4-2007

FPGA Acceleration of Gene Rearrangement Analysis

Jason D. Bakos
University of South Carolina - Columbia, jbakos@cse.sc.edu

Publication Info

Published in *Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'07)*, 2007, pages 85-94.
http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4297231

FPGA Acceleration of Gene Rearrangement Analysis

Jason D. Bakos
University of South Carolina
jbakos@cse.sc.edu

Abstract

In this paper we present our work toward FPGA acceleration of phylogenetic reconstruction, a type of analysis that is commonly performed in the fields of systematic biology and comparative genomics. In our initial study, we have targeted a specific application that reconstructs maximum-parsimony (MP) phylogenies for gene-rearrangement data. Like other prevalent applications in computational biology, this application relies on a control-dependent, memory-intensive, and non-arithmetic combinatorial optimization algorithm. To achieve hardware acceleration, we developed an FPGA core design that implements the application's primary bottleneck computation. Because our core is lightweight, we are able to synthesize multiple cores on a single FPGA. By using several cores in parallel, we have achieved a 25X end-to-end application speedup using simulated input data.

1. Introduction

Phylogenetic analysis is the study of evolutionary relationships amongst a set of species. A *phylogeny* (or *phylogenetic tree*) is as 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 [1].

As shown in Figure 1, a phylogeny is an unrooted binary tree. Each of the n leaves has degree 1 and represents a species that currently exists, while each of the n - 2 internal vertices has degree 3 and represents a 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, find a phylogeny that most closely resembles the species' actual relative evolutionary history. Maximum parsimony (MP) phylogeny reconstruction is generally considered to be among the most accurate reconstruction techniques because it (1) incorporates an evolutionary model into the reconstruction procedure and (2) computes biological data for ancestral vertices. MP techniques operate by performing a bounded exhaustive search over the space of all possible phylogenetic trees to find the phylogeny that minimizes the number of evolutionary

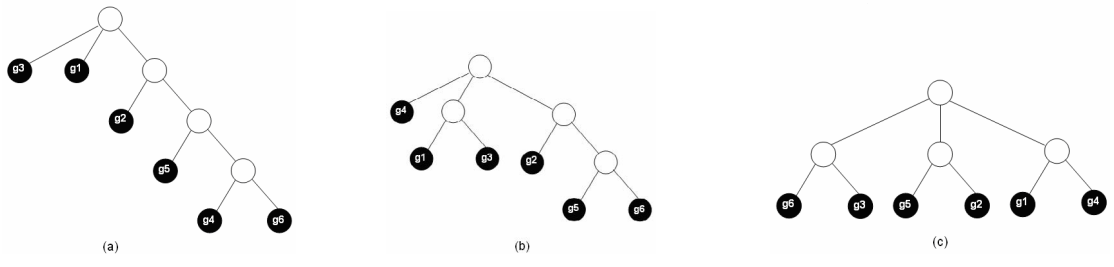

Figure 1. Three of the 105 possible phylogenies for 6 input genomes. Input species (g1, g2, …, g6) are shown in black while ancestral species are shown in white.

0-7695-2940-2/07 $25.00 © 2007 IEEE
DOI 10.1109/FCCM.2007.8

steps required to explain a given input set. In other words, the goal is to find the tree with the minimal *score*, where the score is the sum of all the tree's edge distances.

For n input species, there are $(2n\text{-}5) * (2n\text{-}7) * \ldots * 3$ possible trees. Since the tree space grows exponentially, the (optimal) MP technique is limited to relatively small input sets. When implemented as a branch-and-bound search, reconstruction of reasonably sized datasets are feasible. This search can also be effectively parallelized on a cluster computer, where each processing node searches disjoint regions of the tree space [2]. However, even in this mode, a single reconstruction procedure for 13-17 input species may require months of computation on a large-scale cluster (depending on the data set's genome size and evolutionary rate).

The goal of this work is to design FPGA cores that will allow a single accelerated processing node, with one or two FPGAs, to achieve equivalent performance to a medium- to large-scale cluster. The efficiency of this approach would allow biology labs to have greater access to cluster-class compute capacity for genome analysis. We also seek to demonstrate that applications that rely on combinatorial optimization can be accelerated with FPGA co-processor architectures, as opposed to control-independent arithmetic computations with which FPGA acceleration has traditionally been associated.

2. Gene-Rearrangement Data

In our current project, our goal is to accelerate MP reconstruction for *gene rearrangement* data, which refers to both the *type* of data used to represent each species' genome as well as an implied evolutionary model. When reconstructing phylogenies for this type of data, the edge distances for a given tree cannot be computed until after genome data for each of the internal vertices is computed. Thus, for each candidate tree that is evaluated during the tree search, (1) its internal vertices must be *labeled* with ancestral data, (2) its edge distances are computed, and (3) these distances are summed to determine the tree *score*.

Computing an edge distance can be performed with a fast (linear-time) algorithm, but labeling an internal vertex is extremely expensive (NP-HARD) [3]. For input sets with relatively high evolutionary rate (i.e. large diameter), the labeling computation constitutes the performance bottleneck for the overall reconstruction procedure. In our initial work, we have implemented an FPGA core that performs the labeling computation entirely in hardware. In addition, we developed a top-level architecture where multiple cores can be used in parallel to either (1) speed up the labeling computation for a single internal vertex, or (2) perform multiple labeling computations in parallel to label multiple internal vertices concurrently. When we replace the software version of this computation with our hardware-based one, we have achieved an overall application speedup of 25X for distantly related datasets.

