Deterministic Multi-Core Parallel Routing for FPGAs

Marcel Gort and Jason H. Anderson

Dept. of Electrical and Computer Engineering, University of Toronto
Toronto, Ontario, Canada
{gortmarc|janders}@eecg.utoronto.ca

***Abstract*—We consider coarse and fine-grained techniques for parallel FPGA routing on modern multi-core processors. In the coarse-grained 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. In the fine-grained approach, the task of routing an *individual* load pin on a signal is parallelized using threads. Specifically, as FPGA routing resources are traversed during maze expansion, delay calculation, costing and priority queue insertion for these resources execute concurrently. The proposed techniques provide deterministic/repeatable results. Moreover, the coarse and fine-grained approaches are not mutually exclusive and can be used in tandem. Results show that on a 4-core processor, the techniques improve router run-time by $\sim$2.1$\times$, on average, with no significant impact on circuit speed performance or interconnect resource usage.**

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 multi-core 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 [2].

In this paper, we present techniques for parallel FPGA routing. Two different approaches are proposed, which we refer to as *coarse* and *fine*-grained. 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. We use MPI (message passing interface) as the communication protocol in our coarse-grained approach [1], [20]. In the fine-grained approach, we parallelize the routing of an individual load pin on a signal. Specifically, we parallelize aspects of the low-level maze expansion search (exploration) of the FPGA routing fabric. Threads are used as the fine-grained parallel programming paradigm. Both approaches offer deterministic/repeatable results. We implement and demonstrate both approaches using the VPR framework [12].

In this paper, we make several key contributions:

- The first work on parallel FPGA routing for modern multi-core processors.
- The first deterministic FPGA routing parallelization approach for non-planar FPGA routing architectures.
- A novel parallelization approach that combines coarse and fine-grained parallelism.
- The first work to use MPI for parallel FPGA routing.
- A detailed evaluation of load-balancing techniques for parallel negotiated congestion routing.

Experimental results demonstrate the efficacy of the proposed techniques and suggest that both coarse and fine-grained approaches are viable, depending on the memory architecture of the processor on which the router is executed. 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: threads and MPI. The proposed parallel routing techniques are described in Section III. An experimental study is presented in Section IV. Conclusions and suggestions for future work are offered in Section V.

II. Background

A. FPGA Routing

The two largest FPGA vendors, Xilinx and Altera, use a variant of the PathFinder negotiated congestion routing algo-

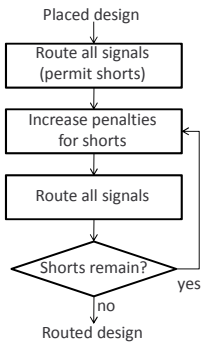

Fig. 1. Negotiated congestion routing flow.

rithm [14] in their commercial routers [17], [16]. PathFinder is also used in the publicly-available VPR FPGA placement and routing framework [12], which we parallelize in this work. 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 coarse-grained approach to parallelization accelerates the "route all signals" step in the negotiated congestion flow.

At the heart of negotiated congestion routing is maze expansion [11] – the algorithm used to route a load pin on a signal and the most computationally expensive aspect of FPGA routing. An understanding of our fine-grained parallelization approach depends on knowing the details of maze expansion, so we review the basics here. 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 modelling is used. Our fine-grained approach to parallel routing accelerates the maze expansion used to route a load pin.

B. Parallel FPGA Routing

Chan and Schlag were the first to parallelize negotiated congestion FPGA routing [7], showing impressive speed-up results of 2.5× using 3 processors. Their work bears some similarity to our coarse-grained 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.

Another work by Cabral et al. described parallel FPGA routing specifically for a "planar" routing architecture [5]. 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 [21]) used planar routing, it has since been shown to negatively impact routability. Modern FPGA interconnect is not planar [3], [22], making Cabral's work inapplicable today.

There is considerable prior work on parallelizing the single source shortest path problem [15], [8], [9] 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 ∼53×, 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 Environments

We use two parallel programming models in our parallel router: threads and MPI. Threads are a popular way to parallelize the execution of several tasks within a single process. The threads within a process share memory, which can lead to race conditions and unpredictable behavior when multiple threads read/write the same memory location in non-deterministic ways. Semaphores and barriers are used to manage access to shared memory, and synchronize thread

execution at specific program points. The attraction of threads stems from their *lightweight* nature – threads can be created rapidly, and context switching between threads on a processor is faster than context switching between entire processes.

