Parallel FPGA Routing based on the Operator Formulation

Yehdih Ould Mohammed Moctar, Philip Brisk
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
{moctar, philip}@cs.ucr.edu

ABSTRACT
We have implemented an FPGA routing algorithm on a shared memory multi-processor using the Galois API, which offers speculative parallelism in software. The router is a parallel implementation of PathFinder, which is the basis for most commercial FPGA routers. We parallelize the maze expansion step for each net, while routing nets sequentially to limit the amount of rollback that would likely occur due to misspeculation. Our implementation relies on non-blocking priority queues, which use software transactional memory (STM), to identify the best route for each net. Our experimental results demonstrate scalability for large benchmarks and that the amount of available parallelism depends primarily on the circuit size, not the interdependence of signals. We achieve an average speedup of approximately 3x compared to the most recently published work on parallel multi-threaded FPGA routing, and up to 6x in comparison to the single-threaded router implemented in the publicly available Versatile Place and Route (VPR) framework.

Categories and Subject Descriptors
B.7.1 [Integrated Circuits]: Types and Design Style – gate arrays; B.7.2 [Integrated Circuits]: Design Aids – placement and routing; D.1.3 [Programming Techniques]: Concurrent programming – parallel programming.

General Terms
Algorithms, Performance, Design.

Keywords
Field Programmable Gate Array (FPGA), Routing, Routing Resource Graph (RRG), Maze Expansion, Irregular Algorithm, Software Transactional Memory (STM).

1. INTRODUCTION
Routing is possibly the most time consuming task in CAD flows targeting FPGAs; computing a legal route is equivalent to the NP-complete problem of finding a set of disjoint paths in a graph (Fig. 1). Parallelizing an FPGA router could yield tremendous productivity benefits, as the process of synthesizing an industrial-scale circuit on a high-capacity commercial FPGA can easily take hours, days or even weeks, depending on the size of the circuit and the target device.


Figure. 1. An instance of the disjoint path problem: (a) a graph G(V, E) with sources S = {s₁, s₂} and sinks T = {t₁, t₂}; (b) an illegal solution, i.e., two non-disjoint paths that share a common vertex; and (c) a legal solution, i.e., two disjoint paths that share no common vertices.

Commercial FPGA CAD tools use a variant of the PathFinder negotiation congestion algorithm for routing [11]. At the lowest level of operation, PathFinder invokes a maze expansion step to explore the nodes of the routing resource graph (RRG) to route individual signals. During maze expansion, nets may share routing resources, a temporarily illegal solution. To legalize the result, PathFinder iteratively rips up and re-routes nets that share routing resources, and imposes a penalty cost on the shared resources to dissuade their use in subsequent iterations; this pushes the algorithm toward convergence. PathFinder fails if a legal solution is not discovered within a user-specified number of iterations.

Maze expansion, a directed breadth-first or A* search of the RRG, dominates PathFinder’s runtime. It is an irregular algorithm, in the sense that it operates on a sparse graph, typically implemented using a linked data structure. Irregular algorithms do exhibit significant parallelism, but are difficult to parallelize statically [14] because the amount of parallelism depends on the content of the data structure (e.g., graph topology) as well as the operations performed on the elements of the data structure at runtime.

Maze expansion is a perfect example of a highly parallel irregular algorithm; it can explore many RRG vertices independently, albeit, with restrictions: namely, parallel operations cannot be applied to two adjacent vertices at the same time. Thus, the amount of parallelism that maze expansion can exploit depends on the RRG’s sparseness; fortunately, RRGs are sparse in practice.

This paper describes a parallel PathFinder implementation using the open source Galois framework [12, 14]. Galois’ programming model, compiler, and runtime synergistically accelerate irregular algorithms that dynamically modify linked-based data structures. Our implementation parallelizes maze expansion using Galois in a shared memory multi-processor platform. It scales well on large benchmarks, and achieves near-linear speedups as the number of cores varies from 1 to 8 (one thread per core) without degrading solution quality. Based on this experience, we strongly believe that Galois’ approach is the right solution for parallel CAD, due to the widespread use of graph-based data structures (e.g., netlists) that exhibit irregular parallelism.

