A Special-Purpose Architecture for Solving the Breakpoint Median Problem

Jason D. Bakos, *Member, IEEE*, and Panormitis E. Elenis, *Student Member, IEEE*

***Abstract*—In this paper, we describe the design for a co-processor for whole-genome phylogenetic reconstruction. Our current design performs a parallelized breakpoint median computation, which is an expensive component of the overall application. When implemented on a field-programmable gate array (FPGA), our hardware breakpoint median achieves a maximum speedup of 1005× over software. When the coprocessor is used to accelerate the entire reconstruction procedure, we achieve a maximum application speedup of 417×. The results in this paper suggest that FPGA-based acceleration is a promising approach for computationally expensive phylogenetic problems, in spite of the fact that the involved algorithms are based on complex, control-dependent combinatorial optimization.**

***Index Terms*—Bioinformatics, computational biology, field-programmable gate array (FPGA), phylogenetic reconstruction, reconfigurable computing.**

I. Introduction

THE USE OF coprocessors has become an increasingly popular technique for accelerating computationally expensive applications. Possibly the best-known use of coprocessors are graphics processor units (GPUs), which accelerate 3-D rendering [1] and high-definition video playback [2]. While GPUs now form a substantial market in consumer computing, we believe that coprocessor acceleration also has the potential to achieve a significant impact for scientific computing.

In high-performance reconfigurable computing (HPRC), field-programmable gate arrays (FPGAs) act as coprocessor(s). In this technique, a special-purpose hardware version of the application's bottleneck computation (sometimes referred to as a "kernel computation") is implemented in FPGA logic. The performance improvement from this approach can be significant. For example, assume an application requires one week of CPU time. Also, assume there is a bottleneck computation that consumes 99% of the application's total execution time. Finally, assume a hardware implementation of the bottleneck computation achieves an average speedup of 200 over software (including the CPU-FPGA communication overhead). According to Amdahl's Law, using the hardware implementation as an accelerator yields an end-to-end application speedup of 67, and the accelerated application now requires only 2.5 h of execution time.

Manuscript received July 23, 2007; revised September 24, 2007. Current version published November 19, 2008.

The authors are with the Department. of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208 USA (e-mail: jbakos@cse.sc.edu; elenis@cse.sc.edu).

Digital Object Identifier 10.1109/TVLSI.2008.2001298

Applications that are best-suited to coprocessor acceleration are those that are dominated by a particular bottleneck computation that is amenable to significant speedup by hardware implementation. For this, it must contain more inherent parallelism than can be exploited by a general-purpose microprocessor and have relatively low input/output (I/O) and storage requirements.

For many applications, an end-to-end application speedup of 67 would be a reasonable expectation for a 128-processor cluster. A 128-processor cluster requires 10 times the initial capital investment of an FPGA coprocessor card, in addition to recurring costs for maintenance, cooling, electricity, and recycling. Many researchers have expressed the opinion that future HPC systems will actually require HPRC techniques to push high-performance computing into the peta-scale performance range while having feasible implementation and power costs [3]–[5]. In fact, Cray's current flagship line of supercomputers, the XT5 and XT5h families, include FPGA coprocessors integrated directly into each processing module [6].

While HPRC has been viewed as an efficient technique for scientific high-performance computing, to date there has been surprisingly little attention given to applying this technique to applications in computational biology. One reason for this is that there are currently only a small handful of universally accepted "textbook" algorithms in computational biology to target for acceleration. This is in stark contrast to fields such as signal processing where there is a well-established standard set of computational methods (i.e., filtering, DFT). However, a few computational biology tools, such as sequence alignment (i.e., BLAST, Smith–Waterman, ClustalW) and distance-based phylogeny reconstruction (i.e., UPGMA, neighbor-joining), have been successfully accelerated [7]–[10]. These computations are control-independent (their execution behavior is independent of the input data) and can generally can be implemented with a simple systolic array.

Our goal is to achieve acceleration of additional applications of in computational biology, particularly those that are extremely computationally expensive, require complex algorithms, and have complex execution behavior. In this paper, we focus on maximum parsimony phylogeny reconstruction for gene rearrangement data. This application consists of two primary components. The first is referred to as tree generation, which explores the space of phylogenetic trees for an optimal tree topology. The second component is tree scoring, which is used to determine the fitness of a particular tree. Tree scoring is an iterative refinement algorithm that repeatedly performs expensive median computations over the internal vertices of a given tree. Not every tree must be scored, but the scoring rate increases with the evolutionary rate of the input data set.

In other words, more difficult data sets are more complex to compute.

In this paper, we present our breakpoint median architecture. Our performance results demonstrate a 1005× speedup over software for the computation alone, and a 417× speedup when the architecture is used to accelerate the entire reconstruction procedure. Section II provides details of the application. Section III provides details of the breakpoint median computation. Section IV describes the breakpoint median hardware design. Section V describes how we finely parallelized the breakpoint median algorithm across FPGA resources, Section VI discusses the design's performance results and limitations, Section VII describes how we integrated the core into the overall application and Section VIII concludes this paper.

II. Gene Rearrangement Analysis

Recent advances in high-speed DNA sequencing have resulted in an explosion in genomic data. In addition, chromosomal and whole genome data are now represented as gene sequences instead of nucleotide sequences. A gene constitutes a basic unit of inheritance and represents a nucleotide sequence responsible for transcription of a specific protein. Genes are assigned a unique integer and can be positive or negative, depending on the gene's orientation (i.e., forward or backward). Rearrangement of gene sequences, such as inversions, transpositions, and other operations such as duplications, deletions, and insertions are known to be an important evolutionary mechanism [11]. Understanding these rearrangements is important in comparative genomics, systematic biology, gene prediction, and other analyses.

Pevzner provided the first method for computing the shortest sequence of rearrangements that would transform one gene ordering into another [12]. Moret's group later provided: 1) a linear-time algorithm to compute these distances [13]; 2) techniques to tackle the breakpoint median and inversion median problems [14]; 3) faster and better-scaling approaches to phylogenetic reconstruction [15]; and 4) the software package GRAPPA which has become one of the most accurate methods for reconstructing evolutionary histories [16]. The extension of genome rearrangements analysis under parsimony and other phylogenetic algorithms (GRAPPA) that uses the heuristic technique of disk-covering [17] (DCM-GRAPPA [18]) runs quickly on large datasets with more than 1000 genomes. As such, the size of the dataset is no longer the main issue. Rather it is the evolution rate of the input data that determines the computational complexity of gene order analysis.

A. Gene Rearrangement Data

We assume a reference set of n genes $\{g_1, g_2, \ldots, g_n\}$. Thus, a genome can be represented by an ordering of these genes and each gene is given with an orientation that is either positive, written g_i, or negative, written $-g_i$. A genome can be *linear* or *circular* and can undergo various rearrangement events such as *inversion*, *transposition*, *deletion*, *duplication*, etc.

Let G be the genome with signed ordering $g_1, g_2, \ldots, g_n$. An *inversion* (also called a *reversal*) between indices i and $j (i \leq j)$ produces the genome with linear ordering

$$g_1, g_2, \ldots, g_{i-1}, -g_j, -g_{j-1}, \ldots, -g_i, g_{j+1}, \ldots, g_n.$$

A *transposition* acts on three indices i, j, and k, with $i \leq j$ and $k \notin [i, j]$, picking up the interval $g_i, g_{i+1}, \ldots, g_j$ and inserting it immediately after g_k. Thus, genome G is replaced by (assume $k > j$)

$$g_1, \ldots, g_{i-1}, g_{j+1}, \ldots, g_k, g_i, g_{i+1}, \ldots, g_j, g_{k+1}, \ldots, g_n.$$

An *inverted transposition* is a transposition followed by an inversion of the transposed subsequence (it is sometimes called a *transversion*).