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 coarse-grained parallel routing approach.

III. Multi-Core Parallel Routing

A. Coarse-Grained Parallel Routing

Our coarse-grained 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.

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 [6].
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[1]. 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 signal sets. After a VPR instance routes a signal, it sends a non-blocking update to all of the other VPR instances. The

[1]When metric #3 is used, there is no history available in the first PathFinder iteration, so we assign each process an equal number of signals in that iteration.

update message contains the signal's routing (as well as the number of routing resource nodes visited while maze routing 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. 2. 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[2].

The pseudo-code for our parallel PathFinder implementation

[2]We 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.

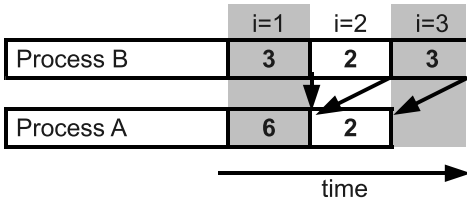

Fig. 2. Blocking receives issued by processes during a PathFinder iteration using number of loads as signal run-time prediction metric.

is shown in Fig. 3. Note that Fig. 3 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. 3 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 $nextSig(SigSet[j])$ function returns the next signal to be routed by processor j.

Lines 1 and 20 in Fig. 3 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 processor 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.

Figs. 4 and 5 illustrate the effectiveness of the different signal time prediction metrics for two benchmark circuits: `clma` and `cf_fir_20_16_16`. The vertical axis shows the average stall time over all VPR instances as a percentage

```
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. 3. Pseudo-code for multi-core PathFinder with load balancing from the perspective of processor # $local$.

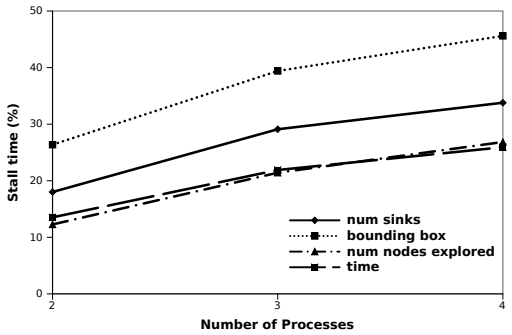

Fig. 4. Stall time (%) for signal route time prediction metrics for `clma`.

of the total time needed to route all signals. Stall time was measured using the hardware counters internal to an Intel Core 2 Quad microprocessor. There are four curves on each figure, 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. Results are given for 2, 3 and 4 VPR instances (processes).

The figures show that the best metric for predicting the run-time 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 Section IV.

1) 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

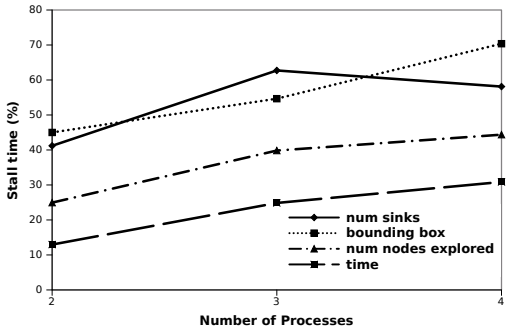

Fig. 5. Stall time (%) for signal route time prediction metrics for `cf_fir_24_16_16`.

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 alive). 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.

B. Fine-Grained Parallel Routing

Our fine-grained approach is to parallelize maze routing itself. We profiled the benchmark circuits used in our experimental study and found that ∼68% of routing time is consumed by costing the neighbors of a node during maze routing expansion, and then inserting the neighbors into the priority queue. Costing comprises RC delay calculation and congestion costing (evaluating the number of shorts). The priority queue in VPR is implemented using a binary heap, which exhibits $O(log\ m)$ insertion time, where m is the number of items in the queue.

Posix threads (pthreads) are used in our fine-grained parallel routing approach. If there are N processor cores available, we establish one main thread and $N-1$ helper threads. The main thread executes the serial portion of maze routing and $1/N$th of the parallel portion of the algorithm. The helper threads are only created once, and "busy-wait" during the serial portions of the algorithm (described below).

