Abstract—We present parallelization and heuristic techniques to reduce the run-time of FPGA negotiated congestion routing. Two heuristic optimizations provide over 3\texttimes speed-up versus a sequential baseline. In our parallel approach, sets of design signals are assigned to different processor cores and routed concurrently. Communication between cores is through the MPI (message passing interface) communications protocol. We propose a geographic partitioning of signals into independent sets to help minimize the communication overhead. Our parallel implementation provides approximately 2.3\times speed-up using 4 cores and produces deterministic/repeatable results. When combined, the parallel and heuristic techniques provide over 7\times speed-up with 4 cores versus the router in the widely used VPR FPGA placement/routing framework, with no significant impact on circuit speed or wirelength.

I. INTRODUCTION

The run-time of field-programmable gate array (FPGA) CAD tools is a major concern for FPGA vendors and their customers. Two factors are at play in worsening run-times for the largest designs. First, high current density in modern processors has created a “power wall”, limiting the rate of increase of clock speeds in processors, and spawning the multicore era. Second, Moore’s Law charges onward – state-of-the-art chips contain two billion transistors and continue to double in size every two years. There is, consequently, a widening gap between the size of chips, and the ability of CAD tools to handle them.

The largest industrial FPGA designs take hours or days to compile from HDL-to-bitstream, with placement and routing being the most run-time intensive steps of the CAD flow. Long run-times reduce engineering productivity, raise cost, and are a strong impediment to the widespread adoption of FPGAs by software developers, who are accustomed to compilation times in seconds or minutes. Improving the value proposition of FPGAs and expanding their user-base are paramount aims for commercial FPGA vendors, and lower tool run-time is key to enabling progress on these fronts. A promising direction to address the run-time challenge is to accelerate CAD tools through parallel computing. Today’s FPGA CAD tools are frequently run on commodity processors with 4 cores, and 8 and 16-core commodity processors are not far down the road. Such processors offer significant potential for run-time reduction through parallelization. By and large, however, the underlying CAD algorithms in today’s FPGA tools are single-threaded, and do not take advantage of the available processing power. The importance of leveraging multi-core parallelism was underscored recently by Altera, who published techniques for multi-core parallel placement [1].

In a previous publication [2], we presented fine and coarse-grained techniques for parallel FPGA routing. Our coarse-grained approach aligns closely with what one would intuitively think of as parallel routing: design signals are partitioned into sets, each of which is routed by a different processor core. Intermittent communication between processor cores is used to communicate routing results, thereby giving each core a global picture of the intermediate routing state.

In this journal paper, we expand on our previous work by first improving the baseline sequential routing algorithm through two “engineering enhancements” that together deliver a considerable run-time reduction. We then apply our coarse-grained method to parallelize the enhanced sequential router. We also present improvements to our parallel method, including a hierarchical geographic partitioning method, which allows signals to be split between processors such that the need for inter-processor communication is reduced. We use MPI (message passing interface) as the communication protocol [3], [4]. Our approach is implemented within the VPR framework [5], and offers deterministic/repeatable results.

We experimentally demonstrate the run-time benefits of our enhancements to the baseline sequential router, as well as the proposed parallelization techniques. We further show that our techniques have no appreciable impact on quality-of-result: circuit speed or wirelength.

The remainder of this paper is organized as follows: Section II provides relevant background on FPGA routing algorithms, parallel routing, and introduces the parallel programming techniques used in this work, namely, MPI. Improvements to the sequential routing algorithm are presented in Section III. The proposed parallel routing techniques are described in Section IV. Section V outlines the methods we used to profile the parallel algorithm, and presents the associated results. Section VI explains improvements we made to our parallel approach, including the hierarchical geographic partitioning. An experimental study is presented in Section VII. Conclusions and suggestions for future work are offered in Section VIII.

II. BACKGROUND

A. FPGA Routing

The two largest FPGA vendors, Xilinx and Altera, use a variant of the PathFinder negotiated congestion routing...
algorithm [6] in their commercial routers [7], [8]. PathFinder is also used in the publicly-available VPR FPGA placement and routing framework [5], which we parallelize in this work. VPR is also part of the SPEC benchmark suite. Fig. 1 gives an overview of the negotiated congestion approach. First, all signals in a placed design are routed in the best manner possible (e.g. minimum delay or minimum resource usage), permitting shorts between the signals (meaning that two different signals may use the same wire on the FPGA). After initial routing, each signal is routed with its ideal routing solution, albeit infeasible, owing to the shorts. Then, the penalties associated with shorts are increased, and the signals are re-routed with consideration of the increased penalties. The process of increasing the penalties for shorts and re-routing the signals continues iteratively until all shorts are removed and the routing is feasible. One pass through the loop of Fig. 1 is referred to as a PathFinder iteration. In essence, signals negotiate amongst themselves for which signal gets to keep a popular/shared FPGA resource. Our parallelization approach accelerates the “route all signals” step in the negotiated congestion flow.

At the heart of negotiated congestion routing is maze expansion [9] – the algorithm used to route a load pin on a signal and the most computationally expensive aspect of FPGA routing. The routing resources in the FPGA are represented as a graph, \( G(V, E) \), called the routing resource graph, where each vertex, \( v \in V \), represents a routing conductor, i.e. a metal wire segment or a pin on a block. Each edge, \( e \in E \), represents a programmable routing switch in the FPGA that may be turned on to electrically connect a pair of conductors. To route a load pin on a signal, maze expansion begins by adding the vertex, \( s \), corresponding to the signal’s source pin, to a priority queue. The algorithm then enters a while loop that executes as follows: the lowest cost node, \( u \), is removed from the priority queue. If \( u \) is the target load pin, then a path from source to target has been found and the expansion terminates (exit the while loop). Otherwise, the nodes adjacent to \( u \) in \( G \) are identified and added to the priority queue. The process of removing a node from the priority queue and expanding it to visit its neighbors continues until the target pin is reached (drawn from the priority queue). The costs assigned to nodes in the priority queue can be based on any number of criteria, e.g. delay, capacitance, distance to the target pin, and the number of signals contending to use a node. Node costing itself can be compute-intensive, especially if sophisticated RC delay modeling is used.

B. Parallel FPGA Routing

Chan and Schlag were the first to parallelize negotiated congestion FPGA routing [10], showing impressive speed-up results of \( 2.5 \times \) using 3 processors. Their work bears some similarity to our approach, with three key differences: First, they target a distributed computing framework of networked workstations, and not a modern multi-core processor. Second, we take a more sophisticated approach to load balancing among processors. Third, and most important, their results are not deterministic – different routing results are produced each time the router is executed, making the approach impractical in an industrial context, where vendors and users expect identical results to be produced each time the tools are executed. Repeatability is especially crucial in early design development and debugging. The non-parallel router in VPR is completely deterministic because nodes are added to a priority queue in the same order on each run, and it is important to maintain this property.

Another work by Cabral et al. described parallel FPGA routing specifically for a “planar” routing architecture [11]. In a planar architecture, wires in the \( i \)’th routing track within a channel may only be programmably connected to other wires that are also on the \( i \)’th routing track. Consequently, routing tracks can be viewed as “planes” that do not intersect with one another. Planarity simplifies the parallelization problem, as processor cores can operate independently on each plane, reducing the need for inter-processor data sharing. While early FPGAs (such as the Xilinx XC4000 [12]) used planar routing, it has since been shown to negatively impact routability. Modern FPGA interconnect is not planar [13], [14], making Cabral’s work inapplicable today.

There is considerable prior work on parallelizing the single source shortest path problem [15], [16], [17] which is related to maze routing, however, these parallelization techniques are not intended to be used in a highly directed (A*) graph search, which is typically used in FPGA routing. In [18], it is observed that in practice, the run-time difference between a directed and breadth first search for FPGA routing is \( \sim 53 \times \), which far exceeds the speed-ups observed with parallel shortest path algorithms. In this paper, our speed-ups are in addition to the algorithmic speed-ups achieved in [18].