B. Phylogenetic Reconstruction

Phylogenetic analysis is the study of evolutionary relationships amongst a set of species. A *phylogeny* (or *phylogenetic tree*) is an unrooted binary tree where each vertex represents information associated with a species and each edge represents a series of evolutionary events that transformed one species into another. Analyzing phylogenies is a fundamental tool that biologists use to infer common characteristics across different species based on their evolutionary relatedness. Analysis of phylogenies is a vital component of research in such areas as drug and vaccine development and bio-pathway discovery.

As shown in Fig. 1, a phylogeny is an unrooted binary tree. Each of the n leaves has degree 1 and represents a species (taxon) that currently exists, while each of the $n - 2$ internal vertices has degree 3 and represents an extinct species that is a common ancestor. Each edge is associated with an evolutionary distance, representing the number of evolutionary events that separate the two corresponding species. Both the topology and the edge distances are important characteristics of the phylogeny.

In general, the problem of *phylogenetic reconstruction* can be summarized as such: given n input species, compute a phylogeny that most closely resembles the species' actual relative evolutionary history. Methods for reconstructing phylogenies include distance-based methods such as neighbor-joining [19] and direct optimization methods. The latter, pioneered by Sankoff and Blanchette in their package BPAnalysis [20] and improved by GRAPPA [16] and MGR [21], are among the most accurate methods. Direct optimization methods rely on finding median genomes. As shown in Fig. 2, the median problem on k genomes is to find a single genome that minimizes the median score (sum of the pairwise distances) between itself and each of the k given genomes.

GRAPPA [16] is an exhaustive search method, which moves systematically through the space of all $(2N-5) \times (2N-7) \times \bullet\bullet\bullet \times 3$ possible trees on N genomes. For each tree, the program tests a lower bound to determine whether the tree is worth scoring. For every tree that is scored, the program will iteratively solve the median problems at internal nodes until convergence, as outlined in Fig. 3. Labeling an internal vertex requires computing a median of three gene orders.

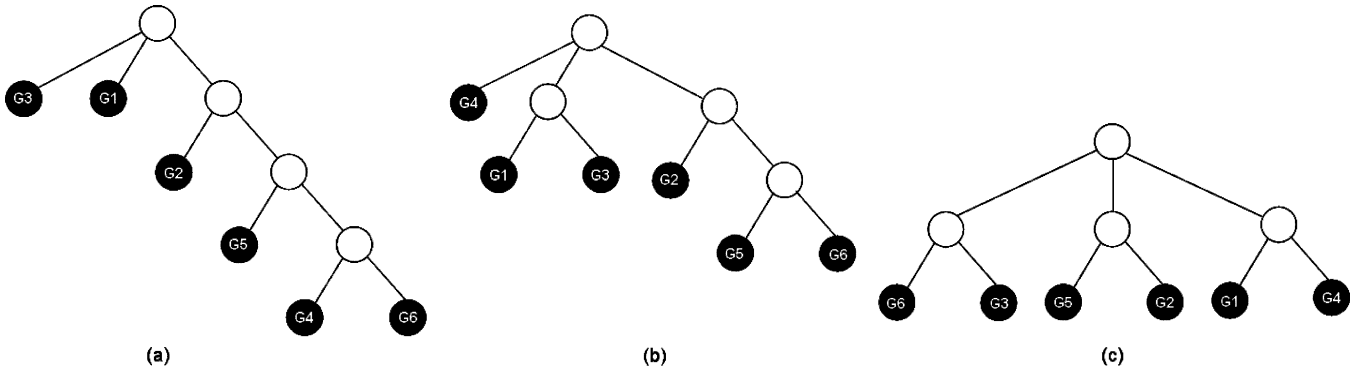

Fig. 1. Three of the 105 possible phylogenies for 6 input genomes. Input species $(G1, G2, \ldots, G6)$ are shown in black while ancestral species are shown in white.

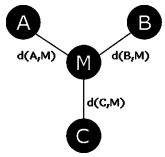

Fig. 2. Given genomes A, B, and C, the median problem is to find genome M that minimizes the median score, where the median score $= d(A, M) + d(B, M) + d(C, M)$, where $d()$ is an edit distance (i.e., breakpoint distance).

Initially label all internal nodes with gene orders
Repeat
  For each internal node v with neighbors A, B and C, do
    Solve median problem on A, B, C to yield m
    If relabeling v with m improves the tree score, then do it
Until no change occurs

Fig. 3. Tree scoring algorithm.

Computing an optimal median is NP-hard relative to sum of the edge lengths involved (i.e., the diameter of the inputs) [14], [22]. As a consequence, the portion of GRAPPA's total execution time that is spent labeling internal vertices of candidate trees sharply increases with the evolutionary rate of the input set.

III. Breakpoint Median

A. Breakpoint Distance

The *breakpoint distance* between signed genomes A and B is defined as the number of adjacent gene-pairs gh that appear in A when neither gh nor $-h - g$ appear in B. In other words, it is simply the number of adjacencies in one genome that is not in the other, irrespective of the relative orientation of the two genomes. For example, (circular) genomes $A = (1 \quad -2 \quad -3 \quad 4)$ and $B = (4 \quad 2 \quad -1 \quad -3)$ have a breakpoint distance of 2, because gene pairs $(-2 \quad -3)$ and $(4 \quad 1)$ appear in A but neither $[(-2 \quad -3)$ or $(3 \quad 2)]$ nor $[(4 \quad 1)$ or $(-1 \quad -4)]$ appear in B.

B. Breakpoint Median

Computing a *breakpoint median* for three genomes requires solving a traveling salesman problem (TSP) formulated in the following way [20]. Given genomes A, B, and C, each consisting of an ordering of n signed genes, construct a fully-connected undirected graph having vertices $V = \{-g_n, \ldots, -g_1, g_1, \ldots, g_n\}$ and define $w(g, h)$ to be the weight between vertices g and h.

For each gene g, define $w(g, -g)) = -\infty$. This is done to guarantee that each gene will appear alongside its reverse polarity counterpart in the TSP solution. Define $u(g, h)$ to be the number of times vertices $-g$ and h are adjacent in the three genomes, and define $w(g, h) = 3 - u(g, h)$.

If $s_1, -s_1, s_2, -s_2, \ldots, s_n, -s_n$ is the solution of the TSP, then the resultant breakpoint median is $m = s_1, s_2, \ldots, s_n$. This solution guarantees that $d(A, m) + d(B, m) + d(C, m)$ is optimally minimal, where $d(a, b)$ is the breakpoint distance.

C. TSP Example

Fig. 4 shows an example of breakpoint median problem. The three input genomes are shown on the upper-left of the figure. A graph is constructed consisting of cities that represent the positive and negative version of each gene. Since this is a TSP graph, the graph is fully connected. We initialize the edges between each gene and its negative counterpart to be weight $-\infty$. We begin with the assumption that each of the four gene adjacencies in each of the three genomes is unique, so we initialize all other edges to weight 2. Since gene adjacency -2 and $+3$ appear in two of the genomes, the edge between $+2$ and $+3$ becomes 1. Since gene adjacency -4 and -3 appear together in the first genome and gene adjacency $+3$ $+4$ appears in the second and third genome, the edge weight between genes -3 and 4 becomes 0 (-4 -3 and $+3$ $+4$ correspond to the same edge). When the TSP is solved, the median genome corresponds to every other city in the tour (even indices or odd indices).

D. Breakpoint Median Algorithm

In GRAPPA, the breakpoint median algorithm is performed using a depth-first branch-and-bound search. Its goal is to find a combination of edges that forms an optimal TSP tour.

1) Sorted Edge List: The search algorithm begins by reading the input genomes and constructing the resultant TSP graph. By definition, each edge in the graph has weight $-\infty$, 0, 1, 2, or 3. Once this is complete, it organizes the weight 0, 1, and 2 edges into a list sorted by edge weights. We refer to this as the *sorted edge list*.