Sequential maze routing uses one priority queue to maintain the list of nodes waiting to be expanded. In our fine-grained parallel router, we use N separate priority queues – one associated with each thread. Each thread only adds nodes to its own priority queue. As a result, the queues are completely privatized during the parallel portion of the algorithm. Privatization is a standard parallel programming technique for achieving thread-safety.

The fine-grained parallel routing flow, shown in Fig. 6, is as follows:

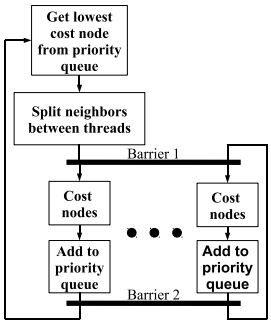

Fig. 6. Fine-grained algorithm flow.

- The main thread peeks at the nodes at the front of the N priority queues ($O(N)$ time) and pops the one with the lowest cost.
- The main thread "expands" the node to visit its neighbors, distributes the neighbors among all threads, then signals the helper threads that new nodes are ready to be operated on (using a barrier[3]).
- All threads, including the main one, cost and add their share of the nodes to their own priority queue. Since each thread has its own priority queue, no mutual exclusion or locks are needed.
- Once the main thread has finished costing and adding nodes to its priority queue, it waits in a second barrier for all other threads to be finished before again looking for the lowest cost node in the priority queues and repeating the loop.

Synchronization and Load Balancing: Two barriers are used in the fine-grained router: the first barrier indicates to the helper threads that work is ready; the second indicates to the main thread that the helper threads have finished their work. Typically, if a thread enters a barrier and cannot proceed past it, the operating system will put the thread to sleep. It will wake when it can safely exit the barrier. This is the case in C's pthreads library, where barriers are implemented using mutexes, rather than using spin locks. The time required to sleep/wake a thread is significant, especially when the granularity of parallelization is very fine. In our router implementation, using the pthread library barriers resulted in an order of magnitude slow down in routing time. Therefore, we implemented a low-overhead barrier using a busy-wait loop: threads "spin" until they can proceed through the barrier.

A round-robin approach is used to split the expansion neighbors between the threads. We found that this approach works well if the time needed to add a node to a priority queue is consistent across the threads. Unfortunately, the time needed for queue insertion depends on the number of elements in the queue, which may differ for each of the N queues. The priority queue imbalance is partly because the number of neighbors of any given node in the routing resource graph is not necessarily divisible by the number of threads (N). To combat this, we assign work to helper threads in a cyclic fashion, making each thread responsible for handling "extra" nodes periodically, and thereby better balancing the queue sizes across threads.

[3]A barrier is a thread synchronization construct placed at a particular location in a multi-threaded program. All threads must reach the location before any are allowed to proceed past the location.

TABLE I
RUN-TIMES FOR COARSE-GRAINED APPROACH. *2 QUAD CORE SYSTEMS ACROSS A NETWORK.

	Number of Processes				
Benchmark	**1**	**2**	**3**	**4**	**2×4***
cf_cordic_v_18_18_18	8.2	5.5	4.8	4.1	6.2
cf_fir_24_16_16	39.7	26.1	19.7	16.8	14.2
clma	23.5	17.0	13.5	11.8	9.8
des_perf	9.2	4.7	5.0	4.1	3.8
ex1010	13.9	9.4	8.4	6.2	5.3
frisc	10.2	6.4	5.5	4.7	3.8
mac2	26.8	20.9	19.8	17.9	15.2