In DNA sequence research, nucleotide sequences are known to undergo various edit events, including insertions, deletions, and substitutions. *Gene rearrangement* data (also known as *gene-order* data), on the other hand, is represented by a ordered sequence of *genes* (usually circular). Each gene itself represents a nucleotide sequence and thus exists in either a positive or negative orientation, where the sign denotes the internal ordering of the nucleotide sequence that the gene represents.

Genome A:	1	2	3	4	5	6	7	8
Genome B:	1	-5	-4	-3	-2	6	7	8
Genome C:	1	-5	-4	6	7	8	-3	-2
Index:	0	1	2	3	4	5	6	7

Figure 2. Genome B is produced from genome A by an inversion from genes 1 through 4. Genome C is produced from genome B by a transposition of genes 5 through 7 to index 3.

According to the Nadeau-Taylor model of evolution [3], gene-order data is subject to *gene rearrangement events*. These events include *inversions* (a gene subsequence is reversed in both order and polarity), *transpositions* (a gene subsequence is relocated within the ordering), and *inverted transpositions* (an inversion is followed by a transposition over the same gene subsequence). Examples of these events are shown in Figure 2. The relative rarity of genomic rearrangement events combined with the increased availability of complete genome sequences make gene-rearrangement data very attractive to biologists. As a result, many biologists have embraced this new type of data in their phylogenetic work [4,5,6,7] while computer scientists are slowly solving the difficult problems posed by analyzing manipulations of gene orders [8,9].

3. GRAPPA

Sankoff and Blanchette pioneered the maximum-parsimony methods in BPAnalysis [10], and Moret et al improved the approach of BPAnalysis in a package called GRAPPA [11]. Extensive tests on biological

and simulated datasets have shown that trees returned by GRAPPA are superior to those returned by other methods [11].

GRAPPA computes a lower bound for each possible tree based on the ordering of its leaves [11]. The search discards any tree, regardless of its topology, whose lower bound is greater than or equal to the best tree scored so far. For example, the lower bound of a 5-leaf tree having leaf ordering (g_5, g_2, g_1, g_4, g_3) is computed as ($d(g_5,g_2) + d(g_2,g_1) + d(g_1,g_4) + d(g_4,g_3) + d(g_3,g_5)$) / 2. In practice, GRAPPA typically prunes more than 99.9% of trees without scoring them.

GRAPPA *scores* each candidate tree not pruned by the lower bound computation. Ultimately the tree score is defined by the sum of its edge distances, but the first (and most expensive) step of the scoring procedure is to label each internal vertex with a *median genome*. The median genome is the optimal (but not necessarily unique) gene order that minimizes the sum of pair-wise distances between itself and the genome labels of each of the three neighbor vertices.

GRAPPA labels the tree's internal vertices using a two-step algorithm. In the first step, GRAPPA initializes the labels of the internal vertices. We refer to this as the *initialization phase*. GRAPPA offers several different initialization methods, but the most effective is to label each internal vertex by computing the median of the three nearest labeled vertices. Initially, only the labelings of the leaves are available. However, since the labels are applied as soon as they are computed, an increasing number of internal vertices are labeled (and available) as the algorithm progresses.

Once the initialization phase initially labels all internal vertices, GRAPPA proceeds with a *re-labeling phase*, which is an iterative refinement algorithm that continually re-computes each internal vertex label using its immediate neighbors. If the new label improves the median score, the new label is applied immediately. The re-labeling is terminated after the first iteration where no labels are updated. Note that this algorithm is guaranteed to converge but only guarantees a locally optimal solution. Once the tree converges, the edge distances are computed with a linear-time distance function and summed to yield the tree score.

4. Median Computation Performance

Labeling an internal vertex requires computing a median of three gene orders. Our performance characterization of GRAPPA has shown that the time required to perform a median computation is an exponential function of the sum of pairwise distances between the three input gene orders and their optimal median. This can also be expressed as the *diameter* of the inputs. This was no surprise, since the median algorithm is NP-HARD. This is shown graphically in Figure 3.

The effect of this is that the portion of GRAPPA's total execution time that is spent labeling sharply increases with the evolutionary rate the inputs. This is also caused by lower pruning rates (thus higher scoring rates), since the lower bound is less effective for more distantly related input sets. In practice, even moderately distant input sets will require over 99.9% of GRAPPA's total execution time computing medians. This is shown graphically in Figure 4.

Note that the median computations performed in the initialization phase typically consume several orders of magnitude more time than the medians computed in the re-labeling phase, since the diameter of these median computations are significantly higher in the early stages of the refinement procedure.

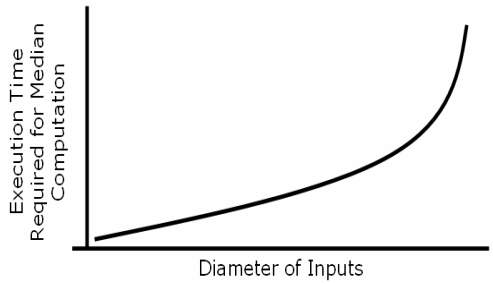

Figure 3. A median computation becomes more time consuming as the diameter of its inputs increase.