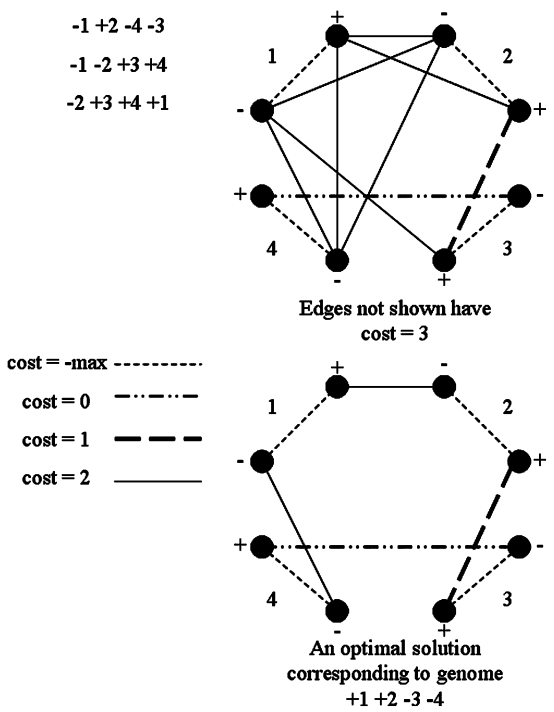

Fig. 4. Breakpoint median TSP formulation.

2) Partial Solution: The algorithm creates an empty edge set to serve as the current search state. We refer to as the *partial solution*. By definition, all weight $-\infty$ edges are guaranteed to be in the final solution, so the algorithm assumes that these edges are always part of the partial solution without explicitly adding them. As a result of this assumption, each vertex is implied to be already connected to another vertex before the search begins, so each vertex is effectively initialized with a degree of one, relative to the TSP tour.

3) Used Table: The search proceeds by inspecting each edge in the sorted edge list in order, and adds any edge to the partial solution that obeys two conditions. The first condition is that no edge may be added to the partial solution that causes any of the vertices in the partial tour (implied by the current partial solution) to have a degree of greater than two, since the salesman tour must not contain branches (i.e., the salesman cannot traverse a fork in the tour). In order to enforce this rule, a table is maintained that records whether the degree of each vertex is currently 1 or 2 with respect to the current partial solution. We call this the *used table*.

4) OtherEnd Table: The second condition is that no edge may be added to the partial solution that creates a cycle in the TSP tour, unless the addition of that edge results in a full tour and, therefore, includes all vertices (i.e., the salesman visits all cities). In order to enforce this rule without having to repeatedly traverse the tour implied by the partial solution, another table is maintained that keeps track of the end-points for partial tour fragments. For example, if edge $(-2, 3)$ is added, followed by edge (2,5), this would result in the following implied partial tour: $-5, 5, 2, -2, 3, -3$. The table records which vertex is at the opposite end of each partial tour fragment. We refer to this as the *otherEnd table*. It is initialized with $\mathrm{otherEnd}(a) = -a$. Each time an edge (a, b) is added, the table is updated as such

$$\mathrm{otherEnd}(\mathrm{otherEnd}(a)) = \mathrm{otherEnd}(b)$$
$$\mathrm{otherEnd}(\mathrm{otherEnd}(b)) = \mathrm{otherEnd}(a).$$

5) Upper Bound: If no edges remain that satisfy these conditions, from the current point forward in the sorted edge list, the algorithm will record the tour implied by the partial solution as a best-found-so-far solution if its score (including any implicit weight-3 edges that must be included to complete the tour) is less then the current upper bound. The *upper bound* is the tour cost of the currently best found tour. Note that the tour cost also corresponds to the breakpoint median score, disregarding the $-\infty$ edges.

6) Lower Bound: Any time the partial solution is modified (by adding or pruning an edge) the algorithm computes a *lower bound* for the partial solution. If the lower bound exceeds the score of the upper bound, the last added edge is pruned.

The search computes the lower bound using the following technique [23]. First, initialize the lower bound to zero. Then, for each vertex that currently has a degree of one in the current partial solution, add the weight of the lowest-weighted available edge that leads to another vertex of degree one (note that this value may be 3).

The lower bound computation disregards any edges that: 1) were previously pruned at or above the current level in the search tree or 2) would result in a tour cycle if the edge were added to the partial solution (unless the cycle includes all vertices).

The lower bound computation must therefore reference the *used table*, *otherEnd table*, and another table called *excluded edges* which we describe in the following.

Once the lower bound accumulation is complete, the value is divided by two to account for the bidirectionality of the edges. This value represents a greedy approach to find the lower bound for the cost of completing the current partial tour. This value is added to the current cost of the partial tour and compared to the cost of the lowest cost tour that was previously found (this value is initialized with the lowest median score among the three input genomes).

7) Excluded Edges: In order to keep track of which edges have been previously pruned under the current search state, another table is maintained that we refer to as the *excluded edges table*. Each time an edge is pruned it is marked as excluded.

If edge A is pruned, and edge A was added to the partial solution before edge B was pruned, then edge B must be unexcluded, since edge B was pruned based on the state of the partial solution which included edge A. Since edge A is no longer in the partial solution, edge B may be re-added to the partial solution in subsequent lower bound computations.

Each time an edge is pruned, the search state, which now includes the *partial solution*, *the used array*, *the otherEnd table*, and the *excluded table*, must be restored to the state before the pruned edge was originally added, except for the edge's new status as being excluded. A stack is used to keep track of this information.