C. Parallel Programming Environment

We use MPI as our parallel programming environment. MPI is a widely used communications protocol that enables separate processes to communicate with one another. The processes may be running on separate cores within a multi-core processor, or may even run on entirely separate processors across a network. Unlike threads, concurrently running processes do not share memory. Communication in MPI, therefore, is handled explicitly through messages between processes, rather than through access to shared variables in memory. In particular, processes send and receive messages with one another, and such sends and receives can be either non-blocking or blocking. Blocking messages are used for process
synchronization. For example, if a process $A$ issues a blocking send to a process $B$, process $A$ must wait for a corresponding receive from process $B$ before $A$ continues execution. When non-blocking messages are used, the initiating process does not wait for the destination process to respond — execution continues immediately in the initiating process. We found MPI to be convenient for implementing our parallel routing approach. Using MPI’s distributed memory programming model, we were able to parallelize VPR without having to make its data structures thread safe, which reduced the amount of effort necessary to create a functional parallel implementation. One of the drawbacks of using a distributed memory model is that the overall memory usage is proportional to the number of cores that are running VPR in parallel. Section VIII describes future work that could address this issue.

III. ENGINEERING ENHANCEMENTS TO 
PATHFINDER ROUTING

In this section, we describe two enhancements we made to the baseline VPR PathFinder implementation that we believe bring it more in line with modern industrial routers. In our prior work [2], we achieved significant speed-ups through parallelization of the baseline (unoptimized) VPR. In this work, we show that similar speed-ups can also be achieved on a more optimized PathFinder, lending further credibility to our parallelization techniques.

A. Propagating Logic Element Outputs

A key low-level building block in an FPGA is called a logic element (LE), which (at a minimum) consists of a $K$-input lookup table (LUT) and a register that can be optionally bypassed for implementing combinational logic. A $K$-LUT can implement any logic function of up to $K$ variables. Logic elements are grouped together into clusters called logic blocks, where a cluster contains $N$ LEs and local intra-cluster interconnect. The local interconnect permits fast connectivity between LEs that are packed into the same cluster. Packing of LEs into clusters happens prior to the placement and routing stages of the FPGA CAD flow. The placer is then concerned with placing the clusters; the router is concerned with routing the inter-cluster connections.

In the most commonly used FPGA architecture model [19], the inputs to a cluster can connect to any of the LE inputs within the cluster — there exists a full crossbar between the cluster inputs and the LE inputs. On the other hand, the output of each LE is connected to a single cluster output. Despite this output limitation, because LEs are identical and due to the input crossbar, the LEs can be swapped with one another to realize output pin flexibility. This means that the routing algorithm is free to use any cluster output for a signal driven by an LE output. In fact, the routing resource graph used in the baseline VPR has edges between an LE output and all outputs of its cluster.

In the baseline VPR PathFinder implementation, during the maze routing of a multi-fanout signal, it is legal for that signal’s source (LE output) to temporarily use several cluster outputs, as shown in Fig. 2. This can prove useful when the cluster output pins are physically located on different sides of the cluster adjacent to different routing channels, and a signal has sinks (load pins) in different directions. The cluster output used in the route from the source to the first sink may not be on the lowest cost path for subsequent sinks, so it makes sense to allow the temporary use of multiple cluster outputs. For a cluster that is not fully utilized, LEs can be duplicated so that using multiple cluster outputs is possible.

We found that the cluster output pin flexibility in VPR increases the congestion seen on the routing resource nodes corresponding to the cluster outputs, increasing the time required for PathFinder to converge. We modified this aspect of VPR to disallow routing resource node expansion around LE outputs (source) once the first sink of a multi-fanout signal has been routed. Subsequent sinks must use the same cluster output as the first sink, as shown in Fig. 3. This means that subsequent signals beginning at this cluster always have a free pin, thereby reducing contention.

B. Skipping the Re-Routing of Uncongested Signals

In the original PathFinder algorithm [6], as well as in its implementation in VPR, each signal is ripped-up and re-routed in every iteration, as depicted in Fig. 1. That is, even signals that do not currently use over-capacity routing resources are re-routed each iteration. We refer to such signals as uncongested signals. We experimentally observed that the re-routing of uncongested signals was not necessary for the PathFinder algorithm to converge to a short-free (feasible) state, and moreover, consumed significant run-time. We therefore modified the VPR router to skip the re-routing of uncongested signals — an easy-to-implement change that, despite increasing the required number of PathFinder iterations, provided a considerable overall run-time reduction. We also experimented with only skipping signals that remain uncongested for multiple consecutive PathFinder iterations. Although this reduced the number of necessary PathFinder iterations, it resulted in an overall increase in run-time.

C. Effectiveness of Engineering Enhancements

Table I shows the run-time, critical path delay, and total wirelength for three scenarios: baseline sequential VPR (columns labeled base), VPR enhanced with output propagation (columns labeled prop), VPR enhanced with both

\footnote{$K = 4$ and $N = 10$ are common parameter values.}
output propagation and the skipping of re-routing uncongested signals (columns labeled skip). The results correspond to an architecture with $K = 4$ and $N = 10$ and length-2 wire segments (the full experimental methodology is described in Section V below). The first column lists the names of benchmark circuits, followed by three groups of three columns showing run-time, critical path delay, and post-routed wirelength (# of wire segments used in the routing). The results show that propagating LE outputs to cluster outputs offers a significant benefit to run-time, $1.39 \times$ vs. baseline VPR, while having a negligible effect on both critical path delay and wirelength. Skipping the rip-up and re-route of uncongested signals significantly improves run-time by $2.38 \times$ on top of the output propagation enhancement, while on average incurring a slight penalty to both critical path delay (2%) and wirelength (2%). Not ripping up and re-routing uncongested signals does have the potential to harm quality of results as critical routes may be forced to choose less desirable paths in late PathFinder iterations, as evidenced by the critical path delay numbers for the benchmark circuit “spla” in Table I. For most benchmarks, however, the effect on quality of results is negligible. Together, both enhancements provide over $3 \times$ run-time speed-up over baseline VPR.

IV. MULTI-CORE PARALLEL ROUTING

Our parallelization approach is to partition the signals into sets, which are then assigned to separate instances of VPR, with each instance running as a separate process on a separate processor. Each VPR instance routes its own set of signals and maintains its own data structures, including a routing resource graph and associated congestion information. The different VPR instances use MPI messages to communicate intermediate routing results with one another and synchronize their respective views of the overall routing state. By using MPI, we avoided both having to alter most of the data structures in VPR, and the requirement that many VPR functions be made thread-safe. MPI provides a mechanism whereby all VPR instances (processes) can be invoked simultaneously.

A. Synchronization and Load Balancing

For PathFinder to converge to a short-free routing for all signals, each VPR instance must have a relatively accurate picture of the congestion contributed by all signals. Simply put, to route signals in a short-free manner, one must know which routing resources are used by other signals and must avoid using such already-used resources. In our router, after a VPR instance routes a signal, it issues a non-blocking send message to all other VPR instances, meaning that it does not need to wait for other VPR instances to receive the update before continuing with its execution. The message contains the new route for the signal, as well as load balancing information (described later). Once such an update is sent, it is held in a queue by MPI, available to be received by a destination VPR instance. The key to achieving deterministic results is the use of blocking receive calls in the destination VPR instances. Blocking receives are issued by each VPR instance at specific/fixed points during routing. A detailed explanation is given below, but the main idea is that each VPR instance routes one or more of its own signals, then receives updates about other signals (from other VPR instances) before it proceeds to route additional signals. The point at which a VPR instance receives the updates and the specific signals about which it receives updates is identical from run-to-run, making our router deterministic.

Ideally, when a VPR instance wants to receive an update about the routing state of other signals, that update will already have been sent by some other VPR instance, and will be available for “pick-up” in an MPI messaging queue. However, if the update has not already been sent, since the receive is blocking, the VPR instance will stall its execution until the update is available (i.e. until the message is sent and arrives). As with any parallel algorithm, it is desirable to minimize processor stall time. In our router, this goal translates into balancing the amount of work between VPR instances, and only issuing a blocking receive at points when it is likely that an update has already been sent.