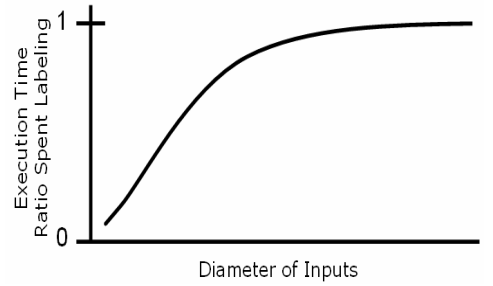

Figure 4. The relative amount of total execution time that GRAPPA spends labeling internal vertices (performing median computations) increases asymptotically to 100% with the diameter of the input data set. Thus performing median computations is the performance bottleneck for relatively "difficult" input data.

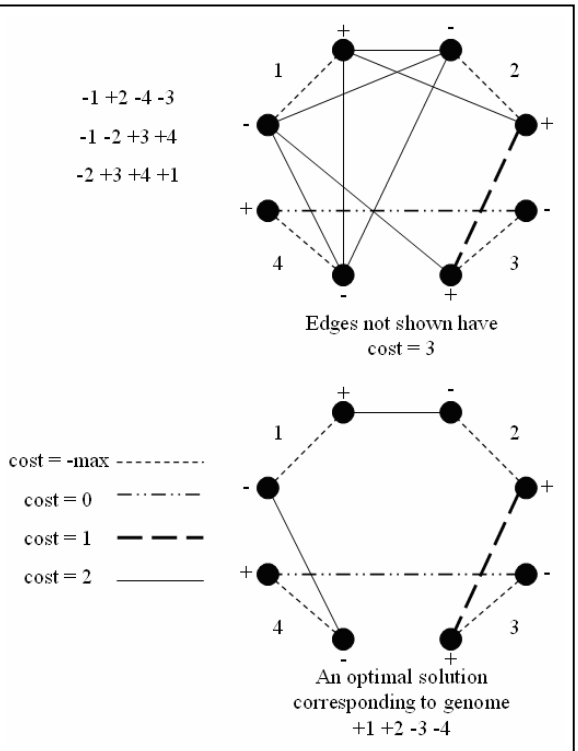

Figure 5. Breakpoint median TSP formulation.

5. Breakpoint Median Algorithm

The *breakpoint distance* is an estimate of the number of rearrangement events that separate two genomes. The breakpoint distance between 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*. For example, (circular) genomes *A*=(1 -2 -3 4) and *B*=(4 2 -1 -3) have a breakpoint distance of 2, because gene pairs (-2 -3) and (4 1) appear in *A* but neither {(-2 -3) or (3 2)} nor {(4 1) or (-1 -4)} appear in *B*.

5.1. Breakpoint Median

Given three genomes *A*, *B*, and *C*, the breakpoint *median* is a fourth genome *M* such that the breakpoint median score, *score = d(A,M) + d(B,M) + d(C,M)*, is optimally minimal where *d(x,y)* is the breakpoint distance between genomes *x* and *y*.

As shown in Figure 5, computing a breakpoint median for three genomes requires solving a traveling salesman problem (TSP) formulated in the following way [10]. Given genomes A, B, and C, each consisting of an ordering of *n* signed genes, construct a fully-connected undirected graph having vertices = {-g_n, …, -g_1, g_1, …, g_n} and define *w(g,h)* to be the weight between vertices *g* and *h*.

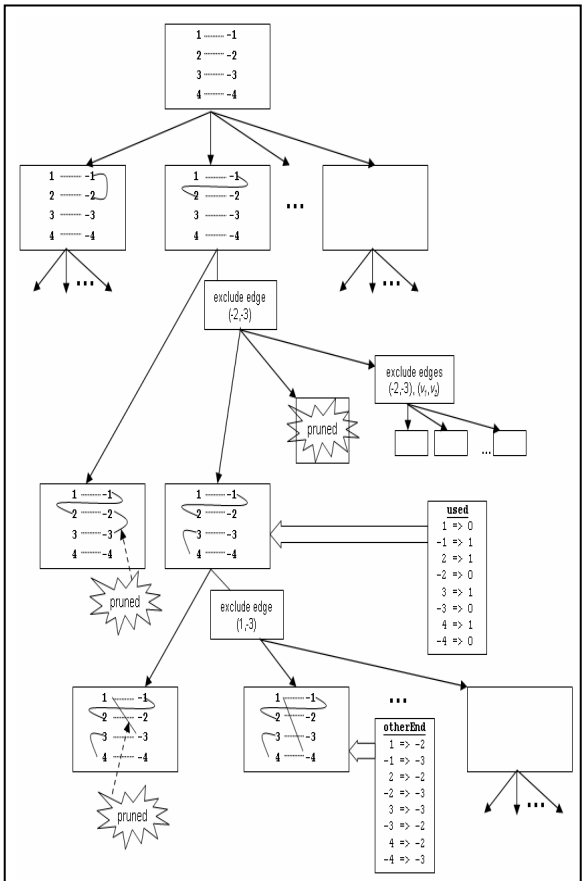

Figure 6. Graphical representation of a breakpoint median TSP depth-first search tree and associated data structures. Pruned edges are excluded from the lower bound computation from the level they are pruned to the bottom of the tree, the "otherEnd" array stores TSP path end-points to prevent cycles that do not include all vertices, and the "used" array keeps track of which vertices in the current solution state have degree 2.

