Appears in the Proceedings of the 23rd IEEE International Conference on Field-Programmable Technology (FPT), 2024

MultiQueue-Based FPGA Routing: Relaxed A* Priority Ordering for Improved Parallelism

Alexandre Singer,* Hang Yan,* Guozheng Zhang,* Mark C. Jeffrey,* Mirjana Stojilović,§ and Vaughn Betz*
*Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
§School of Computer and Communication Sciences, EPFL, Lausanne, Switzerland
{alex.singer@mail, hang.yan@mail, guozheng.zhang@mail, mcj@ece, vaughn@eecg}.utoronto.ca*, mirjana.stojilovic@epfl.ch§

***Abstract*—Routing is a critical part of the FPGA CAD flow. Its solution quality greatly impacts design speed and power, and its run time significantly contributes to overall compile time. As FPGA design size grows but per-core CPU performance stagnates, parallel FPGA routing algorithms that can leverage multiple compute cores become increasingly valuable. Most prior work in parallel FPGA routing uses *coarse-grained* approaches that route nonoverlapping nets in parallel; this work targets a complementary *fine-grained* form of parallelism in which the shortest-path algorithms that complete a single connection are multi-threaded. We speed up several related shortest path algorithms (Dijkstra's, A*, and directed) by utilizing a concurrency-friendly but weakly ordered data structure, the MultiQueue, and enhance the algorithms to compensate for its imperfect ordering of partial routings. Compared to the VTR 8+ router, these techniques achieve routing time reductions of 18.7× and 13.2× on average over the Titan benchmark suite when using Dijkstra's and A* path search, respectively, on a 12-core CPU. These parallel algorithms achieve wirelength and critical path delay quality comparable to the serial router; they are also deterministic and serially equivalent. When applied to directed search routing, the parallel path search achieves a speed-up of 1.98× and a slightly higher quality than the serial VTR router, but is nondeterministic. Thanks to queue improvements, our router at 1-thread is 1.7× faster than VTR's for Dijkstra's and A* search, but comparable in run time for directed search.**

***Index Terms*—Parallel routing, FPGA, Concurrent priority scheduling, Determinism, Parallel path search**

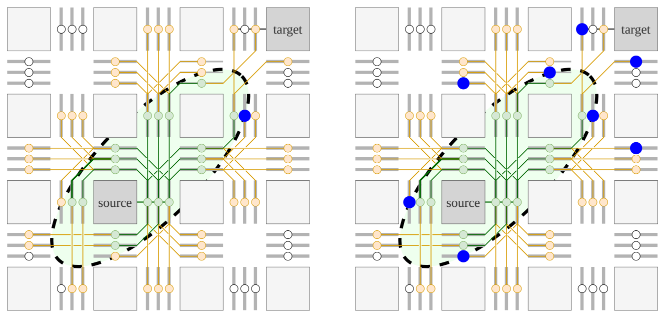
I. Introduction

Field-Programmable Gate Arrays (FPGAs) are seeing adoption in various application domains thanks to their efficiency vs. processor-based solutions and their time-to-market vs. custom computer chips (ASICs) [1]. However, a major barrier to further expansion of FPGA use is their long development cycle compared to software-programmable platforms such as CPUs and GPUs; a major contributor to their lengthy design cycle is the long (often many hours) compile time to implement each iteration of an FPGA design. Research into faster FPGA Computer-Aided Design (CAD) tools has become even more important as process scaling has increased FPGA capacity to millions of logic elements and beyond [2], while single-core CPU performance has nearly stagnated [3].

Routing, which determines the wires and programmable switches each net will use, is one of the most time-consuming steps in the FPGA CAD flow. It has a large impact on the design's maximum operating frequency and the amount of wiring (and hence power) used. State-of-the-art FPGA routers are based on negotiated congestion [4], which repeatedly performs path searches through a large graph (tens to hundreds of millions of nodes) that represents the routing resources of the chip. Despite significant work on incremental approaches to reduce connection re-routing [5], [6] and efficient congestion resolution algorithms [7], [8], finding a legal and performant routing in such a large graph remains time-consuming. Another approach to scaling these algorithms is to divide the work over multiple CPU cores, which is becoming more attractive as the core count of servers continues to increase [9], [10].

Most parallel FPGA routing approaches are *coarse-grained*: they route nets in nonoverlapping FPGA partitions in parallel [11]. In theory, these approaches can extract significant parallelism. However, modern FPGA designs in practice contain high-fanout nets that span much of the device, take a long time to route, and are generally routed serially due to overlap with other nets. We explore a different and complementary *fine-grained* approach that parallelizes the pathfinding process for a single connection.

Fig. 1 illustrates how fine-grained parallel routers can speed up the path search step. Nodes in the routing graph are visited, the costs of their neighbors are calculated, and these neighbors are pushed into a priority queue in a wave that expands from



(a) Serial wave expansion. (b) Parallel wave expansion.

Fig. 1: Serial vs. parallel path-search wave expansion. Explored nodes (routing wires) and the edges (programmable switches) used to reach them are green. Nodes on the expansion frontier (in the queue) and the edges used to reach them are orange. The nodes currently being explored are blue.

the source to the target. A serial wave expansion would only explore one node on the frontier (outlined by a dashed line) at a time. A parallel wave expansion explores multiple nodes on the frontier concurrently to reduce run time. Speeding up the path search at the core of a routing algorithm is highly attractive since this is the largest CPU time consumer in even highly performant algorithms, which use incremental approaches to reduce how often connections are re-routed or coarse-grained parallelism to route nets in parallel.

Parallel path search introduces several challenges. First, the priority queues used to order node exploration typically limit concurrency as multiple threads cannot access the same top-priority element without contention. Concurrency-friendly queues, such as the MultiQueue, can mitigate this bottleneck at the cost of imperfect ordering [12]. Without algorithmic upgrades, this imperfect ordering will lead to suboptimal paths that can hurt solution quality. Finally, a straightforward parallel path search is nondeterministic (i.e., it can return different solutions in different runs), which is undesirable for CAD tools as it complicates debugging and is unacceptable for some security-conscious end users [13].