To balance the amount of work among VPR instances, we must estimate the time needed to route a signal. We considered three different prediction metrics for a signal’s route time:

1) Number of loads. Fanout was also used as the prediction metric in [20].
2) Bounding box. We expect that signals with a large bounding box have a larger distance between pins, implying a longer routing time.
3) Number of routing resource graph nodes visited during maze routing in the previous PathFinder iteration. For each signal, we keep track of the total number of nodes visited during maze routing for the signal in the previous PathFinder iteration; we use this to predict the time needed for the signal in the current iteration.

At the beginning of each PathFinder iteration, we partition the signals into $N$ sets, where $N$ is the number of VPR instances (# of processor cores). Partitioning into signal sets is done such that the sum of the prediction metrics for the signals in each set is approximately equal. Each set is assigned to one VPR instance. Note that metrics #1 and #2 above do not change between PathFinder iterations, so the partitioning of signals is unchanged across PathFinder iterations when either of these metrics are used. Metric #3, on the other hand, is affected by routing congestion and consequently, when this metric is used, the signals may be partitioned into sets differently across successive PathFinder iterations. Note that we cannot use the wall clock time needed to route a signal in the previous iteration as a prediction metric, as wall clock time is non-deterministic.

Having partitioned the signals into sets, parallel routing commences – the VPR instances begin routing their respective signals. After a VPR instance routes a signal, it sends a non-blocking update to all of the other VPR instances. The update message contains the signal’s routing (as well as the number of routing resource nodes visited while maze routing

---

<sup>2</sup>When metric #3 is used, there is no history available in the first PathFinder iteration, so we either assign each process an equal number of signals in that iteration, or use the method described in Section VI-A.
the signal, if metric #3 above is being used). Since the send is non-blocking, the VPR instance may continue routing signals in its set, or it may decide to receive an update from other VPR instances. The decision on whether to receive an update is based on predicting whether the other VPR instances have indeed already sent an update, and, such a prediction is made using knowledge of which signals are being routed by the other instances, as well as the prediction metrics above.

Each VPR instance maintains work counters that estimate the amount of work performed by other VPR instances as they route their respective signal sets. A VPR instance uses its counters to predict whether updates have been sent by another VPR instance and whether to issue a blocking receive. We provide an example in Fig. 4. In this example, the number of loads on a signal is used to predict a signal’s routing time. A set of 5 signals is split between two processes so that VPR instance (process) A has 2 signals and VPR instance (process) B has 3 signals. The figure shows two arrays, indexed by variable i, containing the number of loads on the signals each process is responsible for. The figure shows that the first signal belonging to process A has 6 loads, and the second signal has 2 loads. Arrows in the figure correspond to updates sent by process B and received by process A.

We refer to the work counters for processes A and B as Work_A and Work_B, respectively. At the beginning of a PathFinder iteration, the work counters are initialized to the number of loads on the first signal for each VPR instance, making Work_A = 6 and Work_B = 3. Once the first signal is routed by process A, the new route is sent to process B (arrow not shown), though it is not necessarily immediately received. Then, process A compares Work_B with its own work counter, Work_A, to determine whether it is likely that process B has sent any updates. Since Work_B is smaller than Work_A, process A assumes process B has already sent an update for its first signal. A blocking receive is issued for B’s first signal. Once the update is received, Work_B is incremented by the number of loads on the next signal in process B, making Work_B = 3 + 2 = 5. Since 5 is also smaller than Work_A, another blocking receive is issued for the second signal of process B. Process A then updates the work counter associated with process B with process B’s third signal so that Work_B = 5 + 3 = 8. Since 8 is greater or equal to Work_A, it is unlikely that process B has sent an update for its third signal. No receive is issued and process A proceeds to route its second signal (with 2 loads). Once it is finished routing this signal, it sends a non-blocking route update, and updates Work_A with the number of loads on the signal that was just routed, making Work_A = 6 + 2 = 8. In this case, Work_B is not less than Work_A, but process A would still like to receive an update since it has no more signals to route.3

The pseudo-code for our parallel PathFinder implementation is shown in Fig. 5. Note that Fig. 5 only shows the portion of the router relevant to the parallelization. It does not, for example, show code responsible for updating the congestion costs. The code is written from the perspective of one of N processors participating in the routing. We use the variable j to refer to a processor index; local is the index of the local processor on which the algorithm in Fig. 5 executes. SigSet[j] represents the set of signals assigned to processor j for routing. The N-element array work holds work counters for each processor. The N-element array sig holds estimates of which signal is currently being routed by each processor. The predict(sig) function estimates the amount of work necessary to route a signal sig. The firstSig(SigSet[j]) function returns the first signal in processor j’s set of signals to route. Each successive call to the nextSig(SigSet[j])

---

### TABLE I

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Run-time(s)</th>
<th>Critical Path Delay(s)</th>
<th>Wirelength</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>base</td>
<td>prop.</td>
<td>skip</td>
</tr>
<tr>
<td>cf-cordic_v18.ip</td>
<td>7.96</td>
<td>5.68</td>
<td>2.66</td>
</tr>
<tr>
<td>cf-cordic_v18.vo</td>
<td>18.72</td>
<td>14.72</td>
<td>4.14</td>
</tr>
<tr>
<td>cf-cordic_v18.dcm</td>
<td>22.95</td>
<td>14.55</td>
<td>5.71</td>
</tr>
<tr>
<td>des_perf</td>
<td>8.99</td>
<td>6.89</td>
<td>2.23</td>
</tr>
<tr>
<td>ext1010</td>
<td>13.49</td>
<td>8.36</td>
<td>5.10</td>
</tr>
<tr>
<td>tms</td>
<td>10.21</td>
<td>5.82</td>
<td>4.53</td>
</tr>
<tr>
<td>macr</td>
<td>25.86</td>
<td>20.67</td>
<td>6.00</td>
</tr>
<tr>
<td>macr_noblocking</td>
<td>21.16</td>
<td>18.95</td>
<td>8.64</td>
</tr>
<tr>
<td>pdc</td>
<td>24.15</td>
<td>12.84</td>
<td>6.55</td>
</tr>
<tr>
<td>tmsdec</td>
<td>3.14</td>
<td>3.30</td>
<td>2.99</td>
</tr>
<tr>
<td>x3847</td>
<td>7.83</td>
<td>5.19</td>
<td>2.67</td>
</tr>
<tr>
<td>spla</td>
<td>11.99</td>
<td>7.56</td>
<td>4.07</td>
</tr>
<tr>
<td>sv_chip1</td>
<td>14.38</td>
<td>10.97</td>
<td>4.42</td>
</tr>
<tr>
<td>sv_chip2</td>
<td>14.34</td>
<td>12.05</td>
<td>40.30</td>
</tr>
<tr>
<td>sv_chip</td>
<td>15.23</td>
<td>15.83</td>
<td>13.45</td>
</tr>
<tr>
<td>geomean</td>
<td>22.01</td>
<td>15.78</td>
<td>6.64</td>
</tr>
<tr>
<td>vs. baseline</td>
<td>1.00x</td>
<td>1.39x</td>
<td>3.31x</td>
</tr>
<tr>
<td>vs. propagate</td>
<td>1.00x</td>
<td>2.38x</td>
<td>1</td>
</tr>
</tbody>
</table>

---

![Fig. 4. Blocking receives issued by processes during a PathFinder iteration using number of loads as signal run-time prediction metric.](image-url)

3We synchronize processes at the end of each PathFinder iteration to ensure that all VPR instances are working on the same iteration at the same time.
1: while shorts exist do
2: partition signals into N sets
3: for all j such that j ≠ local do
4: sig[j] = firstSig(SigSet[j])
5: work[j] = predict(sig[j])
6: end for
7: work[local] = 0
8: for all sigLocal ∈ SigSet[local] do
9: route sigLocal
10: send non-blocking update for sigLocal
11: work[local] += predict(sigLocal)
12: for all j such that j ≠ local do
13: while work[j] < work[local] do
14: receive blocking update from processor j for signal sig[j]
15: sig[j] = nextSig(SigSet[j])
16: work[j] += predict(sig[j])
17: end while
18: end for
19: end for
20: end while

Fig. 5. Pseudo-code for multi-core PathFinder with load balancing from the perspective of processor # local.

The function returns the next signal to be routed by processor j.