For each gene *g*, *w(g,-g)* = -∞, guaranteeing 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, …, 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 the breakpoint median score is optimally minimal. Note that the TSP tour cost of the solution, excluding the -∞ edges between each vertex-pair representing the positive and negative version of each gene in the tour, is equivalent to the breakpoint median score of the solution.

The number of cities in the TSP graph is $2n$, where n is the number of genes. Since n is typically less than 1000, optimally solving the TSP is feasible. Finding an optimal solution is important since heuristic methods to compute medians will have a detrimental effect on the accuracy of the reconstruction procedure.

5.2. Algorithm Implementation

As shown in Figure 6, the breakpoint median algorithm bundled with GRAPPA performs a depth-first branch-and-bound search over the space of all possible paths through the graph implied by the three input genomes.

The first step of the algorithm is to establish an initial "best found so far" TSP tour cost to use as the initial upper bound. Recall that the TSP tour cost and the median score of the corresponding solution are equivalent values. During the **re-labeling phase**, this initial upper bound is the median score of the previously computed median label. During the **initialization phase**, there is no previous median label so the initial upper bound is determined by finding which of the three input genomes has a minimum sum of distances to the other two, and uses this sum as the initial upper bound, i.e. *upper bound* = min *(d(A,B)+d(A,C)),(d(A,B)+d(B,C)),(d(A,C)+d(B,C))}*.

If the search exhausts the search space without finding a better result, it returns this input genome as the result. The second step is to read the input genomes and construct the resultant TSP graph. By definition, each edge in the graph has weight - ∞, 0, 1, 2, or 3. The algorithm organizes the weight 0, 1, and 2 edges into a list sorted by edge weight.

It then creates an empty edge set to serve as the current search state, which we refer to as the **partial solution**. All the weight -∞ edges are assumed to be in the current partial solution. Thus, every vertex in the partial solution has a degree of one. Note that the weights of these edges are not included in the tour cost.

The algorithm iterates through the sorted edge list in order (beginning with the first edge) and adds each edge to the partial solution that obeys two conditions: (1) the edge must not cause any of the vertices in the tour implied by the current partial solution to have a degree of greater than two since the TSP tour must not contain branches (in our implementation, this is implemented with the *used* memory), and (2) the edge must not create any cycle in the current partial solution unless the addition of this edge results in a full tour (in our implementation, this is implemented with the *otherEnd* memory).

If no edges from the current point forward in the sorted edge list satisfy these two conditions, the algorithm will record the current tour as the best found solution if its cost (including any weight-3 edges that must also be included to complete the tour) is less then the current upper bound.

The first condition for adding an edge is implemented by keeping track of the degree of each vertex in the partial solution. Condition 2 is implemented with a memory that keeps track of the end-points of all path fragments in the partial solution. This allows a quick way to avoid adding certain edges to the partial solution or including these edges into the lower bound computation if their inclusion would result in a cycle (unless the cycle includes all vertices).

When the search begins, this memory is initialized to indicate that the vertices representing the positive and negative version of each gene form a two-city partial tour fragment, i.e. $OtherEnd(a) = -a$. As the search progresses, these partial fragments grow in size. Each time an edge with vertices a and b is added to the current partial solution, the following assignments are made:

$OtherEnd(OtherEnd(a)) = OtherEnd(b)$ and
$OtherEnd(OtherEnd(b)) = OtherEnd(a)$.

In order to prevent loops in the partial tour, a candidate edge (a,b) will not be added to the partial solution (or lower bound computation) if $OtherEnd(a) = b$, unless adding the edge will complete a full tour.

5.3. Lower Bound Computation

Each time the search adds an edge, it computes a lower bound for the partial solution. If the lower bound is greater than or equal to the upper bound, it prunes the last added edge, re-computes the lower bound, and either adds a new edge or prunes again.

The search computes the lower bound using the following technique [6]. Initialize the lower bound to zero. For each vertex that currently has a degree of one in the current partial solution, add the weight of the lowest weight edge that leads to another vertex of degree one in the partial solution. This technique adds twice as many edges as required, so divide the final sum by two.

The lower bound computation disregards any edges that were previously pruned at or above the current level in the search tree. It also disregards any edges that would result in a tour cycle if that edge were added to the partial solution unless adding the edge would complete the tour. Each time the search prunes an edge, the search re-computes the lower bound because the exclusion of the pruned edge constitutes

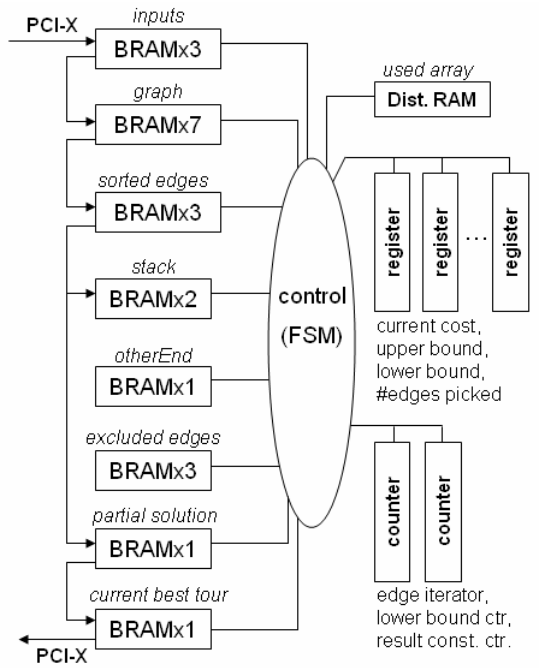

Figure 7. Simplified block diagram for the breakpoint median core. The core design is a large sequential logic circuit that establishes datapaths among several memory elements in each clock cycle. Static interconnects between memory elements are shown, while multiplexed interconnects among memory elements are established through the control unit (integrated within its output logic).