paj_raygentop_ hierarchy_no_mem	21.4	17.5	15.5	12.8	13.2
pdc	23.3	15.6	12.3	9.5	7.6
rs_decoder_2	12.1	7.2	6.2	5.3	4.3
s38417	8.2	5.5	5.1	3.6	3.9
spla	12.2	9.0	6.9	5.0	5.4
geomean	15.3	10.3	8.9	7.2	6.8
speed-up	1.00	1.49	1.72	2.11	2.26

TABLE II
RUN-TIMES FOR COARSE-GRAINED APPROACH ON INTEL CORE I5.

	Number of Processes			
Benchmark	**1**	**2**	**3**	**4**
cf_cordic_v_18_18_18	10.1	5.6	6.1	4.9
cf_fir_24_16_16	44.0	32.1	25.7	21.0
clma	24.8	15.9	12.9	15.0
des_perf	8.8	5.4	5.7	4.5
ex1010	14.8	9.7	8.6	7.1
frisc	10.2	7.9	5.8	5.2
mac2	26.9	20.2	18.9	17.8
paj_raygentop_ hierarchy_no_mem	22.4	16.6	15.8	14.9
pdc	23.4	13.3	11.9	11.4
rs_decoder_2	12.7	7.8	8.7	5.5
s38417	8.8	6.3	5.6	4.8
spla	13.4	8.4	8.2	6.1
geomean	16.1	10.7	9.8	8.4
speed-up	1.00	1.51	1.64	1.92

IV. EXPERIMENTAL STUDY

Experiments were run on a system running Linux (Debian 2.6.26-21) with a Core 2 Quad Q9550 processor and 3 GB of memory. The Core 2 Quad has 4 processors. Processors with IDs 0 and 1 share a first 6 MB of L2 cache, while processors with IDs 2 and 3 share a second 6 MB of L2 cache. The Linux `sched_setaffinity` command was used to ensure that, when possible, threads were assigned to processors that share an L2 cache, promoting faster communication between threads. An Intel Core i5 system running Linux (Ubuntu 2.6.31-20) is also used for comparison. Although the focus of this paper is on parallelization for chip multiprocessors, since the coarse-grained parallel router uses MPI, it can work over a network and remains deterministic. Two networked Core 2 Quad systems were used together when the number of threads/processes exceeds 4.

The 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 [12]. Circuits were mapped into 4-input LUTs using

TABLE III
RUN-TIMES FOR FINE-GRAINED APPROACH.

	Number of Threads			
Benchmark	**1**	**2**	**3**	**4**
cf_cordic_v_18_18_18	8.2	7.4	12.6	12.7
cf_fir_24_16_16	39.7	36.6	72.4	78.3
clma	23.5	19.8	32.7	38.4
des_perf	9.2	7.6	12.8	14.4
ex1010	13.9	10.7	20.6	21.1
frisc	10.2	8.0	13.7	15.3
mac2	26.8	21.2	39.0	42.6
paj_raygentop_ hierarchy_no_mem	21.4	17.7	37.3	42.5
pdc	23.3	17.7	29.6	33.8
rs_decoder_2	12.1	10.3	16.5	20.3
s38417	8.2	6.7	12.8	13.6
spla	12.2	9.1	16.8	20.4
geomean	15.3	12.4	22.4	25.1
speed-up	1.00	1.22	0.68	0.61

TABLE IV
RUN-TIMES RESULTS FOR FINE-GRAINED APPROACH INTEL CORE I5.

	Number of Threads			
Benchmark	**1**	**2**	**3**	**4**
cf_cordic_v_18_18_18	10.1	8.7	7.	8.4
cf_fir_24_16_16	44.0	43.2	48.5	58.7
clma	24.8	21.8	21.9	24.0
des_perf	8.8	9.2	7.3	9.0
ex1010	14.8	12.3	13.1	14.6
frisc	10.2	12.3	10.2	9.8
mac2	26.9	25.7	23.8	26.7
paj_raygentop_ hierarchy_no_mem	22.4	20.6	23.9	29.0
pdc	23.4	19.3	19.5	21.1
rs_decoder_2	12.7	11.7	10.4	13.4
s38417	8.8	8.4	8.1	8.7
spla	13.4	10.4	12.8	11.3
geomean	16.1	14.9	14.6	16.2
speed-up	1.00	1.08	1.11	1.00

ABC [19], then clustered using T-Vpack [13] into logic blocks with 10 4-LUTs and 22 inputs. The FPGA routing architecture targeted contains unidirectional wire segments that span 2 logic block tiles. Across all runs, each circuit was routed using a fixed channel width of 1.3× the minimum channel width needed to route the circuit in the single-core case. 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 were included in our experiments; smaller circuits were excluded. The run-times presented correspond to the time spent routing all signals in the PathFinder algorithm, which represents 86% of total router run-time, on average. The remaining 14% of router time includes loading the benchmark circuit, building the routing resource graph device model, and final post-routing timing analysis.