Lines 1 and 20 in Fig. 5 define a while loop which has so far been referred to as an iteration of PathFinder. Line 2 divides the signals into N partitions, where N is the number of processors. Each processor is aware of the set of signals belonging to every other processor. For each processor j aside from the local processor, lines 3 to 6 initialize the sig[j] variable to the first signal in SigSet[j], and then update the work[j] using the predict function. Line 7 initializes the local work counter to 0. Lines 8 and 19 define a while loop which executes once for each signal in the local processor’s set of signals. Line 9 routes a local signal called sigLocal. Line 10 sends a non-blocking update to all other processors containing the sigLocal’s updated route and some load-balancing information. On Line 11, the predict function is used to update the local work counter with the signal that was just routed (sigLocal). Lines 12 and 18 define a loop which executes once for each processor, j, aside from local. This loop is responsible for requesting and receiving any blocking updates from other processors that are predicted to have already been sent. Line 13 ensures that an update is only requested if it has likely already been sent. Line 14 receives a blocking update (i.e. line 15 will not execute until an update has been received). Lines 15 and 16 update the sig[j] and work[j] variables so that they reflect the next signal to route in SigSet[j]. Updates are requested from processor j until the work counters indicate that all sent updates have been received.

B. Effect of Work Estimate Metric on Stall Time

Figs. 6 and 7 illustrate the effectiveness of the different signal time prediction metrics for two benchmark circuits: clma and cf_fir_24_16_16. The vertical axis shows the average stall time over all VPR instances as a percentage of the total time needed to route all signals. Stall time was measured using the hardware counters internal to an Intel Core2Quad microprocessor. The results are given for 2, 3, and 4 VPR instances (processes). For each VPR instance, four bars are shown, corresponding to four different prediction metrics: number of loads, bounding box, number of nodes visited in maze routing, and wall clock time. Wall clock time is included for comparison only, as using this metric would lead to non-determinism.

The figures show that the best metric for predicting the runtime needed to route a signal is the number of routing resource graph nodes visited in maze routing the signal in the prior PathFinder iteration. This metric leads to the lowest amount of stall time (aside from wall clock time), and it is therefore the best proxy for run-time. We observed the same trends for other benchmark circuits and therefore, we use the number of nodes metric for load balancing in all of the results presented in Sections V-A and VII.

C. Assuring Convergence

Near the end of negotiated congestion routing, when most shorts between signals have been resolved, it becomes more important for VPR instances to have an up-to-date picture of the overall routing solution for all signals. Without this, PathFinder may not converge to a short-free state. With this in mind, we may decrease the number of active VPR instances towards the end of routing if we deem that PathFinder is not making adequate progress. When the number of shorts between signals falls below an empirically-determined threshold of $50 \times N$, we begin monitoring the rate of decrease in shorts between PathFinder iterations. If shorts decrease by less than 5%, we reduce the number of active VPR instances by one (of course, we always keep at least one VPR instance
EVENTUALLY, WE MAY BE LEFT WITH A SINGLE VPR INSTANCE – SEQUENTIAL ROUTING. THE MOTIVATION FOR THIS IS THAT AT THE END OF ROUTING, THERE IS LITTLE WORK LEFT TO BE DONE SO THERE IS LITTLE DOWNSIDE TO USING FEWER PROCESSES. BY REDUCING THE NUMBER OF ACTIVE PROCESSES, WE REDUCE THE POSSIBILITY OF CONVERGENCE PROBLEMS.

V. PRELIMINARY EXPERIMENTAL STUDY

In this section, we present and analyze results obtained using the parallel methods described in the previous section. We evaluate the parallelization approach against both the original VPR router as well as one which includes the engineering enhancements described in Section III.

Experiments were run on 2 systems running Linux (Debian 2.6.26-21), each with a Core 2 Quad Q9550 processor and 3 GB of memory. The Core 2 Quads have 4 cores, each running at 2.83GHz. Benchmark circuits were selected from the 20 largest MCNC benchmarks commonly used in FPGA CAD research, and also from the set of benchmarks that are packaged with VPR 5.0 [5]. Circuits were mapped into 4-input LUTs using ABC [21], then clustered using T-Vpack [22] into logic blocks with 10 4-LUTs and 22 inputs. Table II shows the benchmarks circuits used, along with the number of LUTs, latches, and signals in each. The FPGA routing architecture targeted contains bidirectional wire segments that span 2 logic block tiles. Across all runs, each circuit was routed using a fixed channel width of $1.3 \times$ the minimum channel width needed to route the circuit in the single-core case with a maximum of 100 PathFinder iterations. As it is more important to achieve run-time reductions on large circuits, only those circuits with single-core run-times of more than 10 seconds (when ripping up and re-routing congested signals) were included in our experiments; smaller circuits were excluded. The run-times presented correspond to the time spent routing signals in the PathFinder algorithm, which represents 84% of total router run-time for baseline VPR, on average. The remaining 16% of router time includes loading the benchmark circuit, building the routing resource graph model, performing delay analysis at the end of each PathFinder iteration, and a final post-routing timing analysis.

Table III shows our results with and without our engineering enhancements described in Section III. The first column of the table gives the name of each circuit. The next eight columns present the run-time needed to route each circuit with a given number of processes (processor cores). The 2x4 columns corresponds to using two Core 2 Quad processors across a network – included for comparison purposes only and not necessarily indicative of scalability. We measured an approximately 5x latency difference when sending a non-blocking message across a network (between processors) rather than between cores in a processor. This leads to an approximately 5% run-time penalty when some messages have to travel across a network. We reinforce that for a given number of processes, our router produces the same routing results from run-to-run – it is deterministic. The third last row of the table provides the geometric mean run-time across all circuits for a particular number of processes. The second last row gives the speed-up relative to the sequential (non-parallel) baseline VPR PathFinder implementation. The last row gives the speed-up relative to sequential VPR with our engineering enhancements.

Relative to baseline VPR, the left-half of Table III shows that we achieve $1.44 \times$, $2.30 \times$, and $2.85 \times$ speed-ups with 2, 4 and 2x4 cores, respectively. Relative to the enhanced sequential VPR, the right-half of the table shows speed-ups of $1.31 \times$, $2.03 \times$ and $2.32 \times$, with 2, 4 and 2x4 cores, respectively. Clearly, the benefits of parallelization are less pronounced when applied in tandem with the engineering enhancements. In the next sections, we analyze where run-time is being spent in our parallel implementation, and then describe improvements to our initial parallelization approach.

A. Profiling

In this section, we categorize the different phases of the serial and parallel PathFinder algorithms to better analyze and improve the overall run-time. The run-time of sequential VPR routing can be split into three categories. The first category includes the run-time associated with reading the placement and netlist files from disk, allocating and loading the routing resource graph in memory, performing a final timing analysis, and writing the final results to disk. We do not include this run-time in our analysis. The second category is the route all signals function described in Section II. The third category includes the portion of run-time spent updating delay and slack information at the end of each PathFinder iteration. We separate categories 2 and 3 because in this work, we do not attempt to parallelize category 3, which represents approximately 7% of run-time in our enhanced sequential algorithm and approximately 2% of run-time for the baseline VPR. We include only category 3 run-time in our final speed-up figures.

Although the maximum speed-up of a parallel algorithm implementation over its serial equivalent is $N \times$, where $N$ is the number of processors, speed-ups close to this maximum are rare for algorithms that are not embarrassingly parallel. Several factors prevent us from reaching this maximum. In our parallel implementation of PathFinder, the category 2 run-time mentioned above incurs its own overhead that is not present in the serial implementation. With $N$ total processors, for a processor $i$, we use $route_{time_{i/N}}$ to represent
the total time spent routing signals by the processor, and we use $route_{time_i,N,j}$ to represent the time processor $i$ spends routing signals in iteration $j$ of PathFinder. We use $wait_{time_i,N,j}$ to represent the time processor $i$ spends waiting for updates from other processors in PathFinder iteration $j$. Using these run-time metrics, we attempted to breakdown run-time further as follows:

1) **Ideal time:** We first define the ideal time to be that corresponding to a perfect parallelization without any stalls:

$$t_{\text{ideal,}i/N} = route_{time_i,1_1}/N$$

The ideal time is the sequential time divided by $N$ (the number of processors).