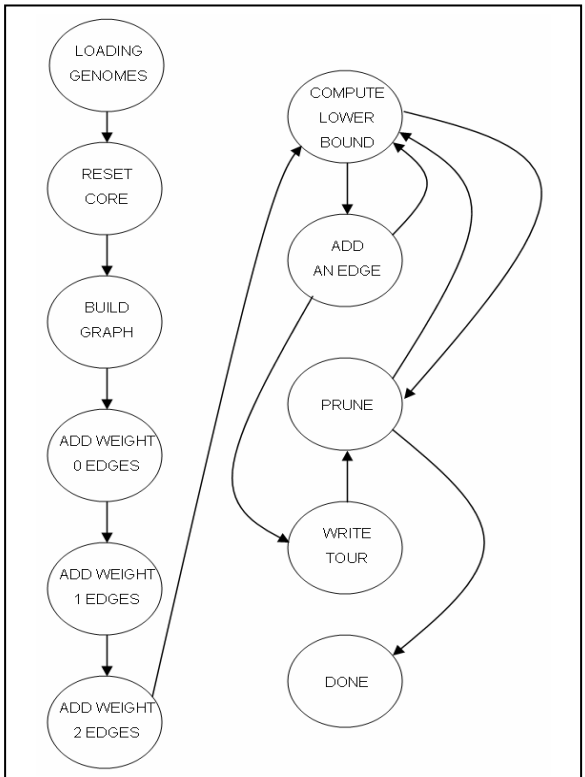

Figure 8. Finite state machine representation of median core controller.

information that was not available before the search added the edge originally.

6. Core Design

We designed our breakpoint median core in custom-written VHDL. It. implements the same basic breakpoint median algorithm as the one bundled with GRAPPA with a few notable differences. GRAPPA's breakpoint median core relies on recursion such that its depth-first search may be realized using the program activation stack. In order to achieve similar run-time behavior, we have implemented a stack memory using an on-chip block RAM (BRAM).

The median core uses this stack to keep track of information required to restore the state of the search when a branch of the search tree is pruned. For each edge that is added to the partial solution, the previous values of the otherEnd memory and the edge's index into the sorted edge list are pushed on the stack. Each time an edge is pruned (due to the lower bound or tour completion), the pruned edge's index in the sorted edge list is pushed on the stack so that edge can be re-included in the lower bound computation when the state of the search that caused the prune is changed.

Before the median core begins operation, the host system performs two tasks in software. First, it computes the initial upper bound. Second, it loads the input genomes and the initial upper bound into a specific set of on-chip memory locations that correspond to the median core to which it wishes to dispatch the median computation. The host performs these data transfers using a programmed I/O transaction across the PCI-X bus.

The core requires a one-time overhead of *2n* cycles to reset the search state memory of the core, *4n* cycles for each input genome to construct the TSP graph, and *10n* clock cycles to construct the sorted list of edges (where *n* is the number of genes).

After the initialization phase, the core proceeds by adding an edge (7 cycles), computing the lower bound (requiring *2n* clock cycles), then either pruning (requiring 3 to 10 cycles) and performing a re-computation of the lower bound, or simply adding another edge. Any time the core reaches the end of the

sorted edges list (and thus the bottom of the search tree), if the cost of the current tour is less then the current best, it constructs the result tour and saves it (requiring *n* clock cycles).

The operation of the core is clearly dominated by the lower bound computation. Even for less expensive median computations, the core will spend nearly all of its execution time in this state.

Figure 7 shows a simplified view of the median core microarchitecture. As shown, the median core design consists of a single block of control logic that is interconnected to a set of on-chip block RAMs (BRAMs), counters, and registers. The controller is designed as a finite state machine with integrated multiplexers that establish datapaths between the set of BRAMs and registers. The state diagram for the controller is shown in Figure 8.

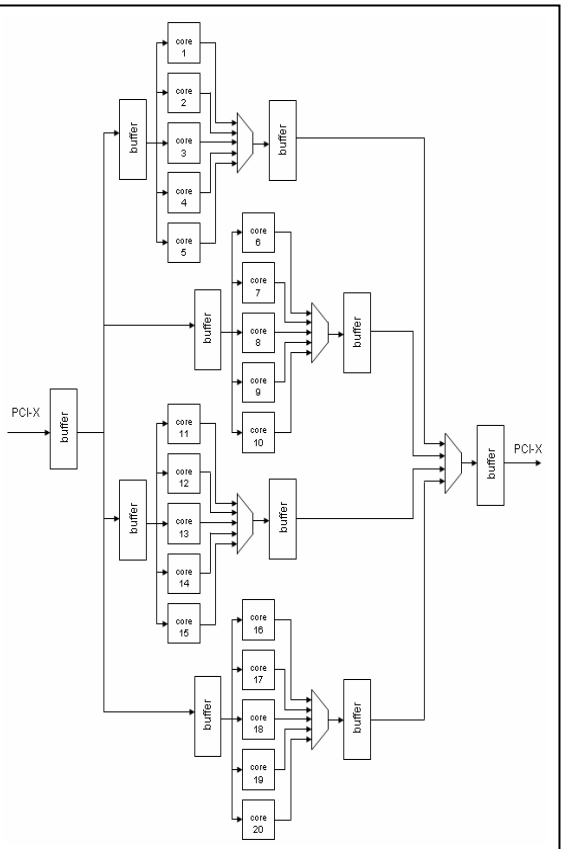

Figure 9. Simplified block diagram for the top-level FPGA implementation of a 20-core design. Due to the high clock speed requirement for the PCI-X interface (an independent clock domain from the slower core clock), the fan-outs and fan-ins are pipelined with degree 4 and 5. Dual-ported split clock BRAMs are used for the input and output data to cross clock domains.