The technical contributions of this paper are as follows:
• The application of speculative parallelization with non-blocking priority queues implemented using software transactional memory (STM) to the maze expansion step of PathFinder.
• An implementation of the parallel router in the open source, publicly available Versatile Place and Route (VPR) [9, 10, 15] simulator for FPGA architecture and CAD research.
• A favorable experimental evaluation that demonstrates the efficacy of our parallelization scheme and its implementation.

2. BACKGROUND & TERMINOLOGY
We assume that the reader is familiar FPGA architectures and CAD (technology mapping, packing, placement, routing), and understands the context in which routing is performed (i.e., after placement). We target the VPR FPGA architecture [9, 10, 15].

2.1 FPGA Routing
The Routing Resource Graph (RRG) \( G = (V, E) \) is the primary data structure that represents the routing resources of the target device [2], as discussed in the preceding section. Each vertex \( v \in V \) represents a wire or pin, and each edge \( e \in E \) represents a switch or other feasible connection between two vertices.

Each net \( N_i = (s_i, t_{i,1}, t_{i,2}, \ldots, t_{i,k}) \) is a signal to route through \( G \). \( N_i \) emanates from a single source \( s_i \in V \) and must connect to a set of sinks \( t_{i,1}, t_{i,2}, \ldots, t_{i,k} \in V \). Let \( P_{ij} \) denote the path from \( s_i \) to sink \( t_{ij} \) in \( G \). Two paths \( P_{ij} \) and \( P_{kl} \) that emanate from the same source may overlap; however, paths emanating from two different sources of two different nets must be disjoint, as shown in Fig. 1(c). The routing tree for net \( N_i \), denoted \( RT(N_i) \), contains the set of paths from the source to all of the sinks.

We parallelized VPR’s routability-driven [19] router, which associates a cost \( c(v) \) with each RRG vertex. Its objective is to minimize the total cost of the vertices belonging to the routing trees for all nets. Other objectives (e.g., timing-driven routers) are also possible, but are beyond the scope of this work.

2.2 PathFinder Algorithm
This section summarizes the aspects of VPR’s routability-driven PathFinder implementation that are relevant to our parallelization scheme [19]. PathFinder is a triple-nested loop [11]: the outer loop is called the global router; the middle loop is called the signal router; and the inner loop is maze expansion. Our parallelization effort focuses on maze expansion.

Global Router: The global router repeatedly invokes the signal router to route all of the nets. It terminates when a legal routing solution is found, or after a fixed number of iterations fail.

Signal Router: Each signal router iteration rips up each net and re-routes it by invoking maze expansion.

Maze Expansion: For net \( N_i \), maze expansion computes a path from the source to each sink in the RRG. All of the RRG vertices that have been uncovered are stored in a priority queue (PQ) based on their cost. Maze expansion extracts the minimum cost vertex \( v_{\text{min}} \) from PQ. If \( v_{\text{min}} \) is a sink, then a backtrack procedure is invoked to construct a path from \( u \) to \( RT(N_i) \), which is the routing tree for \( N_i \); otherwise, each neighbor \( v \) of \( v_{\text{min}} \), which has not previously been discovered, is inserted into PQ and the maze expansion continues. Fig. 2 shows an example.

The path cost of vertex \( v \) is the sum of the vertex costs on the path from source \( s_i \) to \( v \) as uncovered by maze expansion:

\[
\text{PathCost}(v) = c(v) + \text{PathCost}(v_{\text{min}}).
\]  

(1)

When \( v \) is inserted into PQ, \( \text{PathCost}(v) \) is used as its priority.