2) **Stall time:** Our parallel PathFinder implementation uses blocking receive messages to get updates about signals being routed in other processors. If a processor issues a blocking receive call but the corresponding update has not yet been sent, its execution will stall. This is due to inaccuracy in our work estimates. We define stall time as the time penalty incurred by processor stalling that could have been avoided had all receive messages been delayed to the end of an iteration. This is an important distinction because it distinguishes stall time from load imbalance time. Alternatively, the stall time can also be viewed as the time penalty incurred from processor stalling given perfect load balancing. Stall time is estimated as follows:

$$t_{\text{stall,}N} = \sum_{j=0}^{I} \left( \min_{0 \leq i < N} wait_{time_i,N,j} \right)$$

where $I$ is the total number of PathFinder iterations and $t_{\text{stall,}N}$ refers to the stall time across all processors when using $N$ total processors.

The stall time is the wait time due to factors other than a load imbalance. Since the processor with the least amount of work experiences no delay due to load imbalance, all of its wait time can be attributed to stall time, hence the min in (2) above.

3) **Load imbalance:** At the beginning of each PathFinder iteration, the signals are partitioned into $N$ sets, such that the sum of the work estimates in each set is approximately equal. Each of these sets is then assigned to a processor. If the work estimates are inaccurate and the sum of the actual amount of work performed by one processor is more than that performed by another, there will be a load imbalance. For a PathFinder iteration, the time penalty incurred as a result of a load imbalance is approximated as the maximum $route_{time}$ minus the average $route_{time}$:

$$t_{\text{imbalance,}N} = \max_{0 \leq i < N} route_{time_i,N} - \frac{1}{N} \sum_{i=0}^{N} route_{time_i,N}$$

(3)

where $t_{\text{imbalance,}N}$ refers to the imbalance time across all processors when using $N$ total processors.

4) **Slower convergence:** It is possible that stale congestion information in a processor causes it to make sub-optimal routing decisions, leading to slower resolution of over-utilized routing resources. As a result, the number of PathFinder iterations may increase and in general, slower convergence will lead PathFinder to do more work. Let $work_{sum,N}$ represent the total amount of work done when $N$ processors are used, computed as the sum of the total number of routing resource nodes explored for all signals over all PathFinder iterations. The run-time penalty incurred due to the increased work is estimated as follows:

$$t_{\text{converge}} = \left( \frac{work_{sum,N}}{work_{sum,1}} - 1 \right) \times run_{time_{1/1}}$$

(4)

where $run_{time_{1/1}}$ represents the sequential run-time.

5) **Serial time:** Near the end of the routing process, the number of cores being used is reduced to improve convergence. The benefit of doing so can outweigh the cost of using fewer cores. The decision of when to reduce the number of used cores can have a large impact on overall run-time. Reducing core count too soon will waste run-time, as spending even a modest portion of the algorithm in serial can quickly
reduce the resultant speed-up. Conversely, reducing core count too late will waste time if out-of-date congestion information is negatively impacting convergence. Consider an $N$-core PathFinder run where the number of processors is reduced towards the end of the routing process to aid convergence, and let $I_{final}$ be the set of (final) PathFinder iterations where fewer than $N$ processors are used (i.e. at least one processor has been made inactive). $t_{serial}$ is the estimated run-time penalty that results from using fewer than the total number of available cores, and we define it as follows:

$$
t_{serial,N} = \sum_{j \in I_{final}} (\max_{0 \leq i < N} (route_{time_{i,j}}/N,j) - t_{ideal,N,j})$$

(5)

where $t_{ideal,N,j}$ represents the ideal time for iteration $j$ of PathFinder had all $N$ processors had been kept active.

6) Profiling Results: Figs. 8 and 9 show the run-time contributions of $t_{stall}$, $t_{imbalance}$, $t_{converge}$, and $t_{serial}$ with and without our engineering enhancements to the baseline sequential VPR routing algorithm. It should be noted that there are other effects not taken into account that make the total of the run-time contributions in these figures an underestimate by as much as 4%. Nevertheless, they are useful for determining the sources of non-ideality.

A notable difference in Fig. 9 compared to Fig. 8 is the significant increase in $t_{converge}$. It is possible that this could be improved by reducing the number of used processors at an earlier PathFinder iteration, though this would likely result in an increase in $t_{serial}$.

Another notable difference is the increase in $t_{imbalance}$ as a percentage of total run-time. This increase stems from the difficulty of estimating how much work it will take to route a signal, when it is possible that the signal will not be ripped up and re-routed. Even if there is congestion on a signal’s route at the beginning of an iteration (when work estimates are made), when it comes time to route the signal, that congestion may have been resolved. In such a case, the work estimate is a very poor proxy for the actual time needed to process that signal, which is zero (as the signal is not re-routed).

VI. PARALLEL ROUTING IMPROVEMENTS

Improvements to our parallel routing techniques are needed because of the difficulty of parallelizing a PathFinder implementation that incorporates the engineering enhancements of Section III. We offer two methods in this section that aim to improve stall time.

A. Enhancements to the First PathFinder Iteration

In VPR’s implementation of PathFinder, the first iteration is used to discover the best-case delay for each signal, ignoring congestion. This means that the order in which the signals are routed does not matter, since congestion does not play a role in which route is chosen for each signal\(^4\). We can take advantage of this property through a more flexible allocation of signals to processors. Instead of deciding at the beginning of a PathFinder iteration which signals will be routed by which processor, we keep a pool of signals in processor 0. The other processors then issue requests to processor 0 for signals to route. Processor 0 frequently polls for these requests, to ensure that there is very little stall time.

B. Hierarchical Geographic Routing

In VPR’s PathFinder implementation, not all signals have the potential to use the same routing resources. VPR computes a bounding box for each signal, outside of which routing resources cannot be used [18]. This bounding box is the smallest bounding box that contains all pins of the signal plus a buffer in each direction, which by default is equal to 3 FPGA tiles. If two signals have non-intersecting bounding boxes, they cannot use any of the same routing resources, and we say that these signals are geographically independent.

1) Recursive Bipartitioning Technique: We developed a recursive geographic bipartitioning technique that takes advantage of geographical independence between signals. The way in which geographic bipartitioning is applied recursively is best visualized using a slicing tree, shown in Figure 10 for 8 processors. Each node in the slicing tree holds a signal set that is geographically independent from the signals sets in all other nodes at the same level. The processors responsible for routing a node’s signal set are labelled.

We first describe the top level of recursion, which corresponds to the top level of the slicing tree. At the top level, the FPGA is split into two regions. We call these the left and right regions, though they could also be the top and bottom regions.

\(^4\)By default, the VPR routing cost function in the first iteration places a small weight on congestion. In order to make the first iteration entirely congestion agnostic, we altered the cost function to completely ignore congestion.
A signal with a bounding box that resides entirely within one region is geographically independent from a signal with a bounding box that resides entirely within a different region. The signals in the left region form the left_signal_set, while the signals in the right region form the right_signal_set. The signals with bounding boxes that overlap both regions form the crossing_signal_set. Because no geographic independence can be guaranteed between signals in the crossing_signal_set, all processors must receive updates about these signals when they are routed. For this reason, this set is assigned to the top level node in the slicing tree, labelled “P0-P7”. This level of the slicing tree is now fully processed. If there were only 2 processors, the recursive bipartitioning process would be complete.

The left and right regions can now both themselves be bipartitioned, further splitting the signals in the left_signal_set and the right_signal_set into two parts. To do this, we call the recursive bipartitioning function first on the left_signal_set, then on the right_signal_set, specifying the region of the FPGA that corresponds to each signal set, and that the signals should be split between N/2 processors. The base case is reached when a signal set is split between only 2 processors. When this occurs, the left_signal_set is assigned to the node in the slicing tree corresponding to one of the two processors, and the right_signal_set is assigned to the node in the slicing tree corresponding to the other processor. Figure 11 shows a possible result of the recursive-bipartitioning when using 8 cores. This figure shows that regions may be cut vertically or horizontally.