The median core is capable of computing breakpoint medians of any reasonable size using only on-chip memory, although larger genome sizes (> 1000 genes/genome) will increase the resource utilization of each single median core beyond the numbers described below. The median core requires 21 of the 444 BRAMs available on the FPGA (< 5%). A single median core's logic requirements are 944 of the 44,096 logic slices (2%), including the logic overhead required for the PCI-X interface. These resource requirements indicate that the required number of independent on-chip memories is the limiting scalability factor for this design. Even with this limitation, we have successfully implemented 20 independently accessible median cores on our FPGA (although it is possible to fit 21).

Our breakpoint median core implementation is limited to a 56 MHz clock speed (two full orders of magnitude less than the microprocessor to which we are comparing against). Our clock speed is currently limited by the high latency data path required by the lower bound computation.

The PCI-X interface must meet a 133 MHz clock speed requirement. The BRAMs used to store the inputs and outputs are dual-ported/dual-clocked in order for the data to cross clock domains. In order to meet the timing requirements for the PCI-X interface, the PCI fan-out/fan-in to/from the cores are pipelined as shown in Figure 9.

7. Exploiting FPGA Resources

There are several techniques for using the parallelism of multiple median cores to speed up a single median computation. We developed one technique that is intended for the tree initialization phase. During initialization, each median core must rely on its three input genomes to compute an initial upper bound, since no previous label will exist for internal vertices until the re-labeling phase begins.

Under normal circumstances, the median core is initialized with an initial upper bound determined by the minimal median score corresponding to three input genomes. For input genomes *A*, *B*, and *C*, *A*'s median score is *d(A,B) + d(A,C)*, *B*'s median score is *d(B,A) + d(B,C)*, and *C*'s median score is *d(C,A) + d(C,B)*.

If the median core does not find a median solution with a lower score (which would guarantee that none exists), the software driver for the median core returns the corresponding input genome as the optimal result. If the core does find a better solution, the score of this result is guaranteed to be less than the initial upper bound.

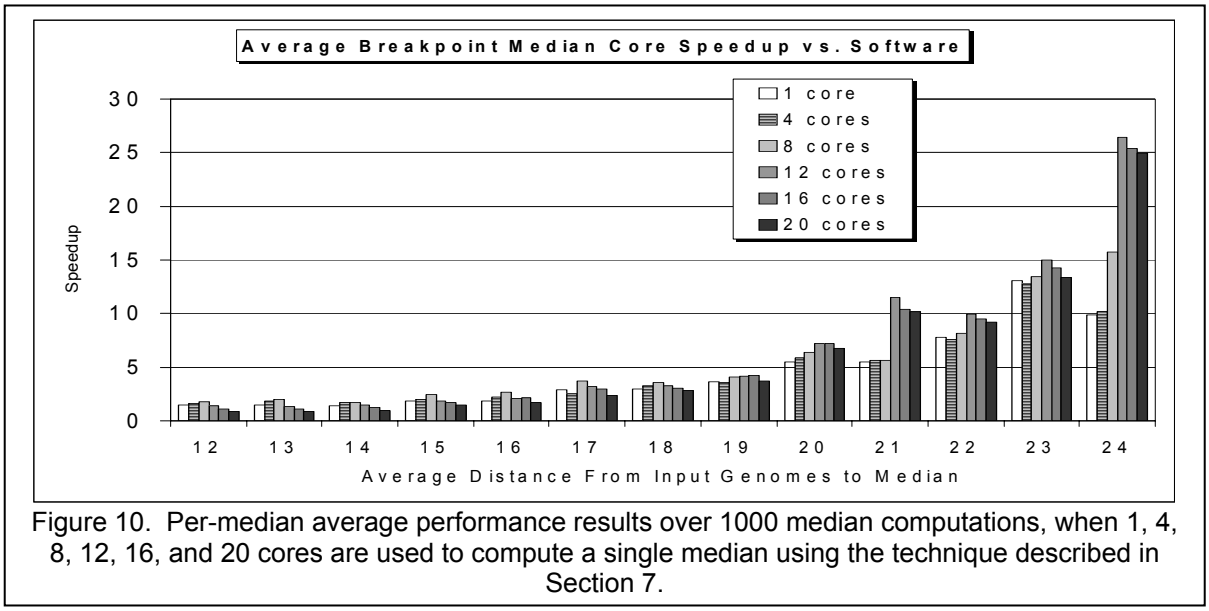

Figure 10. Per-median average performance results over 1000 median computations, when 1, 4, 8, 12, 16, and 20 cores are used to compute a single median using the technique described in Section 7.

