ParaFRo: A Hybrid Parallel FPGA Router using Fine Grained Synchronization and Partitioning

Chin Hau Hoo
Department of Electrical &
Computer Engineering
National University of Singapore, Singapore
chinhau.hoo@u.nus.edu

Yajun Ha
Institute for Infocomm Research (I2R)
A*STAR, Singapore
ha-y@i2r.a-star.edu.sg

Akash Kumar
Technische Universität Dresden
Center for Advancing Electronics
Dresden, Germany
akash.kumar@tu-dresden.de

Abstract—**Routing of nets is one of the most time-consuming steps in the FPGA design flow. While existing works have described ways of accelerating the process through parallelization, they are not scalable. In this paper, we propose ParaFRo, a two-phase hybrid parallel FPGA router using fine-grained synchronization and partitioning. The first phase of the router aims to exploit the maximum parallelism available by routing nets while minimizing load imbalance. Instead of resolving contention with expensive software transactional memory, synchronization among threads is realized using lightweight spin mutexes. In the case where the algorithm detects that convergence is not possible in phase one, it transitions into phase two where convergence is prioritized over maximum parallelism. To achieve convergence, each thread in phase two routes only congested nets that have been assigned to it by a partitioner. The partitioner aims to reduce the contention among threads at the cost of an unbalanced load. In addition, periodic rip up of the entire route tree is employed to break the algorithm out from a local minimum. When only congested nets are rerouted, ParaFRo with 8 threads achieves an average speedup of 26.2X relative to VTR. In contrast, existing works managed to obtain an average speedup of up to 9.42X with 8 threads. Besides, ParaFRo is able to maintain the high speedups while producing similar quality of result as VTR in terms of critical path delay. Finally, the quality of result is relatively independent of the number of the threads.**

I. Introduction

Due to Moore's law, routing, which is one of the most time-consuming steps in the FPGA design flow, can take hours or even days to complete for a complex state-of-the-art design [11]. Therefore, there is a need for faster routing algorithms to cope with the exponential increase in the number of transistors per chip. Reducing the execution time of routing algorithms allow for improved productivity due to the shorter debug cycles. In addition, if routing is fast enough, it can be integrated into the placement stage to provide more accurate timing information, which leads to better placement quality. Faster routing also helps in design space exploration.

With multicore processors being commodities nowadays, parallelization is a promising way to accelerate routing algorithms. Unfortunately, the Pathfinder algorithm [9], which is the most commonly used algorithm for FPGA routing, is inherently sequential and parallelizing it is non-trivial. The routing resources that are available for the current net depends on the routes that are taken by the previously routed nets. Some existing works [5] partition the nets such that nets in different partitions do not have overlapping bounding boxes. However, this approach is not scalable because it is not always possible to have balanced partitions. Besides, there are always nets with bounding boxes that overlap that of nets in other partitions even after partitioning. To make matters worse, the number of such nets increases as the number of partitions increases. Another approach [10] to parallel routing is through speculative multithreading where the algorithm routes nets concurrently even if they have overlapping bounding boxes. It is only when there is a contention on a shared resource that the router rolls back to an earlier state and starts over. This approach scales well with the number of threads only when the possibility of contention is low because the rollback process has a high overhead.

Recognizing the limitations of purely partitioning and purely speculative based parallel router, we propose ParaFRo, a hybrid parallel router that combines the advantages of both. Our router has two phases. The first phase routes nets and resolves routing resources (RRs) contention among threads using lightweight and fine-grained synchronization. The advantage of this approach is that maximum amount of parallelism is exploited with very little overhead. In the second phase where convergence is more important than maximizing parallelism, each thread routes a subset of congested nets where the RR dependency among the subsets is minimized. In other words, ParaFRo's first phase exploits speculative multithreading without the overhead of rollbacks while its second phase leverages on partitioning to achieve convergence in case the first phase fails to do so.

Contributions. In summary, our contributions are as follows:

- A hybrid parallel FPGA detailed router that exploits the best of both speculative and partitioning-based parallel routers.
- Significant speedup of up to 26.2X relative to VTR [8] while maintaining comparable critical path delay.
- Observation that parallel routers based purely on independent bounding boxes are not scalable.