Table I shows the run-time (in seconds) as a function of the number of VPR instances (processes) for the coarse-grained parallel routing approach. The first column of the table gives the name of each circuit. The next five columns present the run-time needed to route each circuit with a given number of processes (processor cores). The 2×4 case corresponds to using two Core 2 Quad processors *across a network* – this column is for comparison only and is not indicative of scalability. Each data point in the table is an average over 5 runs. We reinforce that for a given number of processes, our router produces the same routing results from run-to-run – it is deterministic. The second last row of the table provides the mean run-time across all circuits for a particular number of processes. The last row gives the speed-up relative to serial (non-parallel) routing. Speed-ups of 1.5×, 1.7×, 2.1× are achieved with 2, 3 and 4 processor cores, 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 similar speed-up of 2.2× using 4 cores in their recent parallel placer work [2]. Results are also provided for an Intel Core i5 processor system in Table II. Similar speed-ups are achieved on the Core i5. The Core i5 is a stand-alone (non-networked) machine, and hence 2×4 results are not shown for this processor.

Turning to the results for the fine-grained parallel router, Table III shows run-times as a function of the number of threads used. A speed-up of 1.2× is achieved with two threads; slow downs are observed with more than two threads. The data shows that the suitability of the fine-grained approach depends on the memory architecture of the computer on which routing is executed: In the Core 2 Quad, pairs of cores share an L2 cache. When two threads are used, the threads can communicate with one another through the shared cache. However, when three or four threads are used, communication between some threads must happen through main memory (there is no L3 cache), eliminating the chance for speed-up.

Table IV shows results for the fine-grained approach using the Core i5 system. The Core i5 has a unified L3 cache shared among all four processors, but cores do not share an L2 cache. The results indicate a less significant speed-up than the Core 2 Quad for two threads (because L3 cache is used instead of L2 cache), but better speed-up for more than two threads, because threads on the Core i5 do not have to communicate through main memory. Nevertheless, the speed-up results for fine-grained parallel routing are poor in comparison with those for coarse-grained parallel routing. Coarse-grained parallel routing appears to be the superior parallelization approach. New computer architectures from both Intel [10] and AMD [4] that feature point-to-point connections between cores may provide better performance in the fine-grained case.

Table V shows the run-time results using a *combination* of

TABLE V
RUN-TIMES FOR COMBINED COARSE AND FINE-GRAINED APPROACH. *2 QUAD CORE SYSTEMS ACROSS A NETWORK.

	Number of Processes/ Threads (coarse, fine)		
Benchmark	**1,1**	**2,2**	**4,2***
cf_cordic_v_18_18_18	8.2	5.7	4.0
cf_fir_24_16_16	39.7	25.0	15.4
clma	23.5	13.2	13.8
des_perf	9.2	6.0	3.8
ex1010	13.9	7.5	5.3
frisc	10.2	5.1	5.3
mac2	26.8	16.7	15.8
paj_raygentop_ hierarchy_no_mem	21.4	14.4	13.3
pdc	23.3	10.5	7.5
rs_decoder_2	12.1	6.7	4.4
s38417	8.2	4.3	3.6
spla	12.1	6.6	7.1
geomean	15.3	8.8	7.1
speed-up	1.00	1.74	2.16

TABLE VI
NUMBER OF WIRE SEGMENTS FOR COARSE-GRAINED PARALLEL ROUTING. *2 QUAD CORE SYSTEMS ACROSS A NETWORK.

	Number of Processes				
Benchmark	**1**	**2**	**3**	**4**	**2×4***
cf_cordic_v_18_18_18	19306	19451	19386	19521	19508
cf_fir_24_16_16	19776	19741	19633	19736	19659
clma	38771	39196	39006	39013	38955
des_perf	22931	22819	22972	22994	22932
ex1010	23349	23462	23691	23734	23410