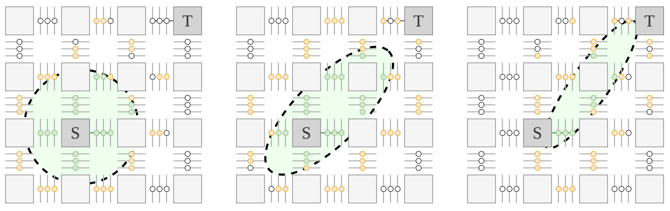
We resolve these challenges via the following contributions:

- We formulate a relaxed version of the A* Shortest Path Algorithm, which is deterministic and serially equivalent regardless of node traversal order (Section III).
- We integrate a fine-grained parallel router into the state-of-the-art routing algorithm in the Verilog to Routing (VTR) project. It is open source so other researchers can build upon it (Section IV).
- We demonstrate a route time speedup of 18.7× and 13.2× on a Dijkstra's and A* path search, respectively, on 12 threads. Even using a directed search, which explores less of the routing graph at the risk of suboptimal paths, we demonstrate a 1.98× speedup on 12 threads (Section V).

II. Background

A. FPGA Routing

The task of FPGA routing is to find an overlap-free set of trees in the routing graph, where each tree is rooted at the source of a net, and its leaves are all the net sinks. The routing graph models the routing resources of the FPGA, where nodes correspond to pins and wires, and edges model the programmable routing switches. The path from a net source to a sink implements a single *connection*. To form a legal routing, the routing trees for different nets must be disjoint (not share any nodes). Routing algorithms seek to minimize the amount of wiring used to reduce power and minimize node and edge delays for timing critical connections to maximize operating frequency [14]. This is a computationally hard problem, so optimal solutions are not feasible for reasonably-sized problems, and heuristics are employed instead. The dominant approach is based on the PathFinder negotiated congestion algorithm [4]. It iteratively routes connections one at a time, applying delay and congestion costs that lead to wiring and timing-efficient routes and force negotiation between connections for in-demand nodes.



(a) Dijkstra's (b) A* (c) Directed

Fig. 2: Wave expansions from source S towards target T.

A key subroutine of PathFinder is the path search, which finds the lowest-cost path from a source node to a sink. Dijkstra's algorithm can be used for this search [15]. It inserts the source node into a priority queue, repeatedly removes the lowest cost node, finds the cost of its neighboring nodes, and inserts them into the priority queue. When the sink (target) is reached, the lowest cost path has been found.

When structural information about the routing graph can be used to predict the remaining cost from a node to the sink, A* algorithms can find the lowest-cost path to the sink while exploring less of the graph [16]. An A* algorithm searches nodes in a priority order, where the priority is the sum of the cost back to the source and the expected cost (heuristic) to the target. To guarantee the lowest-cost path is found, the heuristic must be *admissible*—it must never over-predict the cost to the sink. In FPGA routing, paths that are slightly suboptimal (not guaranteed to be the lowest-cost) are often acceptable, so *inadmissible* heuristics that give a *best guess* of the remaining cost to the sink are often used; these result in directed search algorithms that can explore even less of the graph.

Fig. 2 visualizes the difference between these algorithms. Dijkstra's algorithm does not use a heuristic and hence explores equally in all directions from the source as shown in Fig. 2a. The heuristic of an A* algorithm (Fig. 2b) leads it to preferentially explore nodes closer to the target (sink), reducing the fraction of the graph explored. A directed search algorithm (Fig. 2c) uses an inadmissible heuristic to explore even less of the graph, but may not find the shortest path.

Our work builds on the Adaptive Incremental Router (AIR) of VTR [5], [17]. AIR improves on PathFinder in several ways. First, it reduces the number of times connections are re-routed by only re-routing illegal or timing-suboptimal portions of routing trees. Second, it speeds up high-fanout net routing by only inserting portions of the routing tree of such nets into the priority queue when routing remaining connections. Finally, it computes a heuristic specific to the target FPGA architecture (the *Map Router Lookahead*) via a series of Dijkstra single-source all-destination searches to estimate the minimum cost paths from different types of nodes to destinations a specified distance away. AIR's heuristic is not a guaranteed underestimate (it is inadmissible), so suboptimal paths are tolerated. In practice, AIR often runs in a highly directed mode, scaling up the heuristic to further reduce the graph portion searched.

B. Parallel FPGA Routing

Parallel FPGA routing techniques can be classified into coarse and fine-grained [11]. Coarse-grained routing involves netlist-level partitioning such that the FPGA regions, defined by the spatial locations of net terminals, are disjoint [18], [19], [20], [21], [22]. Hence, routing resources accessed by different threads are independent, which helps determinism and convergence to a legal solution [23]. Coarse-grained routers have drawbacks. First, speed and scalability are limited due to modern FPGA designs having nets spanning large chip areas, often needing serial routing. Second, as the number of threads and netlist partitions increases, more nets cross partitions, impacting parallel scalability. Finally, these routers are highly sensitive to the net routing order [7]. In contrast, fine-grained approaches accelerate the pathfinding for individual source-sink connections without altering the net routing order. Few works have explored fine-grained parallel routing [24], [25].

Gort and Anderson [24] report that 68% of PathFinder's route time (in VPR 5) is spent on maze expansion and propose multithreading to accelerate it. The main thread handles sequential parts and synchronization, expanding a node and signaling helper threads that new nodes are ready for cost evaluation. Threads' only task is to compute the costs of the assigned neighbor nodes in parallel, insert them into local priority queues, and wait at a synchronization barrier for all threads to finish so that the main thread can take the lowest-cost node, insert it into the global queue, and continue the waveform expansion. This method is deterministic; however, it results in only a 1.2× speedup with two threads and slowdowns with more threads. Our approach eliminates frequent barrier synchronization and unleashes more parallelism by allowing threads to perform wavefront expansion in parallel.

Moctar and Brisk [25], [26] accelerate the path search using *speculative* parallelism and the Galois framework [27]. Threads perform waveform expansion in parallel. A thread removes a node from its local priority queue, identifies its neighbors, and attempts to acquire locks for them before proceeding. If a lock cannot be acquired because it is owned by another thread, one of the conflicting activities is rolled back. Unlike Gort and Anderson's router [24], threads access the global queue only when reaching the target node or their local queue is empty. The authors report a 3.67× speedup with eight threads [26], noting that the expensive misspeculation-caused rollbacks and the limited scalability of the deterministic scheduler affect the performance. In comparison, our router is not speculation-based, and the number of locks threads acquire during waveform expansions is greatly reduced. It is difficult to directly compare to this approach, as it uses an older routability-driven router in VTR 5 [26], which lacks key timing optimizations. Our router is built on AIR, the timing-driven router in VTR 8. As an incremental router, AIR performs less work than its predecessors. Consequently, it is several times faster [17] but also leaves less room for parallel scalability.