2) Routing in phases: Once the signals are assigned to nodes in the slicing tree, they must be routed. The routing begins by processing the node at the top level of the slicing tree (level 3). The signal set at this node is split between all processors such that the sum of the work estimates in each processor is approximately even. The signals are then routed, sending updates as explained in Section IV. All processors must finish routing their signals before continuing to the next routing phase (next level in the slicing tree). This ensures that each processor has up to date congestion information in its assigned region at the beginning of the next phase.

The routing process continues by routing signals at next level of the slicing tree (level 2), dividing signals at each node between the processors assigned to that node. In the slicing tree example, the node labelled “P0-P3” at Level 2 contains signals that are split between processors 0 to 3 and are then routed, while the node labelled “P4-P7” contains signals that are split between processors 4 to 7 and are then routed. At this level, no communication is necessary between processor sets 0-3 and 4-7. Processors 0-3 must finish routing all their assigned signals at this level before any of these processors may continue to the next phase of routing. The same is true of processors 4-7.

This routing continues in phases until the leaf nodes at level 0 of the routing tree have been processed. Updates about geographically independent signals that were not sent/received during the routing process are now sent/received. This ensures that each processor has an up to date view of the congestion across the entire FPGA before beginning the next PathFinder iteration.

Figure 12 shows the communication between processors that occurs at each phase in the routing process when using 4 processors. Each phase is labelled with the level of the slicing tree that holds the signals routed at this phase.

3) Choosing a cut: There are three factors that we consider to determine the quality of a cut that bipartitions a region of the FPGA. In our description of these factors, we use the terms work_left and work_right to refer to the sum of work estimates in the left_signal_set and the right_signal_set, respectively.

- The first factor to consider is the load imbalance in the crossing_signal_set that results from having to split it between processors during the routing process. It may not be possible to split the signals in the crossing_signal_set in a way that results in approximately equal amounts of work in each processor. This can be the case if there are many processors or if there are signals with very high work estimates (since a single signal is not split between processors). In such a case, it is beneficial to move additional signals from the left and right signal sets to the crossing_signal_set. When the crossing_signal_set is partitioned, these extra signals will be assigned to processors that have very little work to do, thereby improving load balancing. We measure the load imbalance in the crossing_signal_set by temporarily partitioning the signals in this set between N temporary processors. We
move signals from the left or right signal set (whichever has more work) to the temporary processor with the least amount of work until the maximum work of the temporary processors divided by the average work of the temporary processors is less than 1.05.

- The second factor to consider is that work_left and work_right should be approximately equal. This is important because it reduces the load imbalance at the end of each routing phase. We can make work_left and work_right approximately equal by moving signals that cause an imbalance from the left or right signal sets to the crossing_signal_set. Signals are moved from the left or right signal sets until work_left is approximately equal to work_right. We refer to the average of the now approximately equal work_left and work_right as work_balanced.

- At later routing phases, there are fewer processors communication updates, so there is less opportunity for stall time. Choosing cuts such that more work is done at later routing phases is beneficial to stall time. By that reasoning, it is important to maximize work_balanced, since we want to assign as many signals as possible to lower levels in the slicing trees.

We use a greedy algorithm to choose a vertical or horizontal cut that maximizes work_balanced after having adjusted for load imbalance in the crossing_signal_set and between the left_signal_set and the right_signal_set. Maximizing work_balanced leads to more signals being routed at later routing stages, when communication between processors is reduced.

Fig. 12. Communication between 4 processors during routing when using hierarchical geographic routing

Fig. 13. Run-time contributions for improved parallelization of baseline VPR.

VII. EXPERIMENTAL STUDY

We now evaluate our improved approach to parallel routing (with geographic partitioning) using the same methodology as described in Section V. Table IV shows the run-time (in seconds) as a function of the number of VPR instances (processes) for parallelizing the baseline VPR, as well as VPR with our engineering enhancements.

Without our enhancements to the sequential algorithms, we observe speed-ups of $1.53 \times$, $2.20 \times$, and $2.80 \times$ with 2, 4 and 2x4 processor cores, respectively. Since these speed-ups are similar to the preliminary results reported in Section V, it is interesting to compare Fig. 13, which shows the run-time contributions of $t_{\text{stall}}$, $t_{\text{imbalance}}$, $t_{\text{converge}}$, and $t_{\text{serial}}$ after our parallel routing improvements, with Fig. 8, which shows the run-time contributions before the improvements. Although using our hierarchical geographic partitioning technique did significantly reduce $t_{\text{stall}}$, there was an approximately equal increase to $t_{\text{serial}}$. We believe the reason for this is that geographic partitioning increases the likelihood of high-fanout bounding-box signals being routed concurrently. Such signals are not “local” to any partition, and must be routed in the first phase of routing, when all processors are communicating. Consequently, the congestion associated with such signals is more often out-of-date between processors, making congestion difficult to resolve, increasing serial time.

With our engineering enhancements to the baseline VPR (Section III), we see speed-ups of $1.51 \times$, $2.33 \times$, and $2.84 \times$, with 2, 4 and 2x4 processor cores, respectively. When comparing Fig. 14, which shows the run-time contributions of $t_{\text{stall}}$, $t_{\text{imbalance}}$, $t_{\text{converge}}$, and $t_{\text{serial}}$ after our improvements, with Fig. 9, it is clear that stall time is significantly reduced by the geographic partitioning technique.

Finally, combining our sequential and parallel improvements and comparing the run-times to baseline VPR (second last row of the table), we observe speed-ups of $5.02 \times$, $7.71 \times$ and $9.42 \times$ using 2, 4, and 2x4 processors, respectively. We consider the results encouraging and we believe they should keenly interest FPGA vendors and users. For comparison, we note that Altera reported a speed-up of $2.2 \times$ using 4 cores in their recent parallel placer work [1].

A. Impact on Quality of Results

In our parallel routing approach, the VPR instances executing concurrently operate with slightly stale congestion...
Table IV

RUN-TIMES OF IMPROVED PARALLEL ROUTING (WITH GEOGRAPHIC PARTITIONING) VS. THE SEQUENTIAL BASELINES WITH AND WITHOUT ENGINEERING ENHANCEMENTS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Cores (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>2x4</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enhanced</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cl_cordic</td>
<td>38.98</td>
<td>7.96</td>
<td>4.75</td>
<td>2.27</td>
<td>1.13</td>
<td>2.06</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>cl_decomp_g31</td>
<td>58.72</td>
<td>26.78</td>
<td>17.17</td>
<td>11.44</td>
<td>11.43</td>
<td>7.37</td>
</tr>
<tr>
<td></td>
<td>mem</td>
<td>22.95</td>
<td>16.36</td>
<td>10.15</td>
<td>7.25</td>
<td>7.54</td>
<td>4.89</td>
</tr>
<tr>
<td></td>
<td>des_perf</td>
<td>8.99</td>
<td>4.64</td>
<td>2.79</td>
<td>2.50</td>
<td>2.23</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>ex1010</td>
<td>13.49</td>
<td>8.60</td>
<td>5.80</td>
<td>3.91</td>
<td>3.06</td>
<td>2.95</td>
</tr>
<tr>
<td></td>
<td>firc</td>
<td>10.41</td>
<td>6.97</td>
<td>5.84</td>
<td>5.13</td>
<td>2.53</td>
<td>1.79</td>
</tr>
<tr>
<td></td>
<td>mac2</td>
<td>25.86</td>
<td>18.10</td>
<td>17.52</td>
<td>11.57</td>
<td>6.00</td>
<td>4.34</td>
</tr>
<tr>
<td></td>
<td>s38417</td>
<td>24.15</td>
<td>16.19</td>
<td>14.07</td>
<td>7.09</td>
<td>5.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>rs_decoder_2</td>
<td>12.61</td>
<td>7.70</td>
<td>4.51</td>
<td>3.74</td>
<td>3.99</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td>spla</td>
<td>11.99</td>
<td>7.26</td>
<td>5.01</td>
<td>4.16</td>
<td>4.07</td>
<td>2.76</td>
</tr>
<tr>
<td></td>
<td>sv_ship0</td>
<td>14.38</td>
<td>8.62</td>
<td>5.86</td>
<td>5.87</td>
<td>4.24</td>
<td>2.93</td>
</tr>
<tr>
<td></td>
<td>sv_ship1</td>
<td>144.55</td>
<td>105.24</td>
<td>62.59</td>
<td>55.02</td>
<td>40.50</td>
<td>25.95</td>
</tr>
<tr>
<td></td>
<td>sv_ship2</td>
<td>475.21</td>
<td>341.48</td>
<td>264.61</td>
<td>218.68</td>
<td>137.45</td>
<td>78.41</td>
</tr>
<tr>
<td></td>
<td>geometric</td>
<td>22.01</td>
<td>14.38</td>
<td>9.99</td>
<td>7.36</td>
<td>6.64</td>
<td>4.38</td>
</tr>
<tr>
<td></td>
<td>speed-up vs.</td>
<td>1.00x</td>
<td>1.53x</td>
<td>2.20x</td>
<td>2.80x</td>
<td>3.31x</td>
<td>5.02x</td>
</tr>
<tr>
<td></td>
<td>baseline 1 core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>speed-up vs.</td>
<td>1.00x</td>
<td>1.53x</td>
<td>2.20x</td>
<td>2.80x</td>
<td>3.31x</td>
<td>5.02x</td>
</tr>
<tr>
<td></td>
<td>enhanced 1 core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 14. Run-time contributions for improved parallelization of VPR with engineering enhancements.