frisc	14485	14490	14684	14673	14691
mac2	42435	42731	42930	43110	43405
paj_raygentop_ hierarchy_no_mem	19144	19066	19079	18927	19107
pdc	27876	27688	27810	27822	27489
rs_decoder_2	11143	11218	11238	11265	11169
s38417	17132	17105	17217	17138	17227
spla	17684	17767	17849	17733	17876
geomean	21311	21352	21421	21425	21403
relative to serial	1.000	1.002	1.005	1.005	1.004

coarse and fine-grained parallel routing. In particular, we use multiple VPR instances that communicate with MPI (coarse-grained approach), where the router in each VPR instance is multi-threaded (fine-grained approach). The (2,2) column corresponds to two parallel VPR instances (processes), each of which is dual-threaded – 4 cores are used in total. The (4,2) column corresponds to four parallel VPR instances (processes), each of which is dual-threaded – 8 cores are used in total. Mean speed-ups of 1.7× and 2.2× are realized in the (2,2) and (4,2) scenarios, respectively. It is interesting to compare the (4,2) results with the 2×4 results in Table I, as they both use the same number of processor cores. Using 8 VPR instances provides slightly better results, however, we believe that depending on the memory architecture of the processor on which routing is executed, using a combination of fine and coarse-grained parallelism could offer the best speed-up. Perhaps in the future we will see programs where the approach to parallelization is chosen dynamically, depending on the processor being used.

In our coarse-grained parallel routing approach, the VPR instances executing concurrently operate with slightly stale congestion information, and consequently, it is conceivable that quality-of-result could be adversely impacted. In our fine-grained approach, we expect no quality degradation, however, the routing results are changed as a consequence of splitting the router's priority queue into N queues – nodes with the same cost that reside in different priority queues may be drawn in a different order. 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.

Tables VI, VII and VIII show the number of used wire segments for coarse-grained, fine-grained and combined coarse/fine-grained parallel routing, respectively. The last row of each of these tables gives change in the number of wire segments versus using serial (non-parallel) routing. Observe that in all cases, the average change to the number of used wire segments is less than 0.5%. Tables IX, X and XI provide critical path delay results for coarse, fine, and combined coarse/fine-grained parallel routing. We actually observe *improved* critical path delay in some cases (numbers < 1), however, we believe this is due to noise in the routing algorithm, which is heuristic. Looking at the last row in each of the critical path delay tables, we see that critical path delay was never worsened by more than 1%, on average. From the wire segment usage and critical path delay results, we conclude that the approaches to parallelization do not significantly impact the router's quality-of-result.

TABLE VII
NUMBER OF WIRE SEGMENTS FOR FINE-GRAINED PARALLEL ROUTING.

	Number of Threads			
Benchmark	**1**	**2**	**3**	**4**
cf_cordic_v_18_18_18	19306	19339	19386	19230
cf_fir_24_16_16	19776	19802	19863	19840
clma	38771	38665	38743	38697
des_perf	22931	22965	23157	22844
ex1010	23349	23208	23192	23028
frisc	14485	14484	14501	14422
mac2	42435	42315	42426	42278
paj_raygentop_ hierarchy_no_mem	19144	18988	19262	18948
pdc	27876	27893	27946	27705
rs_decoder_2	11143	11179	11196	11221
s38417	17132	17186	17201	17085
spla	17684	17659	17636	17730
geomean	21311	21294	21358	21244
relative to serial	1.000	0.999	1.002	0.997

TABLE VIII
NUMBER OF WIRE SEGMENTS USING COMBINED COARSE AND FINE-GRAINED PARALLEL ROUTING. *2 QUAD CORE SYSTEMS ACROSS A NETWORK.