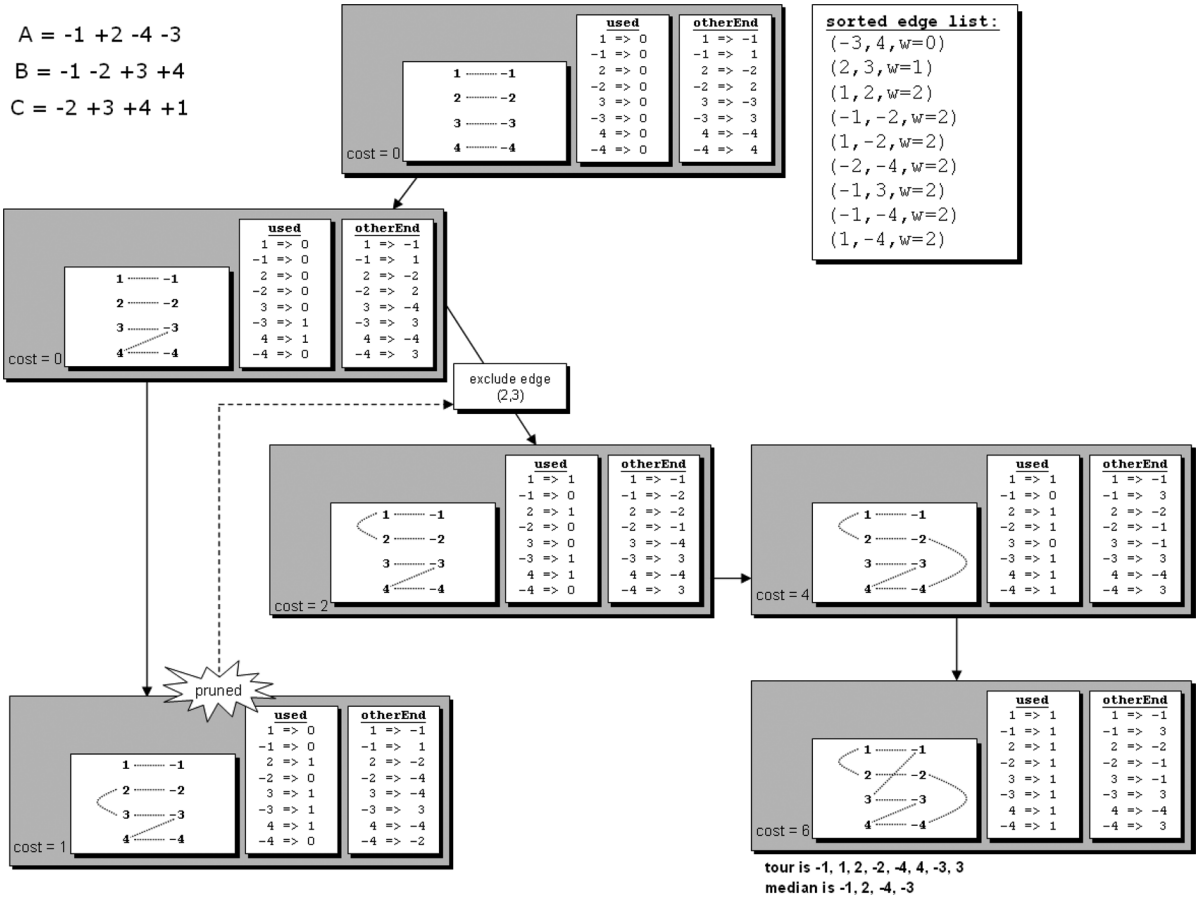

Fig. 5. Graphical representation of a breakpoint median TSP depth-first search tree and associated data structures. Pruned edges are excluded from the lower bound computation from the level they are pruned to the bottom of the tree, the "otherEnd" array stores TSP path end-points to prevent cycles that do not include all vertices, and the "used" array keeps track of which vertices in the current solution state have degree 2.

E. Example Search

Fig. 5 shows a four-gene example of the optimal TSP search. Note that the figure is intended to resemble a depth-first search tree, but due to the limited space the tree is trivially small (and we were are unable to also show the contents of the stack).

The sorted edge list for the TSP graph is shown on the right. The root of the search tree is the partial solution containing no edges except the implicit edges between each gene and its negative counterpart.

The search begins by adding the first and second edges in the sorted edge list. Assume for this example that the lower bound computation causes edge (2,3) to be pruned and thus added to the excluded edge table. Using the stack, the search backs up to the point before edge (2,3) was added and continues by adding the next edge (1,2). The following edge $(-1, -2)$ cannot be added because it would create a cycle. Edge (1, -2) cannot be added because vertex 1 is "used". However, the search is able to add the next two edges $(-2, -4)$ and $(-1, 3)$. At this point, all vertices are used, and the search reaches the end of the sorted edge list.

Assume that the cost of this state is less then the current upper bound, and therefore the current solution is recorded as the "best-so-far" and its cost is recorded as the new upper bound.

F. Use of an Optimal TSP Solver

This algorithm computes an optimal TSP solution. High accuracy is the primary selling point of the "direct optimization" approach to reconstructing parsimony phylogenies. As such, the use of an optimal median solver is crucial, and heuristic solvers are generally avoided in this technique.

Essentially, the breakpoint median algorithm exhaustively tests every valid combination of edges from the edge list but includes a tightened lower bound to minimize the volume of explored search space (through the use of the excluded edge and otherEnd tables). Despite these optimizations, this algorithm is of course still NP-hard in the total number of edges in the TSP graph (i.e., length of the sorted edge list), which ultimately is a function of the sum of breakpoints among the three input gene orders $(d(A, B) + d(A, C) + d(B, C))$. Even so, the optimal breakpoint median has been shown to be tractable for problem sizes that are of practical interest to biologists [15].

IV. Breakpoint Median Core

We designed our breakpoint median "core" using approximately 10,000 lines of custom-written VHDL. In this context, a core is discrete digital circuit and is analogous to a general-purpose CPU executing an equivalent computation as software code. Traditional terms for what we describe include *processing element (PE)* and *intellectual property (IP) block*, although in

this context we believe *core* is the most accurate. Our use of the term "core" is a reference to a processor core of a multi-core CPU (chip multi-processor).

Note that the breakpoint median algorithm was carefully designed to utilize the small number of choices for the pairwise costs, thus, further significant speedup on the software implementation is very difficult if not impossible. Also note that although there has been previous work in designing FPGA architectures for the TSP problem, to our knowledge all of this work involved approximate solvers using genetic algorithms [24]–[26]. Since our goal is to find exact solutions of the breakpoint, this previous work is not directly applicable to this application.

A. Tasks Performed in Software

Before the median core begins operation, the host system, in software, performs several initialization tasks. First, it computes an initial upper bound using the three input genomes. Recall that the tour costs used in the TSP are equivalent to the breakpoint score. Therefore, the initial upper bound is determined by finding which of the three input genomes has a minimum breakpoint score itself, which is calculated as the sum of breakpoint distances to the other two inputs. This value is used as the initial upper bound, i.e., $\{\min(d(A,B)+d(A,C)),(d(A,B)+d(B,C)),(d(A,C)+d(B,C))\}$.

In our current-generation core design, the host (in software) also constructs the TSP graph from the input genomes and organizes the weight-0, weight-1, and weight-2 edges in the sorted edge list. This sorted edge list, instead of the actual input genomes, is used as the input for the median core on the FPGA. This is in contracts to our original core design, where the host transmitted the actual input genomes to the core and the core constructed the graph and sorted edge list entirely in hardware. This reduced communication overhead, but it required a considerable amount of hardware overhead. It also made it impossible for the host to force each core to use unique orderings for the sorted edge list, which is an important component of our current architecture.

The last task for the host is to perform a programmed I/O write operation to transmit the initial upper bound and sorted edge list into specific on-chip memory addresses corresponding to a specific median core on the FPGA. Using a programmed I/O read operation, the host can poll any core on the FPGA to determine its execution state, allowing the host to determine when any particular core has completed execution. When this occurs, the host performs another programmed I/O read operation to read the result genome from that core (the core converts the optimal TSP tour to a genome).

B. Top-Level Core Design

As shown in Fig. 6, the median core design consists of a block of control logic that is interconnected to a set of on-chip block RAMs (BRAMs) that are used to store the search state and stack. The controller is designed as a finite state machine with integrated multiplexers that establish datapaths among the set of BRAMs and registers. As such, the operation of the controller is inherently sequential, unlike traditional special-purpose architectures that are based on pipelines or systolic arrays. Instead,

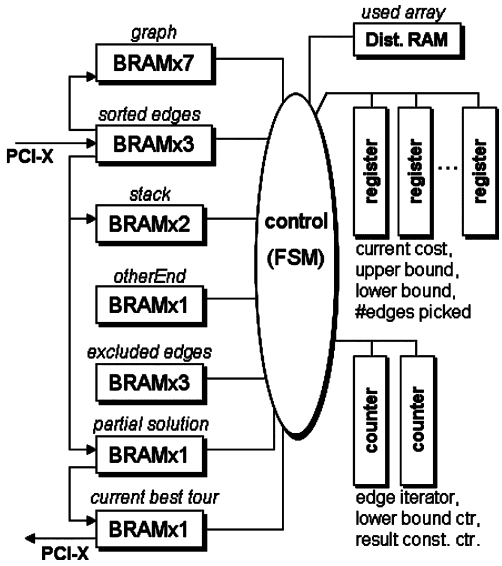

Fig. 6. Simplified block diagram for the breakpoint median core. The core design is a large sequential logic circuit that establishes datapaths among several memory elements in each clock cycle. Static interconnects between memory elements are shown, while multiplexed interconnects among memory elements are established through the control unit (integrated within its output logic).

our breakpoint median architecture achieves parallelization by performing many parallel memory accesses in each clock cycle, since the breakpoint median algorithm is bound by memory access as opposed to traditional special-purpose architectures that rely on parallel arithmetic.