information, and consequently, it is conceivable that quality-of-result could be adversely impacted. We studied the impact on quality-of-result using two metrics: 1) the total number of used FPGA wire segments after routing, and 2) the post-routing critical path delay. The number of wire segments is a proxy for the total capacitance of the routing for all design signals.

We found the impact of parallelization on quality-of-result to be very small. The average change to both the number of wire segments and to the critical path delay was less than 1%. Table V shows the critical path delay and the total wirelength that results from running with 4 cores with our parallel improvements, as compared to 1 core. Critical path delay actually improved slightly in the parallel version, which we attribute to algorithmic noise.

VIII. CONCLUSIONS AND FUTURE WORK

Parallel computing is a promising avenue for reducing the run-time of FPGA CAD tools. In this paper, we presented new techniques for improving the performance of parallel routing for FPGAs as well as described two engineering enhancements to improve the run-time of sequential negotiated congestion FPGA routing. We presented a method of partitioning signals using a hierarchical geographic approach such that communication requirements between processors is minimized. Additionally, we showed how engineering enhancements to the sequential routing algorithm can lead to large speed-ups. We demonstrated that our parallel routing techniques work well for parallelizing a more realistic, aggressive, sequential routing algorithm. Results show that parallelization alone provides about 2.3 × speed-up using 4 cores, and that parallelization combined with the engineering enhancements provides over 7 × speed-up over a popular baseline FPGA routing implementation (VPR). The parallel router produces deterministic/repeatable results, with no considerable impact to quality-of-result (performance or wirelength).

As future work, we believe that new techniques are necessary to push parallel FPGA routing to the many-core era. The main barrier to continued scalability is the presence of high-fanout signals that span a large portion of the chip. We see two potential methods to extract parallelism from these
signals. The first is to use more fine-grained techniques, as we did in [2]. Unfortunately those techniques only provided modest speed-ups, so improvements would have to be made. The second potential method is to break these signals into smaller parts, either in the routing stage of the CAD flow or at an earlier stage. As related future work, the routing resource graph could be split between processors, necessitating a more strict geographic partitioning, but allowing memory use to scale moderately as the number of processors increases.

REFERENCES

APPENDIX A
RESPONSE TO REVIEWER COMMENTS

A. Reviewer 1

This interesting paper discusses techniques to parallelize the popular VPR FPGA router on multicore architectures. The authors use a standard multicore MPI protocol for information exchange. Important issues such as data synchronization between various VPR instantiations are discussed. The performance improvement of VPR is quite good and holds promise for future manycore architectures.

Overall the paper is well written and provides a useful contribution. I had no problem understanding all aspects of the paper although my understanding was probably assisted by my knowledge of VPR.

Most of my comments are minor in nature

1) VPR is a SPEC benchmark. The authors might want to mention this somewhere

>>> Thank you for the suggestion. We have added the following text in the introduction: “VPR is also part of the SPEC benchmark suite.”

2) I was a little confused about the "deterministic" result of the parallelized VPR. Does the user really get the exact same result every time? I was under the impression that the VPR router has some randomness based on the selection of equal cost nodes. Maybe the timing information makes the likelihood of nodes having the exact same cost less. On page 2 there is a comment that users expect identical results to be produced by routing runs. Perhaps it is more accurate to say that users expect the same quality of results to be generated each time.

>>> Thank you for the comment. In fact, the VPR router is completely deterministic because the order in which nodes are added to the priority queue, although arbitrary, remains identical from run to run. As a result, the same ordering in the priority queue is imposed on equal cost nodes, making the removal of nodes from the queue deterministic. Because determinism is mandatory for industrial CAD tools, we have ensured that our router is deterministic. We have added the following text in the introduction to emphasize that VPR routing is deterministic:

“The non-parallel router in VPR is completely deterministic because nodes are added to a priority queue in the same order on each run, and it is important to maintain this property.”

3) In Section III.B the authors indicate that "uncongested" signals are not rerouted. Is this the current congestion or the history value?

>>> We have modified some text in section III.B to clarify that we are referring to current congestion. We avoid explicitly mentioning the terms current and historic congestion because they would require further explanation. This is the text we have modified:

“In the original PathFinder algorithm [6], as well as in its implementation in VPR, each signal is ripped-up and re-routed in every iteration, as depicted in Fig. 1. That is, even signals that do not currently use over-capacity routing resources are re-routed each iteration. We refer to such signals as uncongested signals.”

4) Does it make a difference if a node is uncongested but has a high history value?

>>> It does in fact make a difference. Convergence is slightly improved when ripping up and rerouting signals that have a high historic congestion value (but no current congestion), but the time per iteration increases enough that it is not worthwhile. We have added the following text to answer this question:

“We also experimented with only skipping signals that remain uncongested for multiple consecutive PathFinder iterations. Although this reduced the number of necessary PathFinder iterations, it resulted in an overall increase in run-time.”

5) Section IV.A is nicely written

>>> Thank you!

6) Typo on page 8 - "A notable different..."

>>> We have corrected this typo, thank you.

7) In Section VI.B, how many routing iterations are used for each routing phase?

>>> We have modified the following text to section V: “Across all runs, each circuit was routed using a fixed channel width of 1.3x the minimum channel width needed to route the circuit in the single-core case with a maximum of 100 PathFinder iterations”

8) Section VIII - Is "presense" the proper spelling in Canada?

>>> No it isn’t. Thank you for pointing out this typo.

9) The reference section has a lot of misspellings and other problems The author list in reference [1] has problems. FPGA is not capitalized in other references.

>>> Thank you. We have fixed the reference problems.

B. Reviewer 2

1) 1. Generally, the paper is well written and easy to understand. But it will be much better if the authors can use pictures to explain the first engineering enhancement : propagating logic element outputs.
Thank you for the suggestion. We have modified section III. A to make use of figures to illustrate the concept. Please see Figs. 2 and 3

2) Also Fig. 1 is misleading. My understanding is that the first box "route all signals (permit shorts)" is exactly the same as the third box "route all signals". But the third box does not state "permit shorts" and readers may think no short is permitted for the third step.

Good point! We have changed the first box to read "ignore congestion" instead of "permit shorts" since in this step, only timing is considered.

3) The paper has presented several good ideas to accelerate the FPGA router. However, I think the value of the second engineering enhancement "skipping the re-routing of uncongested signals" is questionable. Obviously this technique provides speedup at the cost of quality. Depending on the particular problem, the quality may be greatly affected by this technique. If you look at the spla benchmark in TABLE 1, using the technique increases the critical path delay by 19% and the wire length also increases by 4.5%.