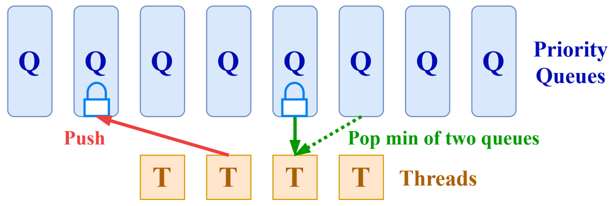


Fig. 3: A MultiQueue with four threads and two queues per thread. Threads access lock-protected queues asynchronously.

C. Concurrent Priority Schedulers

Concurrent Priority Schedulers (CPSs) play a crucial role in parallelizing priority-ordered algorithms such as the path search in FPGA routing. They are often designed for high operation throughput and allow concurrent access across threads. A CPS organizes elements in some programmer-defined priority order and distributes them among threads for execution. For higher scalability, practical CPSs relax the strict priority ordering, dispatching elements in priority order on a best-effort basis. With relaxation, cores execute *some* high-priority elements in parallel, but not necessarily *the* highest-priority element. This trades more parallelism for the cost of reordering elements, potentially causing redundant work. Algorithms that use relaxed scheduling must have the means to tolerate the potential inversions in the priority order, ensuring convergence towards correct or deterministic solutions [28].

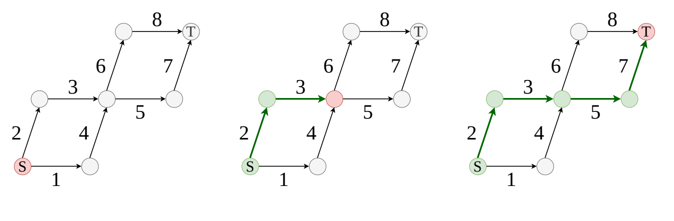
The MultiQueue [12] is a CPS that maintains probabilistic theoretical bounds on the degree of relaxation. Fig. 3 shows a MultiQueue comprising $c \cdot p$ lock-protected priority queues, where p is the number of threads and $c > 1$ is a tuning parameter. It supports two operations: `push` and `pop`. `push` selects a queue at random and inserts an element (i.e., a routing node to explore) to it if its lock is successfully acquired. On the other hand, `pop` first chooses two queues uniformly at random, examines the top elements of both queues and chooses the queue with the higher priority top element (i.e., the lower-cost node). If `pop` acquires the selected queue's lock, it extracts the top element from that queue; otherwise, it loops and selects a new pair of queues. This design ensures that a popped element is, in expectation, in the top $O(cp)$ highest-priority elements in the entire MultiQueue [29].

III. A* with Relaxed Priority Ordering

This work leverages a concurrency-friendly queue to enable parallel A* and related path-search algorithms; however, the relaxed node ordering of this queue introduces challenges in deterministically finding optimal (lowest-cost) paths.

A. Tie-Breaking

In many graphs, multiple optimal paths exist. In a serial implementation of A*, the nodes are always traversed in the same order, so the same path is deterministically returned even when multiple lowest-cost paths exist. However, parallel implementations explore multiple nodes simultaneously. Using the standard A* technique of adding a node to the priority queue the first time it is reached would then lead to nondeterminism as there is a fundamental race in the algorithm.



(a) Four paths (b) Two paths (c) Final path

Fig. 4: Using edge IDs to break ties between multiple equally optimal paths. The red node is being explored and green nodes are on the chosen path to it.

To achieve determinism, one path must always be chosen over all others. In serial A*, this is guaranteed by strict node traversal ordering; the first optimal path found is chosen. For parallel implementations, which do not have a strict node traversal ordering, a tie-breaking condition is needed.

We define the tie-breaking condition as follows: *If a node n is a member of multiple optimal paths, choose the path where the edge entering n is preferred.* Many functions can be used to decide if an edge is preferred over another; however, given two edges, this function must always prefer the same edge. We use edge identifiers (IDs) to break ties, as they are unique in VTR:

$$\textsc{IsPreferred}(e_1, e_2) = \text{ID}(e_1) < \text{ID}(e_2).$$

Since it is unknown which path will ultimately be optimal, ties are broken whenever a new path to n is found.

Fig. 4 illustrates how the above function can break ties and produce a unique optimal path. Fig. 4a shows an example sub-graph with four possible optimal paths from source S to target T. The numbers on the edges correspond to edge IDs. Tie-breaking starts with the red node in Fig. 4b: the path through edge 3 is chosen over edge 4 because of the lower edge ID, reducing the number of optimal path candidates by half. Moving onto the next red node (Fig. 4c), the path through edge 7 is chosen over edge 8, leaving a single optimal path in green. An important observation is that the tie-breaking could have been performed on either node at any time (even concurrently) and the result would always be the same.

This tie-breaking strategy may produce invalid solutions on graphs containing zero-cost cycles (paths in the graphs with zero cost that start and end at the same node). Fortunately, zero-cost loops do not occur in FPGA routing graphs.

B. Stopping Criterion

Unlike a traditional A* implementation, the algorithm cannot terminate when the target is reached the first time. Due to the relaxed node traversal order, the target may be reached by a higher-cost path before an optimal path is found. Additionally, if there are multiple optimal paths, the algorithm must explore them all to ensure the ties are broken deterministically.

Therefore, we introduce a new stopping criterion: For node n, target t, and priority queue Q, stop when:

$$\nexists n \in Q \; s.t. \; \mathit{BestCost}(n) + h(n) \le \mathit{BestCost}(t). \quad (1)$$

$\mathit{BestCost}(n)$ is the best cost found so far for the path from the source node to node n, and $h(n)$ is an admissible prediction for the cost of the optimal path from node n to target t. If no node n that satisfies the inequality is in the priority queue, then no path can be traversed that will be better than the current best path to t; hence, the algorithm can terminate.