![Figure 2. Maze expansion: (a) PQ contains vertices that have been discovered, but have not yet had their neighborhoods expanded; \( RT(N) \) contains the portions of \( N_i \)'s routing tree that have been found thus far. (b) A vertex is selected for neighborhood expansion; its neighbors that have not yet been discovered are inserted into PQ, expanding the “wavefront” of the search. (c) Expanding the neighborhood of the next vertex discovers a sink. (d) A backtrack adds the vertices and edges along the backtracktrace to \( RT(N_i) \), creating a new path to the sink.](image)

3. GALOIS
This section introduces the Galois programming model [12, 14].

Philosophy and Implementation: Galois employs a data-centric approach to irregular algorithm development called the operator formulation. In a graph, active elements are the vertices and/or edges where computation could be performed through the application of an operator. The neighborhood of an activity is the set of vertices and edges that the activity reads or writes.

In Fig. 2, the active vertices are those in PQ; each neighborhood is the set of adjacent vertices to each active vertex, and the operator applied is the neighborhood expansion in which newly discovered adjacent vertices are inserted into PQ (Fig. 2(b)) and sinks may be discovered (Fig. 2(c)). When a sink is discovered, the backtrack process involves a different set of active elements, neighborhood definition, and operator to update the routing tree \( RT(N_i) \).

The operator formulation naturally lends itself to amorphous data parallelism, which permits parallel processing of active vertices, limited by algorithm-dependent neighborhood and ordering constraints. Executing one activity may create others dynamically; e.g., applying the neighborhood operation to a vertex may create new active vertices to insert into PQ. Activities are allowed to modify the graph; the itself RRG is not modified, but the routing tree \( RT(N_i) \) is constructed incrementally, one path at a time.

Conflicting activities cannot execute concurrently. For example, consider two vertices \( u \) and \( v \) that share a common neighbor, \( w \). If both \( u \) and \( v \) expand their neighborhoods concurrently, then each expansion will discover and add \( w \) to PQ with different path costs. This type of conflict must be avoided. Activities that do not modify their neighborhoods can always execute in parallel.

Galois uses locks to ensure that only activities with disjoint neighborhoods execute in parallel. Each graph element has an exclusive lock that must be acquired by a thread before it can access that element. Locks are held until the activity terminates.
If a lock cannot be acquired because it is owned by another thread, the Galois runtime detects the conflict and rolls back one of the conflicting activities. Lock manipulation is performed entirely by the methods in the graph class. To enable rollback, each API method that modifies the graph makes a copy of the data before modification, similar to transactional memory systems. This copy, called an undo log, supports rollback in the case of misspeculation, and is discarded whenever an activity successfully commits. When an activity aborts, all computation performed up to that point is lost; the Galois runtime system takes corrective action to roll back the activity and re-execute it after all other conflicting activities complete.

**Summary:** Galois implements speculative parallelism in a manner that hides the complex underlying details from the programmer. Galois introduces new syntactic constructs that enable the programmer to clearly express algorithms in terms of the operator formulation applied to elements in ordered or unordered sets. Galois provides an API of concurrent data structures, which are challenging to implement; this simplifies application development for the programmer. The Galois runtime automatically detects and rectifies conflicts that cannot be discovered statically. In principle, using Galois is much simpler than requiring the programmer to implement these mechanisms every time he or she parallelizes a new irregular application.

### 3.1 Bottlenecks
The major sources of runtime overhead in Galois are as follows:

**Dynamic assignment of work:** Threads obtain work from a centralized workset. This requires synchronization and leads to challenges in terms of load balancing. If each activity requires minimal computation, then the overhead of synchronization and contention becomes a bottleneck.

**Undo log:** When an activity modifies an element of a graph (or other data structure), a copy of the element is stored in an “undo” log, which enables rollbacks in the case of misspeculation. The time spent creating and maintaining these copies is non-trivial.

**Aborted activities:** When an activity aborts (e.g., due to misspeculation), the computational work performed up to that point is wasted; the Galois runtime system takes corrective action to roll back the activity.