For example, before adding an edge to the partial solution, the breakpoint median algorithm accesses several memories that hold information about the search state and graph (i.e., otherEnd array, excluded edges, used array, sorted edge list, etc.). In software, the parallelism of these memory accesses is limited by the number of read ports to the CPU's cache (or available number of load/store units). In our architecture, we exploit this parallelism to its maximal degree by accessing all required memories in parallel within a single clock cycle.

Fig. 7 shows a simple representation of the finite state machine controller. During operation, the controller alternates between adding an edge, computing the lower bound, and sometimes pruning. Note that the lower bound "loop" is implemented with counters. The "build graph" state is where the controller uses the sorted edge list to construct an alternative (but equivalent) TSP graph representation for use in the lower bound computation, as described in Section V.

The median core is capable of computing breakpoint medians of any reasonable size using only on-chip BRAM memory.

V. Extracting Parallelism

In general, when parallelizing an application (either for cluster/message-passing or SMP/shared-memory), the programmer must manually identify and extract parallelism to divide work among several processors. This type of manual parallelization must also be performed on an FPGA, although it includes the added complication of hardware design. In contrast to traditional techniques for parallelizing code, FPGAs allow fine-grain parallelism such as the ability to perform independent

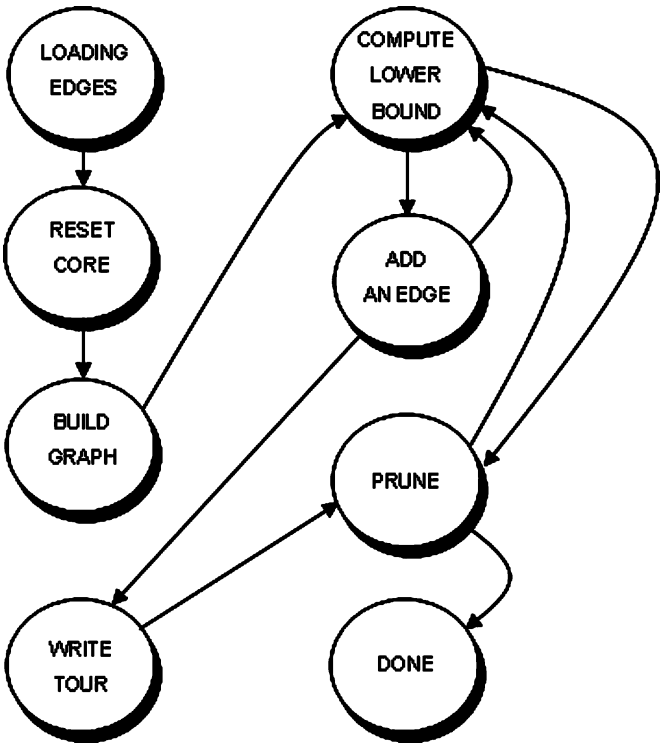

Fig. 7. Finite-state machine representation of median core controller.

arithmetic operations in parallel. For the breakpoint median we exploit both fine- and coarse-grain parallelism.

A. First-Generation Architecture

At this point, we have designed and characterized two different versions of our breakpoint median architecture, where each exploits parallelism using two different strategies.

In our original architecture [27], [28], we used a coarse-grain approach to parallelize the breakpoint median computation. In this approach, we used multiple cores in parallel to perform a single median computation. In this strategy, if the initial upper bound inferred from the input genomes is s, initialize n median cores with initial upper bounds $s, s-1, s-2, \ldots, s-(n-1)$. If the optimal median has score opt and $\mathrm{opt} < s$, the core with initial upper bound $\min(\mathrm{opt}+1, s-(n-1))$ will converge on the optimal solution fastest. In other words, the core having the lowest initial upper bound that is greater than the optimal median score will always finish first (communication overhead required to initialize multiple cores notwithstanding). After the first core completes its search having found a solution, the host will terminate the search on the other cores.

This strategy yielded a maximum average speedup of 26× for the breakpoint median alone and 24× for the entire phylogeny reconstruction procedure after replacing the software version of the breakpoint median with the hardware version. This strategy had a point of diminishing returns at 12 cores, which is approximately 50% of the available FPGA resources (for our Virtex-2 Pro 100 FPGA).

B. Second-Generation Architecture

Our current architecture takes a significantly different approach for extracting parallelism from the breakpoint median algorithm. We made three changes to the hardware design.

First, we redesigned the core into a much lighter-weight version, where TSP graph construction was moved from hardware into software. This version of the core design only implements the combinatorial search portion of the median algorithm. This allowed multiple cores to be initialized such that the sorted edge list (representing the graph implied by the inputs) is constructed uniquely for each core. This forced each core to explore its search space using a unique order of depth-first search paths, ultimately allowing one core to find the solution faster than the others.

Second, a broadcast communication mechanism was added that allows the cores to communicate for the purpose of maintaining a globally shared upper bound (best score found-so-far) value.

Third, the lower bound computation was parallelized, where the lower bound "loop" was effectively unrolled and scanned by multiple "lower bound units" in parallel.

C. Extracting Fine-Grain Parallelism

Adding or pruning an edge for the current partial solution is an inexpensive operation, requiring 2–8 clock cycles. On the other hand, computing the lower bound requires a traversal of the entire TSP graph. As such, the median core spends nearly all of its execution time performing lower bound computations. Fortunately, the lower bound computation consists of a bounded loop and each loop iteration is data-independent. As a result, the lower bound contains fine-grain parallelism, and we exploit this parallelism in the median core design. The lower bound computation is parallelized by replicating both the TSP graph and search state into multiple memories that can be read in parallel by "lower bound units". In other words, we parallelized the lower bound computation by "unrolling" the lower bound loop and inspecting multiple graph vertices in parallel (and, thus, performing multiple iterations of the loop in parallel).

For the lower bound, the TSP graph is represented as an array where each entry in the array represents a vertex (city) and stores one to three possible edges that connect that vertex. Note that this data structure and the sorted edge list both represent the same TSP graph. The median core reconstructs this representation from the sorted edge list provided by the host (our previous core design constructed this representation and then constructed the sorted edge list from it).

The operation of lower bound unit is illustrated in Fig. 8. In this example, a lower bound unit inspects vertex 2 and its three edges (2,11), (2, −19), and (2, −49).

If the degree of vertex 2 is one in the partial solution (i.e., not *used*), the lower bound unit must choose the lowest weight of these edges with the requirement that the edge it chooses must: 1) connect to another unused vertex (requires a copy of the *used table*); 2) must not be excluded (requires a copy of the *excluded edge table*); and 3) must not create a cycle unless the cycle includes all vertices (requires a copy of the *otherEnd array*). If no edges fulfill these criteria, the lower bound value is incremented by an implicit weight 3 (the TSP is fully connected,