We enforce the stopping criterion by exploring all nodes in the queue, and use what we call *Post-Target Pruning* to reduce the number of nodes explored: *Prune a given node (do not explore it) if the cost of the best possible path from the source, through the node, to the target is higher than the cost of the best path found to the target so far.* Combined with the tie-breaking condition described earlier, enforcing this criterion ensures that the path returned by the algorithm is optimal and always the same, regardless of the node traversal order.

C. Heuristic Requirements

Standard A* requires an admissible heuristic to order the traversal of nodes. Since our relaxed order A* can traverse nodes in any order while guaranteeing optimal paths, the heuristic can be split in two.

1) Ordering Heuristic (h^o): This is used to order nodes in the queue, providing a hint for what nodes should be explored first. Since the order of node traversal can be relaxed in this algorithm, this heuristic does not need to be admissible. This allows the use of heuristics that are more accurate (lower error between actual and expected cost), but not necessarily underestimating. We will show in Section V that this freedom can be exploited to make the search more efficient.

2) Stopping Heuristic (h^s): This is used to compute $h(n)$ in the stopping criterion (Ineq. 1). For the algorithm to guarantee optimal paths and determinism, h^s must be admissible.

IV. MultiQueue-Based Parallel Path Search

Below we detail a MultiQueue-based path search that implements the enhancements of Section III. Given an admissible stopping heuristic, this algorithm performs an A* search; with an inadmissible heuristic, it is a directed search; and with a heuristic of zero, it reduces to a parallel Dijkstra's algorithm. To maximize scalability, our algorithm also minimizes locking and prunes nodes (partial paths) as soon as possible.

A. Algorithm Description

The MultiQueue-based parallel path search algorithm is described in Algorithm 1. s denotes the source, t the target, u the node currently being explored, v a neighbor node of u, and e the edge between nodes u and v. g_n is the (known) backward path cost from s to a node n. f_n is the estimated total cost for a path from s, through n and to t. f_n is calculated as the sum of g_n and h^o_n, where h^o_n is the ordering heuristic defined in Section III-C that estimates the cost from n to t.

Compared to a serial path search algorithm, key modifications include replacing the serial priority queue with a MultiQueue, relaxing the exit condition for optimality, adding tie-breaking checks to ensure determinism, and protecting the global node-cost updates with locks. In the following subsections, we discuss the algorithmic changes step-by-step.

Algorithm 1 Parallel MultiQueue-Based Path Search

```
1:  procedure FindShortestPath(source $s$, target $t$)
2:      $MQ \leftarrow$ InitMultiQueue($s$)                ▷ on main thread
3:      while TryPop($u$, $f_u$) from $MQ$ do             ▷ on all threads
4:          if PostPopPrune($u$, $f_u$, $t$) then continue
5:          for $e \in$ all out edges of $u$ do
6:              $v \leftarrow$ the neighbor of $u$ on $e$
7:              $g_v \leftarrow$ BackwardCost($u$, $v$, $e$)
8:              $h^o_v \leftarrow$ OrderingHeuristicCost($v$, $t$)
9:              $f_v \leftarrow g_v + h^o_v$
10:             if PrePushPrune($v$, $g_v$, $e$) then continue
11:             AcquireLock($v$)
12:             if PrePushPrune($v$, $g_v$, $e$) then
13:                 ReleaseLock($v$)
14:                 continue
15:             UpdateGlobalNodeCosts($v$, $g_v$, $f_v$, $e$)
16:             ReleaseLock($v$)
17:             if $v \neq t$ then Push($v$, $f_v$) to $MQ$
18:     return ReconstructPath($s$, $t$)                  ▷ on main thread
```

Algorithm 2 Pruning Functions

```
1:  function PostPopPrune($u$, $f_u$, $t$)                ▷ read-only
2:      if PostTargetPrune($u$, $t$) then return True
3:      if $f_u \neq$ BestTotalCost($u$) then return True
4:      return False                                      ▷ do not prune
5:  function PrePushPrune($v$, $g_v$, $e$)                ▷ read-only
6:      if $g_v >$ BestBackwardCost($v$) then return True
7:      if $g_v =$ BestBackwardCost($v$) then
8:          return !IsPreferred($e$, BestPrevEdge($v$))
9:      return False                                      ▷ do not prune
10: function PostTargetPrune($u$, $t$)                    ▷ read-only
11:     $g_u \leftarrow$ BestBackwardCost($u$)
12:     $h^s_u \leftarrow$ StoppingHeuristicCost($u$, $t$)
13:     $g_t \leftarrow$ BestBackwardCost($t$)
14:     if $g_u + h^s_u > g_t$ then return True
15:     return False                                      ▷ do not prune
```

1) Try-Pop from MultiQueue: The MultiQueue described in Section II-C provides a thread-safe pop operation with relaxed priority ordering. When the path search is running, multiple threads try to pop from the MultiQueue concurrently. Each thread will pop a node, explore its neighbors, and repeat. To ensure the stopping criterion from Section III-B is met, threads continuously pop until the queue is empty. At the same time, the `TryPop` routine waits until all threads reach a consensus that all queues are empty and there is no more work to do. This is essential as the MultiQueue could become transiently empty while a thread is still expanding the neighbors of a node it just popped.

2) Prune Nodes: Both pre-push and post-pop pruning, described in Algorithm 2, are crucial for reducing redundant work, finding optimal paths, and maintaining determinism.

After node u is popped from the queue, `PostPopPrune` decides whether to explore the neighbors of u or to prune. Initially, it performs Post-Target Pruning based on the stopping criterion. This check uses the stopping heuristic (h^s_u) to prune u if it could not possibly lead to a better path to the target (Section III-B). Then, the current total estimated cost of the path through node u (f_u) is compared to the best total cost so far (most recently pushed) for that node and, if the two are different, the node u is pruned. During the wave expansion, u may be pushed to the queue multiple times. For example, node u may be pushed to the queue and then, before u is popped from the queue, a better path to u may be found and pushed to the queue. Here we are using f_u as an optimistic identifier to check if the pair (u, f_u) is the most recently pushed element for node u. This reduces redundant work.

When iterating over the neighbors of u, `PrePushPrune` determines whether a path through u to its neighbor node v has a better backward cost than the best path to v found so far (breaking ties if needed). Section III-A details tie breaking.