To reduce overhead, Galois provides three optimization techniques that we leverage in our PathFinder implementation.

### 3.2 Optimizations using Galois

**Cautionous operators:** A cautious operator reads all elements of its neighborhood before modifying any of them; the reading phase acquires all of the locks. If lock acquisition is successful, then the operator is guaranteed to compete without conflicting with other transactions. In this case, the undo list can be safely discarded. Additionally, Galois’ internal conflict management for the cautious operator can be suppressed since it only accesses elements for which it has already obtained a lock.

**One-shot operator implementations:** It is often possible to predict the neighborhood of an activity without performing any computation, or to compute fairly tight over-approximations. In a one-shot implementation, the neighborhood elements are never read, so the locks can be released once successful completion is guaranteed; in contrast, a cautious operator must hold onto the locks while its activity commences, even after guaranteeing completion. Releasing locks early enables greater concurrency.

**Iteration coalescing:** Iteration coalescing allows one thread to process multiple iterations at once, breaking the one-to-one correspondence between iterations and activities. Galois provides each thread with a local workset. Any activity that generates new active elements places them in the thread’s local workset, as opposed to the global workset, which is accessed by all threads. When an activity completes, the iteration grabs work from its local workset if possible without releasing any abstract locks. This continues until the local workset is empty, a conflict is detected, or the maximum coalescing factor is reached. Each iteration releases all of its abstract locks when it finishes. If a conflict occurs, the currently executing activity aborts, but all prior completed activities that were coalesced into the same iteration commit; the activities in the local workset are then moved to the global workset, where other threads may execute them.

4. PARALLEL PATHFINDER IN GALOIS

Fig. 3 illustrates our parallel implementation of maze expansion in Galois. Prior work has shown that maze expansion accounts for approximately 68% of the total runtime [5]. Our expectation was that Galois could identify a large number of non-conflicting operations, enabling parallel execution of a large number of active vertices. Fig. 3 illustrates iteration coalescing: each thread has its own local priority queue (LPQ) while a global priority queue (GPQ) is stored in shared memory, along with the RRG, a set of routing trees for each net, and an array, VCost, which contains information relating to the cost \( c(v) \) of each RRG vertex; \( c(v) \) depends on several other cost terms that contribute to \( \text{PathCost}(v) \) per Eq.(1) [11, 19]. Each \( \text{VCost} \) entry is a struct that holds these cost values, which are stored separately from the RRG in shared memory to reduce contention for locks. Each thread repeatedly accesses its LPQ to obtain another vertex to expand. If the LPQ is empty, then the access is forwarded to the GPQ. The maze expansion process stops when all sinks are found.

![Figure 3. Multi-threaded parallelization strategy for the Galois implementation of PathFinder. Solid lines indicate control flow within each thread; dashed lines indicate the data structures accessed by each operation.](image-url)
4.1 Maze Expansion Operators

The neighborhood expansion operator is cautious. Each thread picks an active vertex \( v_{\text{act}} \) from its LQ. The operator identifies a set \( S \) of vertices adjacent to \( v_{\text{act}} \) that have not yet been discovered. Galois acquires locks for each vertex \( v \in S \). The operator inserts \( v \) into LQ with \( \text{PathCost}(v) \) as its priority.

The Backtrace operator, which is called if \( v \) is a sink, is one-shot. It creates a path from \( v \) to the routing tree for the current net, and sets \( \text{PathCost}(u) \) to zero for each vertex \( u \) on the path. If any neighbor of \( u \) is locked, then the corresponding activity may access \( u \). If so, then it is better to keep \( u \) locked while updating \( \text{PathCost}(u) \). If none of \( u \)’s neighbors are locked, then it is safe to release the lock and update \( \text{PathCost}(u) \).

