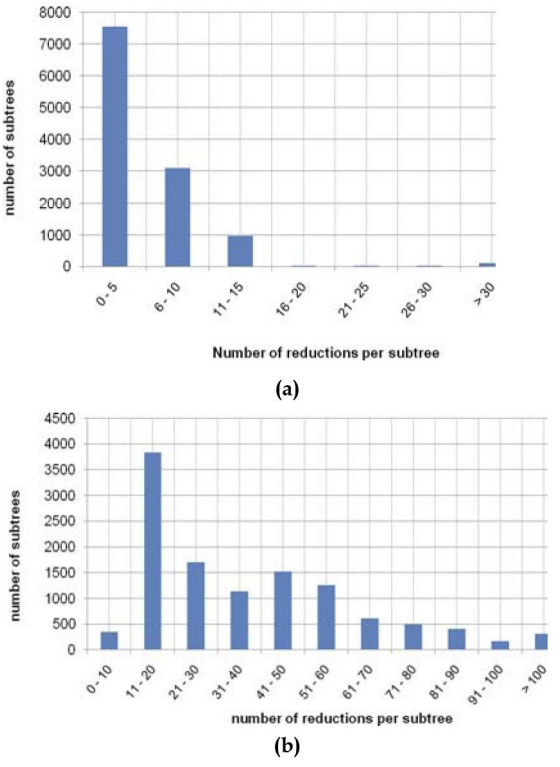

Fig. 18. Histogram of number of number of reductions per subtree for (a) *PoToWh* and (b) *AlAnFe*.

Acknowledgments

We thank biologists Professor Eric Roalson and Amit Dhingra for suggesting the real inputs that were used to test our NoC design. Special thanks to Professor Roalson for discussions and comments that helped us better our understanding of the problem from a scientific perspective. This work was supported by US National Science Foundation (NSF) grant IIS-0916463.

References

[1] D.A. Bader and M. Yan, "High-Performance Phylogeny Reconstruction," *Handbook of Computational Molecular Biology*, S. Aluru, ed., Chapman & Hall/CRC, 2005.
[2] P.H. Harvey and M.D. Pagel, *The Comparative Method in Evolutionary Biology*. Oxford Univ. Press, 1991.
[3] M. Blanchette, G. Bourque, and D. Sankoff, "Breakpoint Phylogenies," *Proc. Genome Informatics Workshop*, pp. 25-34, 1997.
[4] E.L. Lawler, J. Lenstra, A.R. Kan, and D. Shmoys, *The Traveling Salesman Problem*. John Wiley, 1985.
[5] I. Pe'er and R. Shamir, "The Median Problems for Breakpoints Are NP-Complete," *Proc. Electronic Colloquium on Computational Complexity*, vol. 5, no. 71, 1998.
[6] J. Tang et al., "Phylogenetic Reconstruction from Arbitrary Gene-Order Data," *Proc. Fourth IEEE Symp. Bioinformatics and Bioeng.*, pp. 592-599, 2004.
[7] J. Bakos and P. Elenis, "A Special-Purpose Architecture for Solving the Breakpoint Median Problem," *IEEE Trans. Very Large Scale Integration Systems*, vol. 16, no. 12, pp. 1666-1676, Dec. 2008.
[8] J. Bentley, "Fast Algorithms for Geometric Traveling Salesman Problems," *ORSA J. Computing*, vol. 4, pp. 387-411, 1992.
[9] B. Golden, L. Bodin, T. Doyle, and W. Stewart, "Approximate Traveling Salesman Algorithms," *Operations Research*, vol. 28, pp. 694-711, 1980.
[10] G. Reinelt, *The Traveling Salesman Problem: Computational Solutions for TSP Applications*, pp. 172-186. Springer-Verlag, 1994.
[11] S. Lin and B. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman Problem," *Operations Research*, vol. 21, pp. 498-516, 1973.
[12] P. Jog, J.Y. Suh, and D. Van Gucht, "Parallel Genetic Algorithms Applied to the Traveling Salesman Problem," *SIAM J. Optimization*, vol. 1, no. 4, pp. 515-529, 1991.
[13] D.L. Miller and J.F. Pekny, "Results from a Parallel Branch and Bound Algorithm for the Asymmetric Traveling Salesman Problem," *Operations Research Letters*, vol. 8, no. 3, pp. 129-135, 1989.
[14] M. Bellmore and G. Nemhauser, "The Traveling Salesman Problem: A Survey," *Operations Research*, vol. 16, pp. 538-558, 1968.