	Number of Processes/ Threads (coarse, fine)		
Test scenario	**1,1**	**2,2**	**4,2***
cf_cordic_v_18_18_18	19306	19405	19494
cf_fir_24_16_16	19776	19729	19746
clma	38771	38940	38933
des_perf	22931	23065	22918
ex1010	23349	23456	23345
frisc	14485	14548	14632
mac2	42435	42942	43129
paj_raygentop_ hierarchy_no_mem	19144	19045	19010
pdc	27876	27831	27945
rs_decoder_2	11143	11157	11300
s38417	17132	17132	17262
spla	17684	17877	17872
geomean	21311	21381	21428
relative to serial	1.000	1.003	1.005

V. CONCLUSIONS AND FUTURE WORK

Parallel computing is a promising avenue for reducing the run-time of FPGA CAD tools. In this paper, we presented two approaches for deterministic parallel FPGA routing. In the coarse-grained approach, processor cores route different signals concurrently and communicate with one another using MPI. In the fine-grained approach, the maze router expansion for an individual pin is accelerated using threads. The coarse and fine-grained approaches can be used in tandem. Results show that the coarse-grained approach provides $2.1\times$ speed-up using 4 processor cores. Neither of the proposed techniques impact quality-of-result. We further showed that the memory architecture of the processor on which the router executes can significantly affect the results achieved and we expect

TABLE IX
CRITICAL PATH DELAY (NS) FOR COARSE-GRAINED PARALLEL ROUTING.

	Number of Processes				
Benchmark	**1**	**2**	**3**	**4**	**2×4***
cf_cordic_v_18_18_18	5.949	5.949	5.949	5.949	5.949
cf_fir_24_16_16	12.779	12.780	12.783	12.781	12.780
clma	12.529	12.531	12.529	12.637	12.534
des_perf	6.173	6.170	6.179	6.179	6.174
ex1010	9.605	9.711	9.607	9.801	9.820
frisc	11.037	10.828	10.819	10.828	10.829
mac2	30.936	30.834	30.833	30.833	30.727
paj_raygentop_ hierarchy_no_mem	11.115	11.219	11.114	11.115	11.530
pdc	10.455	9.895	9.887	9.678	10.201
rs_decoder_2	19.116	18.985	19.406	19.203	19.101
s38417	7.049	7.047	7.047	7.047	7.129
spla	7.681	8.100	8.505	8.192	8.363
geomean	10.726	10.716	10.761	10.725	10.820
relative to serial	1.000	0.999	1.003	1.000	1.009

TABLE X
CRITICAL PATH DELAY (NS) FOR FINE-GRAINED PARALLEL ROUTING.

	Number of Threads			
Benchmark	**1**	**2**	**3**	**4**
cf_cordic_v_18_18_18	5.949	5.949	5.949	5.949
cf_fir_24_16_16	12.779	12.780	12.780	12.780
clma	12.529	12.525	12.530	12.637
des_perf	6.173	6.177	6.177	6.069
ex1010	9.605	9.605	9.603	9.603
frisc	11.037	10.935	10.829	10.935
mac2	30.936	30.833	30.833	30.828
paj_raygentop_ hierarchy_no_mem	11.115	11.110	11.114	11.115
pdc	10.455	9.578	9.203	10.290
rs_decoder_2	19.116	18.796	18.796	19.223
s38417	7.049	7.044	7.049	7.049
spla	7.681	8.047	7.676	7.781
geomean	10.726	10.662	10.578	10.710
relative to serial	1.000	0.994	0.986	0.998

that both parallelization techniques may benefit from new processor architectures that feature improved point-to-point communication between cores.

A direction for future work relates to how signals are partitioned in the coarse-grained parallel approach. We can partition signals into sets based on pin locations to reduce the likelihood that signals in different sets would short with one another (i.e. geographically-based partitioning). In so doing, we expect that less frequent update messages between processes would be needed, reducing overhead and improving speed-up. Moreover, each process could work on a reduced-size routing graph, lowering peak memory requirements. A second future direction is to modify our algorithms to be *serially equivalent*. Specifically, while our approaches are deterministic/repeatable for a given number of processor cores, the results achieved differ versus the single-core (serial) case. We plan to explore techniques for serial equivalence and their impact on run-time.