4.2 Priority Queue Implementation

Although Galois provides a concurrent PQ, we discovered that its functionality was limited and its performance was a bottleneck. Galois’s PQ is integer-based and can only store the ID number of a vertex; any additional information needs to be mapped to another data structure. Additionally, each thread that tries to insert or remove a vertex from the priority queue must acquire a lock; with a large number of concurrently executing threads, the lock acquisition process becomes a performance bottleneck.

To improve performance, we implemented a non-blocking priority queue based on software transactional memory (STM) [4, 17], which deviates from the Galois model. The STM-based PQ declares the access functions as atomic. The underlying implementation is a binary heap, similar to VPR’s non-concurrent PQ. Insertion and extract operations are executed as transactions.

5. EXPERIMENTAL SETUP

5.1 VPR Implementation in Galois

We ported VPR 5.0 into the Galois system. We modified all of the data structures used by the router to be thread-safe and compatible with Galois. We rewrote the router to be compliant with the operator formalism [14], which is a central requirement of Galois. We implemented the RRG using Galois’ graph model, and the STM-based non-blocking PQ as discussed in Section 4.2.

5.2 VPR Architectural Parameters

VPR generates an FPGA architecture from a set of parameters [2], whose dimensions are approximately equal to the per-benchmark resource requirements. Table 1 lists the parameters we used.

5.3 Benchmarks

We selected 10 of the largest IWLS benchmarks [6] for use in our experiments. Table 2 summarizes them, including the size of the FPGA generated by VPR, the number of nets, and the number of configurable logic blocks (CLBs) used.

VPR repeatedly routes each benchmark using a binary search to identify the smallest channel width, \( W_{\text{min}} \), for which a legal route can be found. VPR also allows the user to specify a chosen channel width \( W \), and VPR will try its best to find a legal route, but may fail. We took that latter approach in our experiments: first, we compute \( W_{\text{min}} \) (using VPR’s serial implementation of PathFinder) and set \( W = 1.4W_{\text{min}} \) for each benchmark.

Table 1. FPGA architectural parameters, taken from the publicly available IFAR repository [7, 8]; we assume 65nm CMOS (BPTM).

<table>
<thead>
<tr>
<th>K</th>
<th>N</th>
<th>W</th>
<th>I</th>
<th>F_{\text{in}}</th>
<th>F_{\text{out}}</th>
<th>CLB Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>1.4W_{\text{min}}</td>
<td>33</td>
<td>0.15</td>
<td>0.1</td>
<td>8069.46</td>
</tr>
</tbody>
</table>

Table 2. Benchmark summary.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Dimensions</th>
<th>Nets</th>
<th>CLBs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac_ctrl</td>
<td>48 x 48</td>
<td>5097</td>
<td>5008</td>
</tr>
<tr>
<td>aces_core</td>
<td>33 x 33</td>
<td>5800</td>
<td>2518</td>
</tr>
<tr>
<td>des_area</td>
<td>16 x 16</td>
<td>1569</td>
<td>695</td>
</tr>
<tr>
<td>mem_ctrl</td>
<td>27 x 27</td>
<td>4464</td>
<td>3158</td>
</tr>
<tr>
<td>pci_bridge32</td>
<td>74 x 74</td>
<td>8016</td>
<td>7815</td>
</tr>
<tr>
<td>spi</td>
<td>13 x 13</td>
<td>923</td>
<td>712</td>
</tr>
<tr>
<td>systemcaes</td>
<td>21 x 21</td>
<td>2509</td>
<td>2173</td>
</tr>
<tr>
<td>systemcdes</td>
<td>12 x 12</td>
<td>1068</td>
<td>706</td>
</tr>
<tr>
<td>wb_funct</td>
<td>40 x 40</td>
<td>5154</td>
<td>4429</td>
</tr>
<tr>
<td>wb_conmax</td>
<td>47 x 47</td>
<td>10430</td>
<td>6297</td>
</tr>
</tbody>
</table>

5.4 Synthesis Flow

The IWLS benchmarks are provided in .blif format. To target VPR, we used ABC [1] for logic synthesis and technology mapping, T-VPack1 for placement, and our Galois-compatible VPR implementation for placement and routing; we did not parallelize VPR’s placer, which is based on simulated annealing.