In our multi-core strategy, if the initial upper bound inferred from the input genomes is s, initialize n cores with initial upper bounds $s - 1$, $s - 2$, …, $s - (n - 1)$. If the optimal median has a score less than s, the core with the lowest initial upper bound greater than the optimal score will converge on the optimal solution fastest. Therefore, after the first core completes its search having found a solution with a score lower than its assigned initial upper bound, the host will stop the other cores.

8. Characterizing the Breakpoint Median Core

Our test system consists of a Dell Precision 650 server containing a 3.06 GHz Intel Pentium Xeon processor. This system is used both to execute the GRAPPA software implementation and to host the FPGA accelerator.

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. Input genomes and the initial upper bound for any core are transmitted to the FPGA across the PCI-X interconnect using a programmed I/O write. After this, the host uses a programmed I/O read to poll the state of any median core on the FPGA. This allows it to determine when any individual core has completed computation. When this occurs, the host performs another programmed I/O read to read the result genome from the core.

Our software breakpoint median performance results were gathered from execution using the microprocessor, and our hardware breakpoint median results were gathered from execution on the set of breakpoint median cores available on the FPGA.

For each test, we generated 1000 three-leaf phylogenies and extracted the leaves to use as the median inputs. Each edge within each of these phylogenies has a distance chosen from a uniform random distribution having a range *distance* +/- 2, where *distance* is a parameter. We performed these tests for a genome size of 100 genes.

For each set of genomes, we invoke GRAPPA's breakpoint median routine **bbtsp** and record its execution time. We then use the same three genomes to invoke the hardware breakpoint median computation and record its execution time. Note the hardware execution time includes the CPU-to-FPGA communication time as well as the time to compute the initial upper bound (which occurs in software).

Speedup is measured in the traditional way, i.e. $time_{sw}$ / $time_{hw}$. A speedup of 1 would indicate equivalent performance between the software median computation and hardware median computation. Our results list the arithmetic mean of the individual speedups relative to software for the set of 1000 individual median computations for each input distance:

$$speedup = \frac{\sum_{i=1}^{1000} \frac{time_{sw}(i)}{time_{hw}(i)}}{1000}$$, where $time_{hw}(i)$ represents the hardware execution time of input data i.

Figure 10 shows our performance results when each breakpoint median is executed over 1, 4, 8, 12, 16, and 20 cores. The results show a clear trend where higher-diameter inputs achieve higher acceleration. This is

primarily because the overheads required to dispatch computations to the FPGA (i.e. the host to send the inputs to the core, for the core to initialize itself, for the host to poll the core's state, and for the host to read the result from the core) have greater relative effect for easy-to-compute input sets. However, the lowest speedup result was still greater than one for a single median core (verified down to *distance*=8). We stopped recording results at *distance*=24 due to very high run times (> 30 minutes average computation per median software computation).

The results also show that more distant input sets are able to take greater advantage of multi-core parallelism than less distant input sets. In fact, less distant input sets actually perform worse with more cores as compared to fewer cores due to the additional communication overheads associated with transferring the inputs into multiple cores. However, there is a point of diminishing returns at 12 cores, as the 16 and 20 core approach consistently performs worse than the 12-core approach for the inputs tested. Our best result is 26.4X for distance 24 over 12 cores.

9. Accelerated-GRAPPA

We made several modifications to the GRAPPA code to accelerate the tree scoring procedure by forcing it to dispatch all median computations to the median cores on the FPGA.

During the initialization phase, GRAPPA derives great performance benefit from computing each initial vertex's label serially using previously computed labels as potential inputs, as opposed to computing each initial label in parallel using only the nearest leaf labels (this technique reduces the average diameter of the three genomes involved in each individual median computation and improves the distances of the trees that enter into the labeling phase). Accelerated GRAPPA uses this same technique, but computes each initial label by dispatching each median computation in parallel to twelve median cores. Because the initialization phase dominates the time GRAPPA spends for tree scoring, this technique contributes significantly to overall application speedup.

In the re-labeling phase, GRAPPA labels each internal vertex serially and applies the label immediately if the new label improves the corresponding vertex's score. Each label that is applied may be used in subsequent median computations within the same iteration. In this phase, Accelerated-GRAPPA dispatches each median computation to a single core, but performs these median computations in parallel for all of the tree's internal vertices for each iteration. Although there is a sufficient number of available cores to dispatch each median to multiple cores (i.e. eight leaves have six internal vertices, requiring 18 of the 20 cores for three-core median computations), the low diameter of the median computations in this phase would make the communication overhead of this approach negate the benefits.

After the slowest median computation has completed for a given iteration, only the median results that improve the score of the corresponding internal vertex label are applied for the next iteration. If no median computation improves the score of the corresponding label, the re-labeling phase is terminated.

A consequence of parallel re-labeling is that only the labels from the previous iteration (or from the initialization phase, in the case of the first iteration) are available for any given iteration. This changes the convergence behavior from the software re-labeling phase, but still yields significant improvement in most experimental runs. Unfortunately, accelerating the re-labeling phase contributes only minor overall application speedup due to the relatively low overall time spent during this component of the scoring procedure.