3) Conditionally Atomically Update Global Node Costs: The parallel path search algorithm maintains the best path information in shared node cost variables. A per-node lock protects the update to these costs to prevent data races, creating a critical section from Line 11 to 16 in Algorithm 1. The pre-push pruning, which determines whether to update the node costs, must be atomic with the update itself to ensure updated data is valid (Line 12). Since different threads rarely work on the same node simultaneously, we use a fine-grained locking strategy of one lock per node to reduce contention. To further reduce lock contention, we add a cheap read-only check before acquiring the lock (Line 10), motivated by Shun et al. [30].

4) Push Neighbors into MultiQueue: During neighbor expansion, nodes are pushed into the queue in a thread-safe manner using the MultiQueue push operation. Locking occurs on individual queues within the MultiQueue, so spreading node insertions and the resulting heap updates across multiple queues reduces lock contention and improves concurrency.

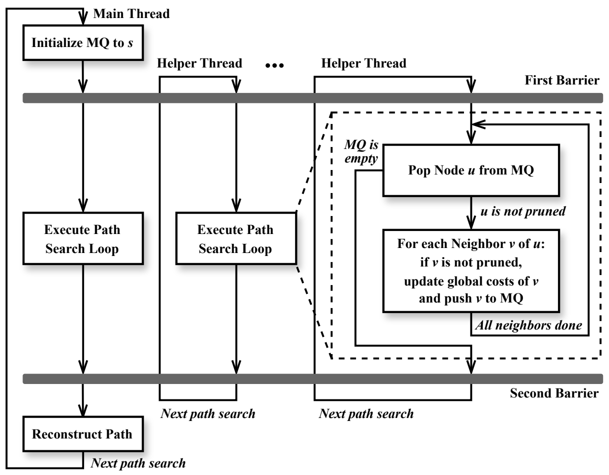


Fig. 5: The multi-threading strategy of the router.

B. Multi-Threading Strategy

The connection router will be invoked many times—once for every connection to be routed. Creating new threads for each invocation would be expensive, so we create all threads at the start of routing and organize them as shown in Fig. 5.

N threads collaboratively but asynchronously route each connection. Initially, threads synchronize at the first barrier, waiting for the main thread to initialize the MultiQueue with the net source. Next, all threads execute the path search loop (Algorithm 1). When all threads agree that the MultiQueue is empty, they synchronize at the second barrier to allow the main thread to then reset the MultiQueue, reconstruct the routed path, reset the global node state, and push the next net source to the MultiQueue for the next connection route. When the routing algorithm completes, the main thread sets a flag in the router destructor, causing helper threads to self-destruct.

C. Integration within VTR

Our parallel router is open source and built on top of the VTR 8+ serial router; allowing for a fair, direct comparison.[1]

1) Data Structure and Memory Optimization: The AIR router within VTR uses a complex dynamic memory allocation scheme so that the heap implementing its priority queue can grow as needed with few calls to `new`. This approach would be challenging to make thread-safe. Instead, we changed the node pruning strategy so that pushing a node to the heap requires storage only for its unique ID (n) and its f_n value. The node ID indexes into the `GlobalNodeCosts` vector, which stores other node states such as the backward cost, g_n. Post-pop pruning (Algorithm 1 Line 4) ensures that the entry in `GlobalNodeCosts[n]` always contains the data associated with the most promising path to n found so far.

2) MultiQueue Enhancement: The MultiQueue can use an arbitrary heap as its underlying data structure. We find that the VTR binary heap outperforms other commonly used heap implementations, such as the STL heap. When used in path search, it is designed to be more cache-friendly and reduce arithmetic operations when maintaining the heap. Using the VTR-like binary heap in the MultiQueue is particularly beneficial when the heap becomes large (i.e., when a large part of the graph needs to be explored).

In some of the path search algorithms discussed in Section II-A, when no further solutions need to be explored to guarantee optimality or determinism after reaching the target, we could empty a queue quickly by discarding all elements within it. To enable this *queue draining optimization*, we enhance the MultiQueue by adding a clear (drain) method.

3) Heuristic Transformation: The VTR heuristic, h_{VPR}, is not guaranteed to be admissible (Section II-A). We apply an affine transformation on h_{VPR} to calculate the stopping heuristic, h^s, and ordering heuristic, h^o, to make them admissible when necessary. We use the following *factor* and *offset* approach and pass these as command-line options to VPR:

$$h^{s/o} = \max\left(0, \left(h_{\text{VPR}} - \mathit{offset}_{s/o}\right) \times \mathit{factor}_{s/o}\right), \quad (2)$$

[1]**GitHub Repo:** vtr-verilog-to-routing **Branch:** mq-parallel-router

TABLE I: Ordering and Stopping Heuristics

Dijkstra's	$h^o = 0$	$h^s = 0$
A*	$h^o = 0.9 \times h_{\text{VPR}}$ §	$h^s_{\text{VTR}} = h_{\text{VPR}} - 3.05 \times 10^{-10}$ $h^s_{\text{Koios}} = h_{\text{VPR}} - 7.2 \times 10^{-10}$ $h^s_{\text{Titan}} = h_{\text{VPR}} - 3.2 \times 10^{-10}$
Directed	$h^o = 1.2 \times h_{\text{VPR}}$ §	$h^s = 1.2 \times h_{\text{VPR}}$ §

All offsets are architecture-specific and were empirically found.
§ Heuristic is not admissible.

where s or o is used depending on the context.

The stopping heuristic h^s must be admissible for our path search algorithm to guarantee optimal paths and be deterministic (III-C). The ordering heuristic only needs to be admissible when the queue draining optimization described above is used; admissibility is necessary in that case to ensure emptying queues of entries with higher costs does not prune possibly superior solutions.

V. Results

We evaluate the performance and quality of results (QoR) of our parallel router used within the overall AIR routing algorithm [5], [17]. Our parallel path search is integrated into VTR 8+ (commit `c1c1e3dc6` of March 2024) and compared to the unmodified VTR 8+ router (in that commit).

A. Methodology

We run our experiments on an Intel Xeon E5-2650 v4 CPU with a base frequency of 2.2 GHz running Ubuntu 16.04 with 12 physical cores in a single socket and 128 GiB of DRAM. We use the MultiQueue implementation by Zhang et al. [31]. We find that setting the number of queues per thread (c in Section II-C) to four works best for our application.