VPR’s routability-driven router [19] is based on PathFinder [11], which terminates if it cannot find a legal route after a user-specified number of iterations. We set the maximum number of iterations allowed to 50. If routing is successful, VPR reports the estimated critical path delay of the circuit.

5.5 Experimental Platform

Our primary experimental objective is to assess the performance and scalability of our parallel implementation of PathFinder on a modern multi-processor. Our experiments were performed on a server featuring 8 Intel Xeon E5540 processors running at 2.53 GHz, with 4 cores per processor and 40 GB shared memory. We ran our router using 1, 2, 4, and 8 threads. Our baseline is the single-threaded VPR 5.0 router, which was implemented in C and does not incur any overhead due to the Galois runtime.

Galois, as presently implemented, imposes several constraints. It supports a maximum of 8 concurrent threads; ideally, we would like to run at least 32 threads on our system. Additionally, the Galois runtime allocates threads to cores automatically; in all cases, it allocated one thread per processor, leaving three cores unused. We were unable to override this decision to experiment with alternative thread allocation policies. In particular, with one thread per processor, we cannot explore the implications of shared cache hierarchies on the router’s performance.

6. EXPERIMENTAL RESULTS

6.1 Routability

Our first concern is routability, i.e., what percentage of nets routed successfully in each experiment? Anything less than 100% would indicate a routing failure. The router successfully routed all of the nets for all benchmarks in all of our experiments.

6.2 Speedup

Fig. 4 reports the speedup attained by increasing the number of threads, normalized to VPR 5.0’s single-threaded execution. On average, Galois achieved a speedup of 1.47x with two threads, 2.99x with four threads, and 5.46x with eight threads. These results indicate favorable scalability up to at least eight threads.

---

1 T-VPack has been deprecated as part of the Verilog-to-Routing (VTR) flow, which expands VPR 6.0 [9, 15]; packing is now integrated into VPR 6.0. The router has not changed significantly from VPR 5.0 to 6.0.
Perfectly linear speedups are not expected, due to the overhead of lock acquisition, aborted activities, and the cost of accessing data structures in shared memory. That being said, these experiments show that maze expansion exhibits ample amorphous parallelism and that a runtime system like Galois can readily extract it.

6.3 Critical Path Delay Variation
The version of Galois that we used to perform our experiments does not guarantee deterministic results, although a deterministic implementation of the Galois runtime has recently been published [13]. We were concerned that varying the number of threads could cause alter the routing results, which could impact the critical path delay. Fig. 5 reports the critical path delays obtained by the router for each benchmark routed by VPR 5.0 and Galois with 1, 2, 4, and 8 threads. A significant difference between the VPR 5.0 results and Galois is observed for several benchmarks; however, differences between Galois using different numbers of threads, per-benchmark, are negligible. This does not imply that Galois obtained identical routing solutions for all benchmarks; it only suggests that all benchmarks achieved similar critical path delays.

6.4 Implementation Choices
The two most important implementation decisions were iteration coalescing and replacing the Galois PQs with STM-based PQs, as described in Sections 3.2 and 4.2. Using 8 threads, we ran our parallel implementation of PathFinder with four configurations: (1) no iteration coalescing with Galois’ PQ; (2) no iteration coalescing and STM PQ; (3) iteration coalescing with Galois’ PQ; and (4) iteration coalescing with the STM PQ. When iteration coalescing is enabled, we use the same PQ implementation (Galois or STM) for both the GPQ and each thread’s LPQ. Fig. 6 reports the speedup of each implementation decision, normalized to Galois running one thread.

Fig. 6 clearly shows that performing both optimizations (iteration coalescing using an STM-based PQ) offers significantly greater speedup compared to enabling one optimization, but not the other (4.38x, on average, compared to 1.38x for using STM-based PQs without iteration coalescing, and 1.87x for enabling iteration coalescing while using Galois’ PQs). These results indicate that we made the correct implementation decisions; one possibility for future work is to consider concurrent PQs other than STM [18].