Figure 11 shows our average speedups for entire GRAPPA runs over 10 unique 8-leaf, 100-gene synthetic datasets. The input sets were produced by

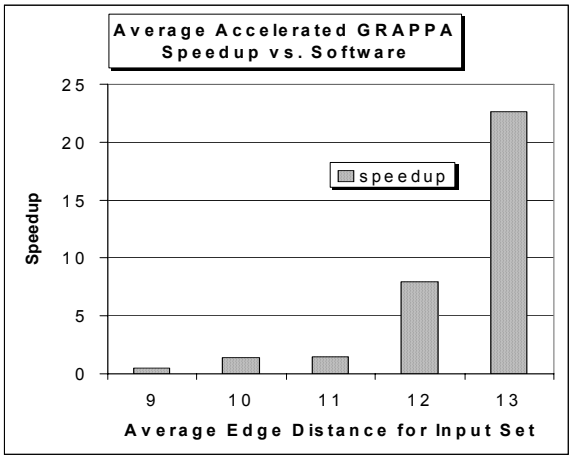

Figure 11. Average performance results over 10 phylogeny reconstructions of 8 input genomes. These results indicate the total end-to-end application speedup for accelerating the median computation within GRAPPA. The X-axis specifies the average edge distance of the randomly generated phylogenies whose leaves are used as inputs.

synthesizing phylogenies using a specified average edge distance. The leaves are extracted for use as inputs. The speedup for each experimental run was computed as $time_{SW}$ / $time_{HW}$). The results shown are the arithmetic mean of the individual speedups relative to software over each set of 10 GRAPPA runs for each input distance, as described in Section 8.

As with the breakpoint median performance results, the results show a clear trend where the average speedup increases with the evolution rate of the input set. These results are very sensitive to the average diameter of the input set. There are two reasons for this. First, higher evolutionary rate input sets force GRAPPA to spend higher portions of its execution time computing medians. In other words, more difficult data sets force the median computation to become more significant a bottleneck. Thus, accelerating the median computation has a higher impact on overall application speedup. Second, the median computations themselves are more greatly accelerated as the diameter of the median inputs increase. Speedup results span from one to 23 as the average input diameter increases.

10. Conclusions and Future Work

Our current results demonstrate that Accelerated-GRAPPA is capable of achieving a 23X speedup for input sets that have a relatively high evolution rate. Our most significant problem is that we can only effectively utilize the parallelism from 57% of the FPGA resources (12 out of a maximum of 21 cores). Even when using these resources, the performance improvement does not scale efficiently with increasing numbers of cores. Our future work is focused on more efficient exploitation of hardware resources.

We are currently developing an enhancements to our median core design to allow for extraction of finer-grain parallelism and allow for more efficient use of median cores. In one approach, multiple median cores will search disjoint regions of a single TSP search space and broadcast updates to a global upper bound. In this approach, all cores must exhaust their search space and the best result found across all the cores is guaranteed to be optimal.

In another approach, each core will search over the entire search space but choose equal-weight edges in different orders, since following different search paths will allow some cores to find lower upper bounds faster than others. A globally maintained upper bound is also used in this approach (using an upper bound broadcast). As in the previous approach, at least one core is guaranteed to find the optimal solution.

In addition, we are developing a tree generation and bounding core that will explore the phylogeny search space. Our current design requires only two BRAMs, indicating that it is possible to implement 222 parallel tree generation cores on a single Virtex-2 Pro 100. Since this is exactly how GRAPPA runs in cluster mode, we refer to this approach as "cluster-on-a-chip". Our ultimate goal is 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.

11. References

[1] B.M.E. Moret, J. Tang, L.-S. Wang, T. Warnow, "Steps toward accurate reconstructions of phylogenies from gene-order data," Journal of Computer and System Sciences, 2002, vol. 65; part 3, pages 508-525.

[2] D.A. Bader, B.M.E. Moret, "GRAPPA runs in record time," HPC Wire, 9(47), 2000.

[3] B.M.E. Moret, J. Tang, T. Warnow, "Reconstructing phylogenies from gene-content and gene-order data," Mathematics of Evolution and Phylogeny, O. Gascuel, ed., Oxford Univ. Press, 2005, 321-352.

[4] Raven, J.A., J.F. Allen (2003), "Genomics and chloroplast evolution: what did cyanobacteria do for plants?" Genome Biol 2003, 4, 209.

[5] Olmstead, R. and J. Palmer (1994), "Chloroplast DNA systematics: a review of methods and data analysis," Amer. J. Bot. 81, 1205–1224.

[6] G. Bourque and P. Pevznerr, "Genome-scale evolution: Reconstructing gene orders in the ancestral species," Genome Research 12, 26—36 2002.

[7] T. Garland, P. E. Midford, A. R. Ives, "An Introduction to Phylogenetically Based Statistical Methods, with a New Method for Confidence Intervals on Ancestral Values," American Zoologist 1999 39(2):374-388; doi:10.1093/icb/39.2.374.

[8] Felsenstein, J. (1978).,"The number of evolutionary trees," Systematic Zoology 27, 27–33.

[9] W. H. E. Day, D. Sankoff, "Computational Complexity of Inferring Phylogenies by Compatibility," Syst. Zool., 35(2):224-229, 1986.

[10] M. Blanchette, G. Bourque, D. Sankoff, "Breakpoint phylogenies," S. Miyano and T. Takagi, editors, Genome Informatics 1997, pages 25-34, Univ. Academy Press, Tokyo, 1997.

[11] B.M.E. Moret, S. Wyman, D.A. Bader, T., Warnow, and M. Yan, "A new implementation and detailed study of breakpoint analysis.", Proc. 6th Pacific Symp. on Biocomputing (PSB 2001), World Scientific Pub., 583-594.