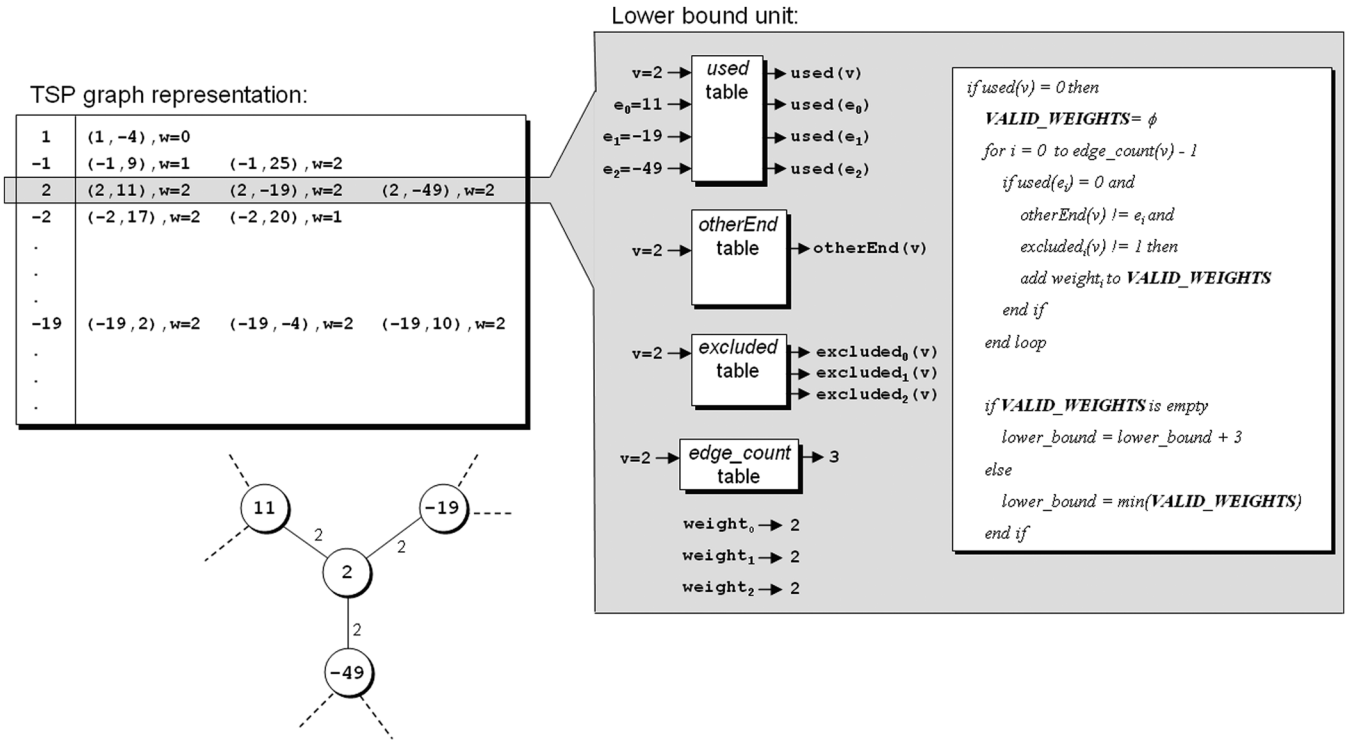

Fig. 8. Illustration of the operation of a lower bound unit and a portion of the TSP group focusing on vertex 2. During the lower bound computation, the TSP graph (contructed from the sorted edge list) is scanned in parallel by mutliple lower bound units. In this example, a lower bound unit is inspecting vertex 2, which has three weight-2 edges to vertices 11, -19, and -49. Since vertex 2 is unused in the current solution state, the lower bound unit must add to the lower bound value the minimum edge weight of the edges that: 1) lead to another vertex that is not used and 2) do not form a tour cycle (unless the cycle includes all vertices), and has not been excluded (pruned at or above the current level in the search tree).

but only edges with weights 0, 1, and 2 are included in the graph representation). Otherwise, the chosen edge's weight is added to the lower bound value.

If the degree of vertex 2 is two in the partial solution (i.e., already *used*), the lower bound unit would not increment the lower bound.

Note that a copy of the **edge_count** table is also needed, which stores the number of edges for each vertex. This table is part of the TSP graph representation.

Since each of the FPGA's on-chip memories has two ports, we need to replicate copies of the necessary memories $n/2$ times (each memory maintained as exact copies), allowing the design to perform n reads per clock cycle. We refer this as a core having n "lower bound units".

This approach requires a significant number of BRAMs, which limits the number of lower bound units per core. This approach also requires a significant amount of routing complexity for synthesis, which effectively increases the minimum clock period after logic synthesis and place-and-route. We decided to stop adding additional lower bound units after a 15% reduction in clock speed (relative to having one lower bound unit). This occurred after we had reached 20 lower bound units.

D. Extracting Coarse-Grain Parallelism

The median computation also has potential for coarse-grain parallelism, such as the ability to use multiple parallel median cores to perform a single median computation. Note that this was the only approach we used in our original median architecture. Our improved architecture uses a different technique.

The ideal technique for extracting coarse-grain parallelism in a multiple core arrangement is to dynamically partition the search tree, have each core search disjoint sub-trees, and have all cores share a global upper bound value. There are several reasons why it is not practical to use this technique for parallel median cores.

First, the breakpoint median algorithm has unpredictable pruning behavior. If multiple cores were used to search disjoint regions of the search tree, they would need to be load balanced. In other words, even it were possible to initially assign each core an equal-sized region of the search tree, core A may be able to quickly prune its entire assigned sub-tree while core B would be forced to do an exhaustive exploration of its assigned sub-tree. Core B would need to realize that core A is currently idle, re-partition its assigned sub-tree, and transmit a new search state and boundary conditions to core A in order for it to begin a search its newly assigned sub-tree. This behavior would be: 1) very complex to implement in custom hardware; 2) extremely difficult to fully verify; 3) require substantial hardware and routing overhead; and 4) require high communication overhead. In order to demonstrate point #4, assume that core A's newly assigned sub-tree is also pruned immediately. The re-partitioning (load balancing) process would need to be repeated and would lead to core B spending a significant amount of time re-partitioning rather than searching. Depending on the search behavior, it's evident that the performance improvement may be negated by the overheads.

Second, recall that the full search state includes the following memories: *partial solution (edge set)*, *used array*, *otherEnd*

TABLE I
PERFORMANCE RESULTS FOR THE MEDIAN ARCHITECTURE FOR 1000 MEDIAN EXECUTIONS. ARITHMETIC AND GEOMETRIC MEANS ARE PROVIDED FOR EACH SET OF TESTS

Average Events Per Edge	Average Speedup 10 Lower Bound Units								Average Speedup 20 Lower Bound Units							
	1 core		2 cores		3 cores		4 cores		1 core		2 cores		3 cores		4 cores	
	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$	Σ/n	$\Pi^{1/n}$
17	7.213	3.517	9.480	4.287	9.278	4.140	9.954	4.491	8.757	4.063	10.659	4.737	11.762	5.164	10.864	4.878
18	8.113	3.751	10.875	4.916	10.221	4.688	11.720	5.151	9.652	4.373	12.163	5.491	13.272	5.996	16.583	5.741
19	12.308	4.195	17.986	5.574	18.305	5.461	21.011	6.030	15.044	4.989	21.387	6.413	23.859	7.212	23.624	6.707
20	12.061	4.724	51.971	6.837	48.023	6.773	52.877	7.579	14.507	5.713	53.880	7.983	56.233	9.276	61.392	8.717
21	21.768	6.095	41.680	8.687	38.372	8.614	43.309	9.748	26.749	7.465	45.546	10.392	51.610	12.230	48.719	11.275
22	36.394	6.529	76.261	10.268	70.928	10.537	83.459	12.129	45.961	8.068	89.814	12.555	95.277	15.364	98.204	14.274
23	42.354	7.765	111.544	11.050	142.996	13.928	192.664	15.656	53.535	9.702	141.927	14.527	168.046	18.153	193.127	17.000
24	74.593	8.647	129.327	12.769	140.728	16.367	148.314	18.911	90.092	10.829	159.567	17.429	165.511	21.038	172.714	20.926
25	77.135	10.478	738.574	15.169	794.870	19.676	810.267	23.103	123.234	17.042	952.335	21.218	1004.998	25.267	917.852	25.594

array, *excluded array*, and *stack*. In order to instruct a core to begin searching at a specified point in the search tree, the contents of all of these memories would need to be transmitted between cores. Aside from requiring a large switching network, this would increase the fan-in and fan-out at each of the core BRAMs and further increase place and route complexity.