The rest of the paper is organized as follows. Section II introduces the background of classical and parallel routing problems. Section III describes the existing works on parallel FPGA routers. Section IV motivates the need for a new approach to parallel FPGA routing. Section V explains the design of our hybrid parallel router. Section VI evaluates our algorithm in terms of speedup and quality of result and compares it to existing works. Section VII describes some possible extensions to our current work. Finally, Section VIII concludes our paper.

II. Background

The routing problem is a classic problem in FPGA or VLSI design. The problem is usually modeled using a graph, $G(V, E)$ where V is the set of vertices that represent the RRs available on the silicon while E is the set of edges that represent the interconnections between the RRs. In the case of FPGAs, the set V contains prefabricated wires in the device while the set E contains programmable switches that connect the prefabricated wires together. In addition, there is a set N that represents all the nets to be routed using the RRs in the graph G. The objective of the routing problem is to find a tree for each net such that the union of all the trees results in no overused RRs and other requirements such as critical path delay are satisfied.

The parallel routing problem is an extension of the classic routing problem. In addition to the aforementioned objectives of the classic routing problem, the parallel routing problem is concerned with the acceleration of the classic routing algorithm when multiple processing elements (eg. processor cores) are available. A parallel routing algorithm generally breaks the classic routing problem down into smaller subproblems and solves them concurrently. In the ideal case where every subproblem is of the same size and there is no overhead, the speedup of a parallel routing algorithm is equal to the number of processing elements. However, in practice, the ideal speedup is rarely achieved due to the need for synchronization and communication when executing the parallel routing algorithm. In addition, the parallel routing problem is irregular [12]. The amount of parallelism is determined by the input to the problem. Therefore, it is impossible to determine at design time the optimal schedule to route the nets in parallel. In contrast, regular algorithms such as matrix multiplication can be parallelized easily.

Existing parallel routing algorithms can be broadly divided into two groups. The first group of algorithms [2] [5] is based on the independence of net bounding boxes. By restricting nets to use only RRs within their bounding boxes, nets with independent bounding boxes can be routed in parallel without the need for synchronization. Unfortunately, as discussed in Section IV-A, it is difficult to sustain the number of independent nets at or above the number of processing elements. The second group of algorithms [10] is the opposite of the first where nets can be routed in parallel even if they do not have independent bounding boxes. Instead, when a contention is detected at runtime (due to overlapping bounding boxes), the routing state up to the contended node is discarded and the routing process is restarted. We have discovered empirically that the second group of algorithms is more scalable, and our algorithm is based on that.

III. Related Works

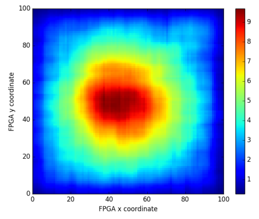
Gort and Anderson [5] parallelized VPR's [1] Pathfinder algorithm by spatially partitioning the FPGA. Nets that are fully contained in different partitions are routed concurrently. In Gort's approach, the number of partitions was equal to the number of processors executing the parallel router, and a heuristic was introduced to minimize the number of nets that cross the partition boundaries. Deterministic routing result was also guaranteed by calling the blocking version of the Message Passing Interface (MPI) receive function. However, the blocking function calls increased stall time when there was load imbalance among the processors, and the authors introduced a greedy algorithm that considered three factors – net fan-out, net bounding box and the number of visited routing resource nodes by a net when balancing the load among processors. Unfortunately, this spatial partitioning approach is not scalable as evident in the diminishing speedup as the number of processors increases. The reason is that as the number of processors (and equivalently the number of partitions) increases, the number of nets that cross partition boundaries increases, requiring more inter-processor communication and reducing speedup. Gort's results agree with our findings in Section IV, which motivates the need for another approach to parallel FPGA routing.