This work is tested on three benchmark suites: the eight largest circuits from the VTR Benchmark suite (general workloads) [32], the 11 largest circuits from the Koios 2.0 Benchmark suite (deep learning workloads) [33], and 22 designs from the Titan23 Benchmark suite[2] (large workloads) [34]. We run the VTR benchmarks on VTR's flagship architecture [17], the Koios benchmarks on a 22 nm Intel-like architecture described by Arora et al. [33], and the Titan benchmarks on a Stratix IV architecture capture [34].

VTR can perform either *Fixed Channel Width Routing* or a *Minimum Channel Width Search*. We consider the former as an example of (mostly) low-stress routing (limited congestion) and the latter as an example of high-stress routing (considerable congestion). The VTR benchmarks have a fixed channel width set to 1.3× the minimum channel width (found by VTR's serial router) for each circuit; the Koios and Titan benchmarks have a channel width set to 300. A minimum channel width search performs a binary search over different channel widths for each circuit to find the smallest channel width that is routable; hence, much of the routing performed will be under high-stress conditions.

[2]The largest Titan design (gaussian_blur) was excluded as it does not successfully route in VTR with default placement effort.

TABLE II: Parallel Router (12T) Speedup Over VTR 8+

Task	Serial Algorithm	Parallel Algorithm	VTR Speedup	Koios Speedup	Titan Speedup
Fixed Channel Width	Dijkstra's	Dijkstra's	8.32×	11.4×$^{\star 5}$	18.7×$^{\star 14}$
	A*	A*	5.51×	11.0×	13.2×$^{\star 2}$
	Directed	Directed †	1.28×	1.63×	1.98×
	Directed	A*	0.67×	0.11×	0.43×
Min Channel Width Search	Dijkstra's	Dijkstra's	6.05×$^{\star 1}$	-	-
	A*	A*	3.85×	-	-
	Directed	Directed †	2.18×	2.68×	-
	Directed	A*	1.51×	1.31×	-

† The parallel router is nondeterministic for directed search.
⋆ Number of dropped circuits due to serial router timeouts (12 hours).

TABLE III: Parallel Router (12T) Quality of Results

Serial Algorithm	Parallel Algorithm	VTR		Koios		Titan	
		CPD	WL	CPD	WL	CPD	WL
Dijkstra's	Dijkstra's	1.00	1.00	1.03	1.00	1.00	1.00
A*	A*	1.00	1.00	0.99	1.00	1.01	1.00
Directed	Directed †	1.00	1.00	1.02	0.98	0.98	0.99
Directed	A*	1.00	0.99	0.99	0.96	0.99	0.99

All values are normalized to the results of VTR's serial router.
† The parallel router is nondeterministic for directed search.

Our parallel router can be configured to perform one of three path search algorithms by selecting ordering and stopping heuristics, shown in Table I. Setting both heuristics to zero gives Dijkstra's algorithm. Although Dijkstra's algorithm is not commonly used in FPGA routing because it is much slower than A* and achieves the same QoR, it is used within VTR to compute the Map Router Lookahead, which relies on single-source all-destinations searches (Section II-A). Computing the router lookahead for an FPGA architecture can take an hour or more, motivating faster Dijkstra implementations.

An admissible stopping heuristic yields an A* search. For each architecture we found an admissible heuristic by observing the difference between h_{VPR} and the actual cost of paths. Since VTR is used for architecture exploration, h_{VPR} is usually computed at run time. To make the run time manageable, VTR coarsens its heuristic, leading to estimation errors.

h_{VPR} can be scaled up to produce a directed search. VTR does this by default ($h_{\text{VPR, DEF}} = 1.2 \times h_{\text{VPR}}$) to find a path faster at the expense of quality. Since $h_{\text{VPR, DEF}}$ is not admissible, the path returned is not guaranteed to be the lowest-cost path; however, FPGA routing algorithms make many approximations, so a small quality loss due to using $h_{\text{VPR, DEF}}$ is tolerable. Using $h_{\text{VPR, DEF}}$ as the stopping heuristic in our parallel router will lead to nondeterministic results.

The queue draining optimization (Section IV-C) generally provides around an 8% improvement in run time for path search; however, using queue draining on deterministic A* resulted in around a 30% slower run time than not using queue draining due to needing an admissible ordering heuristic (which worsens the node exploration of A*). Consequently, all path search algorithms except A* use queue draining.

B. Parallel Router Performance

Table II summarizes the speedups achieved by the parallel router running with 12 threads compared to VTR's serial router on different path search algorithms for both fixed channel width routing and minimum width searches. Table III similarly shows the critical path delay (CPD) and wirelength (WL) for those same tests, normalized to the results of VTR's serial router. Each circuit was given a timeout of 12 hours. Although the parallel router completed all circuits within the time limit, if VTR's serial router timed out on a circuit, that circuit was excluded from the geomean speedup. The dashes in the table are when all of the circuits timed out on VTR's serial router.

The first three rows of the tables show the parallel and serial routers performing the same path search algorithm. For Dijkstra's and A*, the parallel router achieves the largest speedups and has QoR similar to the serial router. A desirable quality of the parallel router for these searches is that it is *serially equivalent*; meaning it always obtains the same result, regardless of the number of threads used. This is due to the stopping criterion and tie-breaking condition, described in Section III, ensuring that the same path is always returned. For directed, the parallel router is faster and achieves slightly better QoR (as threads explore more nodes before the target is reached). The parallel router is nondeterministic when performing a directed search; however, the standard deviation of the CPD and WL normalized to their means is below 0.07%.

The fourth row of the tables shows the speedup and QoR for the parallel router running A* (determinism is desired) compared to the VTR's serial router running directed (default settings). Since the parallel router returns the lowest-cost paths, it often gets a better QoR than the serial router; however, since it explores more of the graph, it is ~1.5× (VTR benchmarks) to ~9.1× (Koios benchmarks) slower. The wide variation in the slowdown is due to the varying accuracy of h_{VPR} for the different architectures. As Table I shows, h^o for Koios requires a larger offset to h_{VPR} to guarantee admissibility. This demonstrates the importance of a good quality heuristic.