For these reasons, we have adopted a simple technique for partitioning the search tree. In our technique, each core is initialized to search the entire search tree, but each explores the tree in a different order. Because each parallel core explores the tree along a unique search path, each core's upper bound will be decremented at a different rate. Since the upper bound is globally shared among all cores, each core will therefore prune different regions of the search tree and the tree therefore will be dynamically partitioned among the cores. We refer to this as "virtual search space partitioning". In other words, our current technique for exploiting coarse-grain parallelism relies on two concepts. The first concept is to force each core to explore an identical TSP search space using a unique search order. To do this, we initialize each core with the same sorted edge list but equal-weighted edges are ordered differently.

The second concept is to allow the cores to communicate with each other in order to maintain a global minimum upper bound value. Before each core computes its lower bound, it compares its current local upper bound with the lowest upper bound among all parallel cores. If this global upper bound is less than the core's local upper bound, the core adjusts its local upper bound to become the global upper bound plus one. Adding one to the local copy of the global minimum prevents any core from pruning the optimal solution, guaranteeing that all cores are capable of finding the optimal solution. Since each of the cores will eventually find the optimal solution, the median computation is considered complete when the first core completes its search.

VI. MEDIAN CORE PERFORMANCE RESULTS AND LIMITATIONS

A. Test HPRC System

Our test system consists of a Dell Precision 650 server containing a 3.06-GHz Intel Pentium Xeon processor. The FPGA accelerator card is an Annapolis Microsystems Wild-Star II Pro card with a single Xilinx Virtex-2 Pro 100 FPGA. It is connected to the host though a PCI-X interconnect.

B. Test Methodology

In order to determine the hardware speedup, we generated 1000 random three-leaf phylogenies and extracted the leaves to use as the median inputs. The number of rearrangement events along each edge for each phylogeny is chosen from a uniform random distribution with range *distance* ± 3, where *distance* is a parameter for the experiment.

For each set of genomes, we invoke the breakpoint median routine **bbtsp** (the software implementation of the breakpoint median that is integrated within GRAPPA) and record its execution time. We then dispatch the same three genomes to the FPGA's breakpoint median architecture and record its execution time. Note the FPGA execution time includes the CPU-to-FPGA communication time and the time required for the host to construct the TSP graph, construct the corresponding sorted edge list(s), and compute the initial best score (these initialization tasks occur in software, even for the FPGA-based median computation).

For each three-genome input, we measure speedup in the traditional way, i.e., $\mathrm{time}_{\mathrm{sw}}/\mathrm{time}_{\mathrm{hw}}$. A speedup of 1 would indicate equivalent performance between the software median computation and hardware median computation. Because the median's execution time is dependent on the inputs, and since the inputs are random, our results list both the arithmetic mean and the geometric mean over the set of 1000 individual median computations.

C. Performance Results and Discussion

Table I lists performance results of the median architecture against software. We tested the following eight configurations of the median architecture: one, two, three, and four median cores having 10 lower bound units and 20 lower bounds units. These results include software initialization time and host-FPGA communication time.

The first column, labelled "Average Events per Edge" is the *distance* parameter for the random input generation. This represents the diameter, or relatedness, of the inputs. For each configuration and each input diameter, the column labeled "Σ/n" lists the arithmetic mean and the column labeled "$\Pi^{1/n}$" lists the geometric mean for the speedups across the 1000 random three-genome inputs.

The results show a trend where the average speedup increases non-linearly with the diameter of the inputs. To explain this, recall that the size of the search tree scales exponentially with the size of the sorted edge list, which itself scales linearly with the sum of breakpoints between the three input genomes $(d(g_1, g_2) + d(g_1, g_3) + d(g_2, g_3))$. As a result, the time spent searching increases with the evolutionary diameter of the inputs. The ratio between the time spent searching and the time spent in startup costs also increases with the evolutionary diameter. Since the startup costs are fixed for any input, median computations that require longer searches spend a larger relative amount of time searching and thus result in a higher speedup. The speedup trend shown in the results indicate that the median architecture has a faster search rate than the software implementation.

The performance results also show a drastic performance improvement when the architecture is scaled from one to two cores, but only minor improvements (or slowdowns for low diameter input sets) from adding additional cores. We believe this is caused by two factors.

The first factor is communication overhead. We minimize communication overhead by forcing each core to begin execution immediately after it receives the sorted edge list and initial upper bound. In other words, the core n begins execution as core $n+1$ is receiving its inputs, effectively overlapping execution and host-FPGA communication. However, there are cases where a core completes execution while the host is transmitting input data to or polling another core, which results in reduced performance.

The second factor is more difficult to characterize, but is evidently caused by a point of diminishing returns in our virtual search partitioning technique. The edge list sent to the first core is sorted deterministically according to the ordering of the input genomes. The edge list sent to the second core is sorted such that equal-weight edges are reversed relative to the first core's edge list. This causes the first and second cores to follow significantly different search orders and results in a large speedup. Beyond two, additional cores are initialized by scrambling equal-weight edges in the original sorted edge list. These additional edge lists result in different search orders, but in most cases the search order is not different enough to yield significant improvement.

VII. Accelerated-GRAPPA

We made modifications to the GRAPPA code to accelerate the tree scoring procedure by forcing it to dispatch all its median computations to the (4-core, 20 lower bound per core) median architecture on the FPGA.

A. Performance Results

Table II shows our average speedups relative to the all-software GRAPPA for end-to-end runs over 10 unique 8-leaf synthetic datasets. As before, each set of input genomes were produced by synthesizing phylogenies using a specified average number of rearrangement events per edge and extracting the leaves. In this case, however, the synthesized phylogenies have eight leaves instead of three, as in the median performance tests. The results shown are the average speedups, using both arithmetic and geometric mean, across ten runs for each input diameter. For each dataset, we also list the arithmetic mean of the individual software execution times when run without acceleration.

TABLE II
Performance Results for Accelerated-GRAPPA

Average Events per Edge	Average SW Execution Time	Application Speedup	
		Σ/n	$\Pi^{1/n}$
10	36 seconds	12.2433	2.8905
11	1.5 minutes	15.5771	3.3484
12	18 minutes	391.1187	25.3422
13	2.5 hours	416.8365	74.2543

As with the breakpoint median performance results, the results show a clear non-linear trend where the average speedup increases with the evolution rate of the input set. There are two reasons for this. First, higher evolutionary rate input sets force GRAPPA to spend higher portions of its total execution time computing medians. Thus, accelerating the median computation has a higher impact on overall application speedup. Second, the median computations themselves are more greatly accelerated as the average diameter of the median inputs increase, which increases with the evolutionary rate of the inputs. Note that execution time increases with the evolution rate of the inputs, so we achieve increasingly higher speedups for datasets that require increasingly more time to compute.

B. Discussion

In all of our tests, we used synthetic input data, as this gives us tight control over its characteristics and allows us to relate these characteristics with observed performance. The characteristics of actual biological data greatly depend on the type of genomic data (i.e., mitochondrial, nuclear, chloroplast, etc.), the type of species being analyzed (i.e., prokaryote, eukaryote, etc.), and the evolutionary relatedness of the species in the dataset. As a result, it is impossible to directly relate our performance results with biological examples. However, we know from the literature that the number of evolutionary events in our test inputs are conservative compared to a biological case study for *Drosophila* (fruit fly genus) [29]. In this study, a phylogenetic reconstruction (based on gene rearrangements) for set of eight fly species across a common genus resulted in a phylogeny whose edge lengths ranged from 15 to 565 breakpoints (an estimation of rearrangement events).

VIII. Conclusion and Future Work

Our results indicate that our accelerated GRAPPA is capable of achieving an order 100 speedup for input sets that have a relatively high evolution rate.