This is a good point. We agree that there is some potential loss in QoR because there are fewer route options in late PathFinder iterations. If a critical path is still unrouted in these iterations, it may be the case that it is forced by congestion costs to use longer routes. For most circuits, however, this problem does not occur. The geometric mean impact on critical path delay of skipping uncongested signals is only 1%. We have added the following text to be more up front about the potential downside of using this approach:

“Not ripping up and re-routing uncongested signals does have the potential to harm quality of results as critical routes may be forced to choose less desirable paths in late PathFinder iterations, as evidenced by the critical path delay numbers for the benchmark circuit “spla” in Table I. For most benchmarks, however, the effect on quality of results is negligible.”

4) As pointed out by the authors, the 2x4 data column does not reflect the scalability of the algorithm. So it will be nice if the authors add some analysis and project the scalability of the algorithm on more cores. With the partitioning strategy and the final serial phase of the routing process, the scalability is not very obvious.

As you say, the scalability is not very obvious. It is difficult to project future scalability because of this. In this article, we have isolated different run-time effects (Figs. 8,9,13,14) and outlined trends using this data.

In addition to the analysis already in the article, we have run additional experiments to measure the degree to which scalability is affected when using 2 quad-core processors across a network for the 2x4 case. We first measured the overall latency of all non-blocking send messages issued for our benchmarks when running all processes on a single chip. We then ran the same experiment, with processes on two different processors with communication between the processors over a network. The results indicate that the latency of non-blocking send messages that travel across a network is 5 times longer than the latency of non-blocking send messages within a chip. Noting that for our 4x2 case, non-blocking send messages take up approximately 6% of run-time, the overall run-time could be reduced by 5.4% by running all eight processes on a single 8-core chip (ignoring memory effects). We have added the following text in the article to present this scalability analysis:

“We measured an approximately 5x latency difference when sending a non-blocking message across a network (between processors) rather than between cores in a processor. This leads to an approximately 5% run-time penalty when some messages have to travel across a network.”

5) Table V is the quality comparison between the final parallel implementation and the "enhanced" 1-core implementation, right? The authors should clarify this.

Thank you, that is correct. We have modified the caption to offer clarification.

C. Reviewer 3

The paper presents the parallelization of the algorithm for FPGA routing and its porting on multicore processors. The parallel programming model used is MPI. Authors defined and applied also two main modifications on the sequential algorithm, namely about output propagation and skipping of rerouting uncongested signals.

Strength points:
1) The topic addressed by authors is relevant and compliant with the call of the journal.
2) The paper is well written and structured.

Thank you.

Weak points:
1) The main concern is on the programming model chosen by authors. I think that, given the application characteristics and the parallelization pattern, MPI is not the best way to do it. MPI is a well-known model, but it has usually a higher overhead and poor scalability when dealing with intra-core communication. I think that this kind of application fits very well with the openCL programming model. Moreover, openCL automatically enables the exploitation of heterogeneous systems enhanced with hardware accelerators, like GPUs. I strongly believe that the solution openCL+MPI (for inter-processor communication) is a better choice. Authors should explain and motivate why they selected MPI.
We hadn’t considered OpenCL as a programming model but it is certainly an interesting idea. We mainly chose MPI for its maturity and because it does not use a shared-memory model, which, for our purposes, made parallelization much easier. Since our parallelization is very coarse-grained, using a shared memory model would have required making many data structures in VPR thread safe. As VPR contains approximately 50,000 lines of code, this is a daunting task. Using MPI, we were able to get a parallel implementation of the VPR router working in a matter of weeks, after which we were able to focus on the efficiency of the implementation. The techniques presented in this paper are not limited to an implementation that uses MPI and could be adapted to an OpenCL-based programming model.

2) Authors should measure also the overhead difference between inter- and intra-processor communication. We have run additional experiments to measure the difference between inter- and intra-processor communication. Please note that the description of our experiment is also included in a response to reviewer 2. We first measured the overall latency of all non-blocking send messages issued for our benchmarks when running all processes on a single chip. We then ran the same experiment, with processes on two different processors with communication over a network. The results indicate that the latency of non-blocking send messages that travel across a network is 5 times longer than the latency of non-blocking send messages within a chip. Noting that for our 4x2 case, non-blocking send messages take up approximately 6% of run-time, the overall run-time could be reduced by 5.4% by running all eight processes on a single 8-core chip (ignoring memory effects). We have added the following text in the article to present this scalability analysis:

“We measured an approximately 5x latency difference when sending a non-blocking message across a network (between processors) rather than between cores in a processor. This leads to an approximately 5% run-time penalty when some messages have to travel across a network.”

3) The considered case studies seem very simple: the maximum execution times reported in tables are only of the order of 150 seconds. In fact, in the introduction it is claimed that “long run times slows engineering productivity, raise costs and are a strong impediment to the widespread adoption of FPGAs by software developers”, but then the considered benchmark circuits have a routing time that takes, in the baseline case, at most two minutes. Thus, authors should try their framework with more complex benchmarks. In fact, it would be more interesting if the authors can show the effects of these enhancements on cases where the speedup gives an effective benefit or, at least, prove that it remains constant when increasing by significant quantities the problem size.

>>> Thank you for the feedback. We agree that the sizes of academically available benchmarks are regretfully small. We have added the largest benchmark for FPGA research that we know of to the set of benchmarks in this article (sv_chip2). The single core routing run-time is 477 seconds. The parallelization speedups for this benchmark are in fact better than the geometric speedups, indicating good scalability with respect to circuit size.

4) Finally, the authors need to cite some fact regarding the impacts on memory requirements introduced by their proposal with respect to parallel platforms requirements.

>>> We have added the following text to acknowledge that memory requirements increase with the degree of parallelization:

“Using MPI’s distributed memory programming model, we were able to parallelize VPR without having to make its data structures thread safe, which reduced the amount of effort necessary to create a functional parallel implementation. One of the drawbacks of using a distributed memory model is that the overall memory usage is proportional to the number of cores that are running VPR in parallel.”

Section VIII describes future work that could address this issue:

“the routing resource graph could be split between processors, necessitating a more strict geographic partitioning, but allowing memory use to scale moderately as the number of processors increases.”

5) The paper is a bit on the verbose side. For example, you can cut a lot of the descriptions of the routing, and just mention Dijkstra and move on.

>>> Thank you for your feedback. We appreciate that for someone so familiar with routing algorithms, the second half of column 1, page 2 does not provide any new information. We have attempted to keep the background material as brief as possible, though we also wish to make the article as accessible as possible. This may make it seem a bit verbose, but we believe that many readers will appreciate the review of maze routing, especially considering that it is only 1/2 of one column.

D. Additional comments from reviewers received late

1) I’m not sure if the solution in Fig 2 is the best way to enforce determinism. If you allow for determinism, but not necessarily the same result as VPR, then you can let all nets be routed independently, and only update the congestion map at the end of the routing iteration when all nets have been routed (i.e., no syncing before that). This strategy is different from VPR’s in the sense that in VPR routing an individual net in the same iteration does affect the congestion map which is used by another lower priority net. But on the other hand, you could change the weighting of the nets to that congested resources appear less congested to higher priority nets, hence emulating the VPR’s effect (say, a routing...
resource’s cost is $\alpha \cdot \text{congestion} - \beta \cdot \text{netCriticality}$). I agree that this might hurt routing quality, but you can either do more iterations or just use this strategy of mine in the first, say, 4/5 of the iterations and then switch to yours.

>>> Thank you for the suggestion. We had previously attempted to reduce the frequency with which updates are received, which is a half way point between our current solution and the one that you have suggested. Unfortunately, this leads to slower convergence of the algorithm and ultimately to higher overall run-time. It appears as though having up-to-date congestion is quite important for PathFinder to converge. Your suggestion to use this method for the first iterations then switch to full communication makes intuitive sense, and in fact works for some benchmarks. For others, however, having up-to-date congestion information is important even in the first iterations.

2) As a followup to the above suggestion, you can use wall clock time for a better load balancing metric.

>>> This would certainly lead to better load balancing, but would unfortunately lead to non-determinism, since the partitioning of signals among processors would be dependent on wall time, which is itself non-deterministic. We explore wall clock time as a load balancing metric for comparison purposes only in Fig.6 and Fig.7. We also mention briefly that wall clock time would lead to non-determinism:

“Wall clock time is included for comparison only, as using this metric would lead to non-determinism.”