Shen and Luo [13] also proposed a parallel FPGA router based on MPI and recursive spatial partitioning where the nets are split into three subsets: two consisting of potentially overlap-free nets and one subset of potentially overlapping nets. The former two subsets are routed in parallel before routing the latter subset. Shen and Luo's parallel router achieved a speedup of up to 7.02X with 32 processes but has the same limitation as Gort's parallel router.

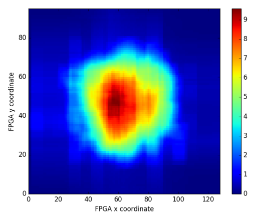
Instead of spatially partitioning the FPGA, Cabral *et al.* [2] leveraged on the routing resource independence property of FPGAs with the disjoint switch box topology. In the disjoint topology, the wires that are on the i-th track can only be connected to other wires on the same track. This restriction significantly simplifies the parallel routing algorithm by allowing each processor to independently route a subset of nets using a subset of the wires. As expected, the algorithm achieves an almost linear speedup because there is minimal inter-processor communication. However, the disjoint topology provides limited routability and is no longer used in state-of-the-art commercial FPGAs.

A fine-grained parallel router based on speculative parallelism was proposed by Moctar and Brisk [10]. Instead of relying on independent net bounding boxes and RRs like the aforementioned approaches [5] [2], Moctar parallelized the maze expansion step of the VPR router by using lock-based expansion operator and software transactional memory (STM) based priority queue. The approach has a good speedup of 5.46X with 8 threads. However, the overhead of lock acquisition and rollback due to STM reduces speedup as the number of threads increases.

Hoo *et al.* [6] modeled the routing problem as a linear program and identified that it can be decomposed into independent subproblems through Lagrangian relaxation. As a result, the nets can be routed in parallel, and high speedup of up to 7.05X was achieved. However, the method was applied only to the global routing problem, and it is unclear whether the method



(a) Heatmap of LU32PEEng benchmark from VTR 7



(b) Heatmap of stereo_vision benchmark from Titan

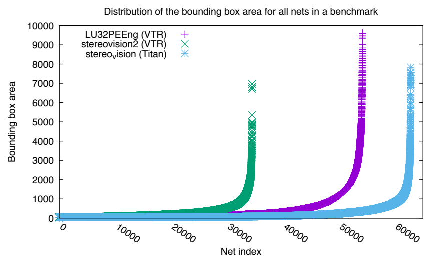


Fig. 2: Bounding box size distribution

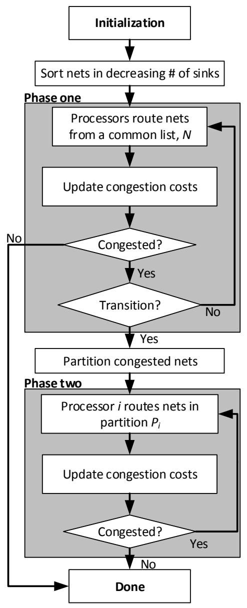


Fig. 3: Flowchart of the overall algorithm flow

can be extended to the detailed routing problem.

Chan *et al.* [3] analyzed how and when the congestion cost of the Pathfinder routing algorithm should be updated to increase parallelism and ensure convergence. They found that the second order congestion cost need not be broadcasted to other processors immediately after routing a net, and can be delayed until the end of a routing iteration. Therefore, the amount of inter-processor communication can be reduced, and parallelism is increased. On the other hand, the first order congestion cost must be broadcasted immediately.

IV. Motivation

In this section, we explain why the first group of parallel routing algorithms described in Section II is not scalable. To support our claims, three different net statistics of some FPGA benchmark circuits are considered. The statistics are bounding box spatial distributions and overlaps, and bounding box areas. They are illustrated in the following subsections.

A. Bounding Box Spatial Distributions and Overlaps

The heatmaps in Figures 1a and 1b show the spatial distribution of nets in large FPGA benchmarks – LU32PEEng (54217 nets) and stereo_vision (62824 nets), from VTR 7 [8] and Titan [11] respectively. The x and y axis of the figures corresponds to the x and y coordinates in the FPGA respectively. The temperature at a point in the heatmap is the percentage of nets whose bounding box covers the point. It can be observed that the nets are uniformly distributed in the FPGA. The problem with this is that there is no clear boundary at which the nets can be partitioned. Therefore, the resulting partitions contain a high number of nets with bounding boxes that span more than one partition. These nets cannot be routed in parallel.