We are currently developing a *tree generation and bounding* core that performs parallelized tree space exploration. This will allow us to combine tree generation and bounding cores with median cores on a single FPGA, allowing candidate trees from any of the tree generation and bounding cores to be scored with median cores on the same FPGA. This will also allow communication between the tree generation cores and median cores to

be performed entirely on-chip, reducing host-FPGA communication overhead.

Having the ability to accelerate both median computation and tree generation will allow us to generate accelerator architectures that are customized to the inputs. In other words, since GRAPPA's performance bottleneck is tree generation for closely related data sets and median computation for distantly related data sets, we will have the ability to tailor the accelerator architecture to the characteristics of the data set.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful comments that allowed for significant improvements to this paper.

REFERENCES

[1] M. P. de Moraes Zamith, E. W. G. Clua, A. Conci, A. Montenegro, P. A. Pagliosa, and L. Valente, "Parallel processing between GPU and CPU: Concepts in a game architecture," in *Proc. Comput. Graph., Imag. Visualization (CGIV)*, Aug. 2007, pp. 115–120.

[2] B. Pieters, D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, "Motion compensation and reconstruction of H.264/AVC video bitstreams using the GPU," in *Proc. 8th Int. Workshop Image Analy. Multimedia Interactive Services (WIAMIS)*, Jun. 2007, pp. 69–72.

[3] P. K. Agarwal and S. R. Alam, "Biomolecular simulations on petascale: Promises and challenges," in *J. Phys.: Conf. Series*, 9, 2006, vol. 46, pp. 327–333.

[4] R. Sass, W. Kritikos, A. Schmidt, S. Beeravolu, P. Beeraka, K. Datta, D. Andrews, R. Miller, and D. Stanzione, Jr, "Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing," in *Proc. IEEE Symp. Field Program. Custom Comput. Mach.*, Apr. 2007, pp. 127–138.

[5] T. El-Ghazawi, "Is high-performance, reconfigurable computing the next supercomputing paradigm?," presented at the ACM/IEEE Supercomput. Conf. (SC), Tampa, FL, 2006.

[6] 2007 [Online]. Available: http://www.cray.com

[7] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the BLAST algorithm," *J. VLSI Signal Process.*, vol. 48, no. 3, pp. 189–208, Sep. 2007.

[8] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong, *A Smith-Waterman Systolic Cell*. New York: Springer-Verlag, 2003, vol. 2778, Lecture Notes in Computer Science, pp. 375–384.

[9] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," *Bioinformatics*, vol. 21, no. 16, pp. 3431–3432, 2005.

[10] J. P. Davis, S. Akella, and P. H. Waddell, "Accelerating phylogenetics computing on the desktop: Experiments with executing UPGMA in programmable logic," in *Proc. 26th Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. (IEMBS)*, 2004, pp. 2864–2868.

[11] M. Blanchette, T. Kunisawa, and D. Sankoff, "Parametric genome rearrangement," *Gene*, vol. 172, pp. GC11–GC17, 1996.

[12] G. Bourque and P. Pevzner, "Genome-scale evolution: Reconstructing gene orders in the ancestral species," *Genome Res.*, vol. 12, pp. 26–36, 2002.

[13] D. A. Bader, B. M. E. Moret, and M. Yan, "A fast linear-time algorithm for inversion distance with an experimental comparison," *J. Comput. Biol.*, vol. 85, pp. 483–491, 2001.

[14] A. Siepel and B. M. E. Moret, "Finding an optimal inversion median: Experimental results," in *Proc. 1st Workshop Algs. Bioinformatics (WABI)*, 2001, vol. 2149, pp. 189–203.

[15] B. M. E. Moret, J. Tang, L.-S. Wang, and T. Warnow, "Steps toward accurate reconstructions of phylogenies from gene-order data," *Comput. Syst. Sci.*, vol. 65, no. 3, pp. 508–525, 2002.

[16] B. M. E. Moret, J. Tang, and T. Warnow, "Reconstructing phylogenies from gene-content and gene-order data," in *Mathematics of Evolution and Phylogeny*, O. Gascuel, Ed. Oxford, U.K.: Oxford Univ. Press, 2005, pp. 321–352.

[17] D. Huson, S. Nettles, and T. Warnow, "Disk-covering, a fast converging method for phylogenetic tree reconstruction," *J. Comput. Biol.*, vol. 6, no. 3, pp. 369–386, 1999.

[18] J. Tang and B. M. E. Moret, "Scaling up accurate phylogenetic reconstruction from gene-order data," in *Proc. 11th Conf. Intell. Syst. Mol. Biol. Bioinform. (ISMB)*, vol. 19, pp. i305–i312.

[19] N. Saitou and N. Nei, "The neighbor-joining method: A new method for reconstrucing phylogenetic trees," *Mol. Biol. Evol.*, vol. 4, pp. 406–425, 1987.

[20] M. Blanchette, G. Bourque, and D. Sankoff, "Breakpoint phylogenies," in *Genome Informatics 1997*, S. Miyano, T. Takagi, and editors, Eds. Tokyo: Univ. Academy Press, 1997, pp. 25–34.

[21] G. Bourque and P. Pevzner, "Genome-scale evolution: Reconstructing gene orders in the ancestral species," *Genome Res.*, vol. 12, pp. 26–36, 2002.

[22] I. Pe'er and R. Shamir, "The median problems for breakpoints are $\{NP\}$-complete," *Elec. Colloq. Comput. Complexity*, vol. 71, 1998.

[23] M. Blanchette, G. Bourque, and D. Sankoff, "Breakpoint phylogenies," *Genome Inform.*, pp. 25–34, 1997.

[24] M. A. Vega-Rodriguez, R. Gutierrez-Gil, J. M. Avila-Roman, J. M. Sanchez-Perez, and J. A. Gomez-Pulido, "Genetic algorithms using parallelism and FPGAs: The TSP as case study," in *Proc. Int. Conf. Workshops Parallel Process. Workshops (ICPP)*, Jun. 2005, pp. 573–579.

[25] P. Graham and B. Nelson, "A hardware genetic algorithm for the traveling salesman problem on splash 2," in *Proc. 5th Int. Workshop Field Program. Logic Appl.*, Oxford, U.K., Aug. 1995, pp. 352–361.

[26] I. Skliarova and A. B. Ferrari, "FPGA-Based implementation of genetic algorithm for the traveling salesman problem and its industrial application," in *Proc. Appl. Artif. Intell. Expert Syst. (IEA/AIE)*, Cairns, Australia, Jun. 2002, pp. 19–34.

[27] J. D. Bakos, "FPGA acceleration of gene rearrangement analysis," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2007, pp. 85–94.

[28] J. D. Bakos, P. E. Elenis, and J. Tang, "FPGA acceleration of phylogeny reconstruction for whole genome data," presented at the 7th IEEE Int. Symp. Bioinform. Bioeng., Boston, MA, Oct. 2007.

[29] A. Bhutkar, W. M. Gelbart, and T. F. Smith, "Inferring genome-scale rearrangement phylogeny and ancestral gene order: A Drosophila case study," *Genome Biol.*, vol. 8, pp. R236–R236, 2007.

Jason D. Bakos (M'96) received the B.S. degree in computer science from Youngstown State University, Youngstown, OH, in 1999, and the Ph.D. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, in 2005.

He worked as a Research and Teaching Assistant with the University of Pittsburgh from 1999 to 2005. He is currently an Assistant Professor with the Department of Computer Science and Engineering, University of South Carolina, Columbia.

Dr. Bakos was a winner of design contests from the Design Automation Conference in 2002 and 2004. He is a member of the ACM and Computer Society.

Panormitis E. Elenis (S'06) received the B.E. degree in electrical and computer engineering and the B.S.A.S. degree in information technology from Youngstown State University, Youngstown, OH, in 2006. Currently, he is a graduate student with the Department of Engineering and Computing, University of South Carolina, Columbia.