7. RELATED WORK
In 1997, Chan and Schlag [3] parallelized PathFinder’s signal router on a distributed network of workstations. Using three processors, they achieved a 2.5x speedup. The drawback of their approach is that the results are highly sensitive to the order in which signals are routed. Consider two nets $N_1$ and $N_2$, and assume that $N_2$ would route before $N_1$ in a serial implementation. If so, $N_2$ would read congestion costs of any routing resource used by $N_1$, and may choose a different routing resource as a result; in the parallel implementation, $N_2$ may not read those congestion costs and could therefore make an ill-advised routing decision.

A deterministic scheduler [13] could rectify the issue by imposing a deterministic order on the nets; however, determining the best ordering a-priori appears to be an open problem [16]. Moreover, the overhead of speculatively parallelizing this particular scheme would be quite high. Galois would need to acquire locks for each routing tree $RT(N_i)$ before any signal route could commit; thus, threads would acquire and hold on to locks for a long time, inhibiting parallel execution. This would create large undo lists, and the cost of conflict resolution would be exorbitant, as very large partially-computed routing trees would need to be discarded.

Zhu et al. [20] addressed this concern, but in a limited way. They partition high-fanout nets into sets of low-fanout nets, which are routed individually; low-fanout nets with non-overlapping bounding boxes are routed in parallel because they are unlikely to conflict. Zhu et al. achieved a speedup of 1.9x on a quad-core machine with 2.3% degradation in critical path delay. Our approach using Galois yielded comparable speedups for two cores and higher speedups for four and eight cores, without requiring specialized handling of high-fanout nets; however, our approach parallelized the maze router, rather than the signal router.
Gort and Anderson [5] parallelized the signal router by partitioning the netlist into groups of disjoint subnets, and routed each subnet using independent instances of VPR running on different processors, communicating via MPI; blocking receive calls ensure deterministic results. Each VPR instance routes one signal at a time and then synchronizes with the other instances to update the relevant costs functions before routing the next net. To limit the synchronization overhead, nets to be routed are load-balanced among VPR instances based on fanout and bounding box size. They achieve a speedup of 1.5x on two cores 2.1x on four cores. The synchronization overhead suggests that parallel signal routing may not scale as well as mesh expansion.

Gort and Anderson [5] also parallelized the maze router using pthreads. Each pthread has an LPQ, similar to Galois’ iteration coalescing, along with a GPQ in shared memory. The PQs are lock-based, similar to the PQ provided by Galois. As Fig. 6 indicates, greater speedups can be obtained by using a non-blocking PQ. Unlike our work, their threads update the GPQ after each expansion; ours only access the GPQ when a sink is found or if a thread’s LPQ is empty. Gort and Anderson achieved a speedup of 1.2x with two threads on a quad-core PC; increasing the number of threads yielded slowdowns. In contrast, our results achieve far greater speedups for up to 8 threads using Galois.

8. CONCLUSION

This paper demonstrates that speculative parallelization and the operator formalism, central to Galois’ programming model and philosophy, is the best choice for irregular CAD algorithms that operate on graph-based data structures. These algorithms exhibit significant parallelism, but require a runtime system like Galois to detect and exploit it. The speedups we obtained depend on implementation choices, as we have shown that non-blocking priority queues and iteration coalescing aided us significantly.

Future work will explore the usage of Galois’ non-deterministic scheduler [13], as well as a non-blocking priority queue based on skipped lists [18], rather than software transactional memory. We also plan to implement VPR’s timing-driven router in Galois.

ACKNOWLEDGMENT

Y. Moctar was supported by a DoD SMART fellowship. The authors acknowledge technical assistance with Galois from Donald Nguyen and Keshav Pingali (University of Texas, Austin).

REFERENCES