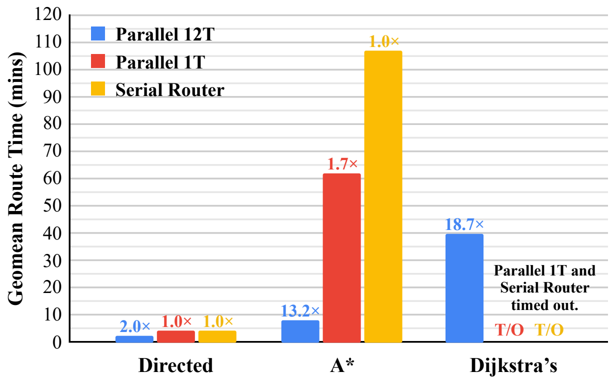


Fig. 6: Geomean route time for the parallel router using 12 or a single thread vs. VTR's serial router on the Titan benchmarks. Above the bars are the speedups over the serial router.

The parallel router generally achieves higher speedups on larger circuits with more complex FPGA architectures (Titan) than on smaller circuits with less complex FPGA architectures (VTR). It also tends to achieve higher speedups for Dijkstra's algorithm than for A*, and for A* than for directed. This is due to the increased amount of parallelizable work when larger graphs are traversed and more of the graph is explored.

Fig. 6 shows how the parallel router achieves a 13.2× speedup with 12 threads on the Titan benchmarks for A* during fixed channel width routing. As discussed in Section IV-C, the parallel router uses a queue optimized for parallel node storage. This optimization benefits larger wave expansions, while the serial VTR is better tuned for smaller expansions. Consequently, the parallel router achieves a 1.7× speedup over serial VTR on A* with a single thread; however, for directed path search, both routers have similar run times with one thread.

Fig. 7 shows the proportion of the route time for path search for the different search algorithms using VTR's serial router on Titan with fixed channel width routing. Amdahl's Law [35] implies that the highest achievable speedup on a directed search of Titan is $\frac{1}{(1-0.61)} = 2.5\times$. Compared to VTR's serial router with default settings (directed path search), our parallel router achieves 2× overall (nondeterministic) route time speedups, nearly reaching this scalability ceiling.

For minimum channel width searches, the parallel router always outperforms VTR's serial router. Even when determinism is required, and the parallel router is using an admissible stopping heuristic while the serial router is not, the parallel router achieves a 1.3× (Koios benchmarks) and 1.5× (VTR benchmarks) speedup. We attribute this to many of the routing instances of minimum channel width search being in high stress conditions with considerable congestion that makes a more directed heuristic less effective and requires more of the graph to be explored, playing to the parallel router's strengths.

C. *Parallel Router Scalability*

Fig. 8 shows the geomean run time speedup of the parallel path search (excluding other routing tasks) running path searches on the Titan benchmarks. The larger wave expansions produced by Dijkstra and A* searches lead to good scalability to 12 threads, while the smaller wave expansions of directed searches have limited scalability beyond six threads.

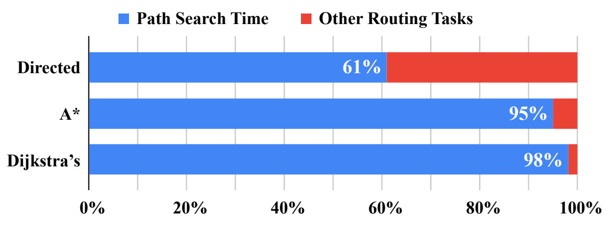


Fig. 7: Path search time as a proportion of overall route time for the Titan benchmarks using VTR's serial router. Other routing tasks include path reconstruction, timing analysis, graph congestion cost updates, and resetting global node costs.

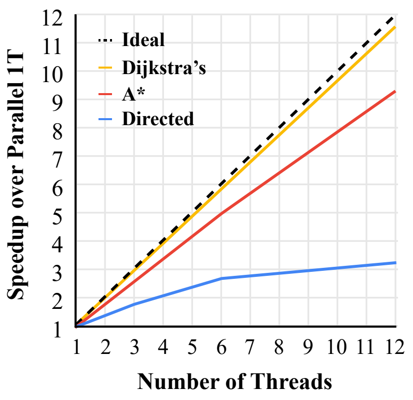


Fig. 8: Geomean path search run time speedup on Titan.

VI. Conclusions and Future Work

We present an open-source MultiQueue-based, fine-grained parallel router with relaxed node traversal ordering based on VTR's serial router. Our router deterministically returns the same optimal path, regardless of the number of threads, through algorithmic upgrades to the A* Shortest Path Algorithm. Compared to VTR's serial router, we measure route time speedups at 12 threads of 18.7× and 13.2× on Dijkstra's and A* path searches, respectively. Compared to the VTR router's default (directed search) settings, our router achieves better QoR 2× faster but is nondeterministic in this mode.

A limitation of this work is that the algorithm cannot produce deterministic results when the stopping heuristic is inadmissible. However, our algorithm could be run with an inadmissible heuristic (enabling a directed search) by designers who are comfortable with nondeterminism (potentially during design tuning); a change of heuristic enables deterministic results for final implementation if needed. The results presented in this work also use a simple scaling and shifting technique to generate admissible heuristics from VTR's inadmissible heuristic. Improvements to VPR's architecture-aware heuristic generator to make it more accurate (and hence require less shifting and scaling to become admissible) would speed up our parallel A* router, potentially enabling it to overtake the serial directed search in speed while enhancing quality. Since our parallel algorithm performs well on Dijkstra searches, it could enable a more accurate VTR router lookahead by sampling more routing start nodes within a time budget. Parallel path search can also be combined with coarse-grained parallel techniques that route multiple nets or connections in parallel to scale to higher core counts.

VII. Acknowledgments

This work was supported by the Intel/VMware Crossroads 3D Academic Research Center, NSERC, and QuickLogic. Mirjana Stojilović was partially supported by the Swiss National Science Foundation (grant No. 218864). Guozheng Zhang was partially supported by an Ontario Queen Elizabeth II Graduate Scholarship.

References

[1] A. Boutros and V. Betz, “FPGA Architecture: Principles and progression,” *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, May 2021.

[2] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken, “The Stratix™ 10 highly pipelined FPGA architecture,” in *Proceedings of the 24th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA: ACM, Feb. 2016, pp. 159–168.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2019.