ACKNOWLEDGEMENTS

The authors thank Scott Chin for suggesting that node expansion might be parallelized on a fine-grain level.

REFERENCES

[1] Open MPI: Open source high performance computing. *http://www.open-mpi.org/*, 2010.

[2] V. Betz, A. Ludwin, and K. Padalia. High-quality, determinstic parallel placement for FPGAs on commodity hardware. In *ACM Int'l Symp. on FPGAs*, pages 14–23, 2008.

TABLE XI
CRITICAL PATH DELAY (NS) FOR COMBINED COARSE AND FINE-GRAINED PARALLEL ROUTING. *2 QUAD CORE SYSTEMS ACROSS A NETWORK.

	Number of Processes/ Threads (coarse, fine)		
Test scenario	**1,1**	**2,2**	**4,2***
cf_cordic_v_18_18_18	5.949	5.949	5.949
cf_fir_24_16_16	12.779	12.779	12.779
clma	12.529	12.531	12.534
des_perf	6.173	6.066	6.072
ex1010	9.605	9.605	9.718
frisc	11.037	10.936	10.932
mac2	30.936	30.833	30.729
paj_raygentop_ hierarchy_no_mem	11.115	11.425	11.114
pdc	10.455	9.484	9.764
rs_decoder_2	19.116	18.997	19.173
s38417	7.049	7.047	7.047
spla	7.681	7.749	8.257
geomean	10.726	10.639	10.714
relative to serial	1.000	0.992	0.999

[3] Altera Corp., San Jose, CA. *Stratix-III FPGA Family Data Sheet*, 2008.

[4] AMD. Magny-cours and direct connect architecture 2.0, 2009. `http://developer.amd.com/documentation/articles/pages/Magny-Cours-Direct-Connect-Architecture-2.0.aspx`.

[5] L. Cabral, J. Aude, and N. Maculan. TDR: A distributed-memory parallel routing algorithm for FPGAs. In *Int'l Conf. on Field Programmable Logic and Applications*, pages 263–270, 2002.

[6] P. Chan and M. Schlag. New parallelization and convergence results for NC: a negotiation-based FPGA router. In *ACM Int'l Symp. on FPGAs*, pages 165–174, 2000.

[7] P. K. Chan and M. D. F. Schlag. Acceleration of an FPGA router. In Proc. of FCCM, page 175, 1997.

[8] E. Cohen. Using selective path-doubling for parallel shortest-path computations. *J. Algorithms*, 22(1):30–56, 1997.

[9] J. Driscoll, H. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11), 1988.

[10] Intel. Intel quickpath architecture, 2008. `www.intel.com/technology/quickpath/whitepaper.pdf`.

[11] C.Y. Lee. An algorithm for path connections and its applications. *IRE Trans. on Electronic Computers*, EC-10(2):364–365, 1961.

[12] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, M. Fang, and J. Rose. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *ACM/SIGDA Int'l Symp. on FPGAs*, pages 133–142, 2009.

[13] A. Marquardt, V. Betz, and J. Rose. Using cluster based logic blocks and timing-driven packing to improve FPGA speed and density. In *Int'l Sym. on FPGAs*, pages 37–46, Monterey, CA, 1999.

[14] L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *ACM/SIGDA Int'l Symp. on FPGAs*, pages 111–117, 1995.

[15] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *Proc. 6th Int'l Euro-Par Conf. on Parallel Processing*, pages 461–470, 2000.

[16] W. Chow, R. Fung, and V. Betz. Simultaneous short-path and long-path timing optimization for FPGAs. In *IEEE Int'l Conf. on Computer Aided Design*, pages 838–845, 2004.

[17] L. Farragher, Q. Wang, S. Gupta, and J. Anderson. CAD techniques for power optimization in Virtex-5 FPGAs. In *IEEE Custom Integrated Circuits Conf.*, pages 85–88, 2007.

[18] J. Swartz, V. Betz, and J. Rose. A fast routability-driven router for FPGAs. In *ACM/SIGDA Int'l Symp. on FPGAs*, pages 140–149, Monterey, CA, 1998.

[19] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, Release 70930. `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[20] E. Lusk, W. Gropp, and R. Thakur. *Using MPI-2*. MIT Press, Cambridge, MA, 1999.

[21] Xilinx Inc., San Jose, CA. *XC4000 FPGA Data Sheet*, 1999.

[22] Xilinx Inc., San Jose, CA. *Virtex-5 FPGA Data Sheet*, 2007.