[15] E. Horowitz and S. Sahni, "Branch-and-Bound," *Fundamentals of Computer Algorithms,* pp. 370-421, Computer Science Press, 1984.
[16] T.S.T. Mak and K.P. Lam, "High Speed GAML-Based Phylogenetic Tree Reconstruction Using HW/SW Codesign," *Proc. IEEE Bioinformatics Conf.,* pp. 470-473, 2003.
[17] N. Alachiotis et al., "Exploring FPGAs for Accelerating the Phylogenetic Likelihood Function," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing,* pp 1-8, 2009.
[18] S. Zierke and J.D. Bakos, "FPGA Acceleration of the Phylogenetic Likelihood Function for Bayesian MCMC Inference Methods," *BMC Bioinformatics,* vol. 11, no. 1, pp. 1-12, 2010.
[19] F. Patas et al., "Fine-Grain Parallelism Using Multi-Core, Cell/BE, and GPU Systems: Accelerating the Phylogenetic Likelihood Function," *Proc. Int'l Conf. Parallel Processing,* pp. 9-17, 2009.
[20] R. Marculescu et al., "Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 28, no. 1, pp. 3-21, Jan. 2009.
[21] K. Lee, S.J. Lee, and H.J. Yoo, "Low-Power Network-on-Chip for High-Performance SoC Design," *IEEE Trans. Very Large Scale Integration Systems,* vol. 14, no. 2, pp. 148-160, Feb. 2006.
[22] L. Bononi and N. Concer, "Simulation and Analysis of Network on Chip Architectures: Ring, Spidergon and 2D Mesh," *Proc. Design, Automation and Test in Europe,* pp. 154-159, 2006.
[23] Circuits Multi-Projects (http://cmp.imag.fr/). Last Date Accessed: Mar. 2011.
[24] NCBI Database for Eukaryotic Organelles (http://www.ncbi.nlm.nih.gov/genomes/genlist.cgi?taxid=2759&type=4&name=Eukaryotae%20Organelles). Last Date Accessed: Mar. 2011.
[25] Genome Evolution Laboratory - Mauve Genome Alignment Software (http://asap.ahabs.wisc.edu/mauve/). Last Date Accessed: Mar. 2011.

Turbo Majumder received the BTech (Hons.) degree in electronics and electrical communication engineering and MTech degree in automation and computer vision, both from Indian Institute of Technology, Kharagpur, India, in 2005. He is currently working toward the PhD degree at the School of Electrical Engineering and Computer Science at Washington State University, Pullman. His research interests include network-on-chip and multicore system-on-chip design for biocomputing applications, VLSI design, and parallel and high-performance computer architectures. Prior to joining the PhD program, he worked for Freescale Semiconductor and Nvidia Graphics. He is a student member of the IEEE.

Souradip Sarkar received the MTech degree in computer science from Indian Statistical Institute, Kolkata, India, in 2006. He received the MS and PhD degrees from Washington State University, Pullman, in 2007 and 2010, respectively. He is a researcher at the Intel ExaScience labs, Leuven, Belgium, and also at Ghent University at the Elektronica en Informatiesystemen (ELIS) Department, Gent, Belgium. His primary interests include System-on-Chip and Network-on-Chip-based application specific hardware platforms, parallel and high performance system architectures, and synchronization issues in multiple clock domain-based systems. He is a member of IEEE.

Partha Pratim Pande received the MS degree in computer science from the National University of Singapore and the PhD degree in electrical and computer engineering from the University of British Columbia. He is an associate professor at the School of Electrical Engineering and Computer Science, Washington State University. His current research interests are 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. He currently serves in the editorial board of IEEE Design and Test of Computers and Sustainable Computing: Informatics and Systems (SUSCOM). He is the guest editor of a special issue on Sustainable and green computing systems for ACM *Journal on Emerging Technologies in Computing Systems*. He also serves in the program committee of many reputed international conferences. He is a senior member of the IEEE.

Ananth Kalyanaraman received the bachelor's degree in computer science and then joined the Computational Biology and Scientific Computing Laboratory at Iowa State University, where he received the MS and PhD degrees in 2002 and 2006, respectively. He is an assistant professor at the School of Electrical Engineering and Computer Science in Washington State University (WSU). He is also an affiliate faculty in the WSU Molecular Plant Sciences graduate program and in the Center for Integrated Biotechnology at WSU. His research interests include computational biology and bioinformatics, parallel algorithms and applications, and combinatorial pattern matching. The primary focus of his work has been on developing high-performance computing solutions for data-intensive problems originating from the areas of computational genomics and metagenomics. He continues to serve on the program committee of several conferences in the areas of parallel processing and computational biology, and also has chaired workshops and special sessions in this area. He is a member of the ACM, the IEEE, ISCB, and LSS.