[4] L. McMurchie and C. Ebeling, “PathFinder: A negotiation-based performance-driven router for FPGAs,” in *Proceedings of the third ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA: ACM, Feb. 1995, pp. 111–117.

[5] K. E. Murray, S. Zhong, and V. Betz, “AIR: A fast but lazy timing-driven FPGA router,” in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Beijing, China: IEEE, Jan. 2020, pp. 338–344.

[6] D. Vercruyce, E. Vansteenkiste, and D. Stroobandt, “CRoute: A fast high-quality timing-driven connection-based FPGA router,” in *Proceedings of the 27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. San Diego, CA, USA: IEEE, Apr. 2019, pp. 53–60.

[7] R. Y. Rubin and A. M. DeHon, “Timing-Driven Pathfinder Pathology and Remediation: Quantifying and reducing delay noise in VPR-pathfinder,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA: ACM, Feb. 2011, pp. 173–176.

[8] Y. Zha and J. Li, “Revisiting PathFinder routing algorithm,” in *Proceedings of the 30th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Virtual Event, USA: ACM, Feb. 2022, pp. 24–34.

[9] C. Gianos, “Architecting for flexibility and value with next gen Intel® Xeon® processors,” in *2023 IEEE Hot Chips 35 Symposium (HCS)*. Los Alamitos, CA, USA: IEEE, Aug. 2023, pp. 1–15.

[10] K. Troester and R. Bhargava, “AMD next generation “Zen 4” core and 4th gen AMD EPYC™ 9004 server CPU,” in *2023 IEEE Hot Chips 35 Symposium (HCS)*. Palo Alto, CA, USA: IEEE, Aug. 2023, pp. 1–25.

[11] M. Stojilović, “Parallel FPGA routing: Survey and challenges,” in *Proceedings of the 27th International Conference on Field-Programmable Logic and Applications (FPL)*. Ghent, Belgium: IEEE, Sep. 2017, pp. 1–8.

[12] H. Rihani, P. Sanders, and R. Dementiev, “MultiQueues: Simple relaxed concurrent priority queues,” in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. Portland, OR, USA: ACM, Jun. 2015, pp. 80–82.

[13] A. Ludwin and V. Betz, “Efficient and deterministic parallel placement for FPGAs,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, pp. 1–23, Jun. 2011.

[14] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Norwell, MA, USA: Kluwer Academic publishers, Mar. 1999.

[15] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[17] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, “VTR 8: High-performance CAD and customizable FPGA architecture modelling,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, pp. 1–55, Jun. 2020.

[18] M. Gort and J. H. Anderson, “Accelerating FPGA routing through parallelization and engineering enhancements,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 61–74, Jan. 2012.

[19] M. Shen and G. Luo, “Accelerate FPGA routing with parallel recursive partitioning,” in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*. Austin, TX, USA: IEEE, Nov. 2015, pp. 118–125.

[20] C. Zhu, J. Wang, and J. Lai, “A novel net-partition-based multithread FPGA routing method,” in *Proceedings of the 23rd International Conference on Field-Programmable Logic and Applications (FPL)*. Porto, Portugal: IEEE, Sep. 2013, pp. 1–4.

[21] C. H. Hoo and A. Kumar, “ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling,” in *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA: ACM, Feb. 2018, pp. 67–76.

[22] Y. Zhou, D. Vercruyce, and D. Stroobandt, “Accelerating FPGA routing through algorithmic enhancements and connection-aware parallelization,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 4, pp. 1–26, Aug. 2020.

[23] C. H. Hoo, Y. Ha, and A. Kumar, “ParaFRo: A hybrid parallel FPGA router using fine grained synchronization and partitioning,” in *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL)*. Lausanne, Switzerland: IEEE, Aug. 2016, pp. 1–11.

[24] M. Gort and J. H. Anderson, “Deterministic multi-core parallel routing for FPGAs,” in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT)*. Beijing, China: IEEE, Dec. 2010, pp. 78–86.

[25] Y. O. M. Moctar and P. Brisk, “Parallel FPGA routing based on the operator formulation,” in *Proceedings of the 51st Design Automation Conference (DAC)*. San Francisco, CA, USA: IEEE, Jun. 2014, pp. 1–6.

[26] Y. Moctar, M. Stojilović, and P. Brisk, “Deterministic parallel routing for FPGAs based on Galois parallel execution model,” in *Proceedings of the 28th International Conference on Field-Programmable Logic and Applications (FPL)*. Dublin, Ireland: IEEE, Aug. 2018, pp. 21–214.

[27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, “The tao of parallelism in algorithms,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA: ACM, Jun. 2011, pp. 12–25.

[28] A. Lenharth, D. Nguyen, and K. Pingali, “Priority queues are not good concurrent priority schedulers,” in *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*. Springer Berlin Heidelberg, Jul. 2015, pp. 209–221.

[29] D. Alistarh, J. Kopinsky, J. Li, and G. Nadiradze, “The power of choice in priority scheduling,” in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. Washington, DC, USA: ACM, Jul. 2017, pp. 283–292.

[30] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons, “Reducing contention through priority updates,” in *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, Jul. 2013, pp. 152–163.

[31] G. Zhang, G. Posluns, and M. C. Jeffrey, “Multi bucket queues: Efficient concurrent priority scheduling,” in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, Jun. 2024, pp. 113–124.

[32] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, K. B. Kent, J. Anderson, J. Rose, and V. Betz, “VTR 7.0: Next generation architecture and CAD system for FPGAs,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 1–30, Jul. 2014.

[33] A. Arora, A. Boutros, S. A. Damghani, K. Mathur, V. Mohanty, T. Anand, M. A. Elgammal, K. B. Kent, V. Betz, and L. K. John, “Koios 2.0: Open-source deep learning benchmarks for FPGA architecture and CAD research,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 3895–3909, May 2023.

[34] K. T. Khoozani, A. A. Dehkordi, and V. Betz, “Titan 2.0: Enabling open-source CAD evaluation with a modern architecture capture,” in *Proceedings of the 33st International Conference on Field-Programmable Logic and Applications (FPL)*. Gothenburg, Sweden: IEEE, Sep. 2023, pp. 57–64.

[35] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of AFIPS Spring Joint Computer Conference*. Atlantic City, New Jersey, USA: ACM, Apr. 1967, pp. 483–485.