The number of overlaps was determined using the METIS [7] tool for different values of $ubvec$. $ubvec$ is the parameter that controls the load balancing among partitions. It has a minimum value of 1, and the higher the value, the more unbalanced are the partitions. The output of METIS showed that up to 30% of the edges are inter-partition edges when all the partitions are totally load balanced ($ubvec = 1$). The high percentage of inter-partition nets significantly reduces the amount of parallelism. Therefore, algorithms that rely on bounding box independence do not scale well.

B. Bounding Box Areas

Figure 2 shows the bounding box area of all nets in different benchmarks. It can be seen that there is a significant number of nets with very large bounding boxes. These nets cannot be routed in parallel because they overlap with one another. To make matters worse, it has been shown that they dominate the total routing time [4]. Therefore, relying only on independent bounding boxes for parallel routing is not good enough.

V. Hybrid Parallel FPGA Router

Before describing the details of the algorithm shown in Algorithm 1, we start by highlighting its gist shown in Figure 3. The algorithm works in two phases. In the first phase, we consider two different scheduling policies: static and greedy. In the static policy, each processor/thread is allocated a fixed set of nets to route while in the greedy policy, each processor/thread continuously works as long as there are still nets that are not routed. The algorithm transitions into the second phase when the number of overused RR nodes stops decreasing monotonically. To prepare for the second phase, nets that use congested RR nodes are split into a number of partitions that is equal to the number of threads. The nets in each partition are then dispatched to a thread to be routed in the second phase. It is important to note that in phase two, only congested nets from phase one are routed.

Instead of imposing a strict requirement of net bounding box independence for parallel routing found in existing works

Algorithm 1 ParaFRo

```
function PARAFRO($N$)
    Input: List of nets to route, $N$
    Output: Route trees of nets in $N$

    Build RR graph
    $prev\_n\_overused \leftarrow \infty$
    $routed \leftarrow$ false
    $p\_i \leftarrow 0$
    $phase \leftarrow PHASE\_ONE$
    Sort $N$ in decreasing order of number of sinks

    while $i < max\_iterations$ and $!routed$ do
        if $phase == PHASE\_ONE$ then
            parallel_route($N$)                          ▷ Algorithm 2
        else
            partitioned_parallel_route($P$, $p\_i$)      ▷ Algorithm 3
            $p\_i \leftarrow p\_i + 1$
        end if

        $n\_overused \leftarrow$ get_n_overused_nodes()
        if $n\_overused == 0$ then
            $routed \leftarrow$ true
        else if $phase$ != $PHASE\_TWO$ and $n\_overused > prev\_n\_overused$ then
            $congested\_nets \leftarrow$ get_congested_nets()
            $P \leftarrow$ partition($congested\_nets$)   ▷ Section V-D
            $phase \leftarrow PHASE\_TWO$
            $p\_i \leftarrow 0$
        end if
        $prev\_n\_overused \leftarrow n\_overused$

        Update RR node congestion costs                  ▷ [8]
        $i \leftarrow i + 1$
    end while
    if $routed$ then
        return Route trees of $N$
    else
        return NULL
    end if
end function
```

Algorithm 2 Parallel Router

```
procedure PARALLEL_ROUTE($N$)
    Input: List of nets to route, $N$

    if $GREEDY\_POLICY$ then
        $cur\_net \leftarrow 0$
    else
        $cur\_net \leftarrow tid$
    end if
    parallel_while $cur\_net < sizeof(N)$
        $net \leftarrow N[cur\_net]$
        Rip up congested parts of $net$                  ▷ Section V-B
        route_one_net($net$)                             ▷ Algorithm 4
        if $GREEDY\_POLICY$ then
            ++$cur\_net$                 ▷ Atomically increment $cur\_net$
        else
            $cur\_net \leftarrow cur\_net + num\_threads$
        end if
    end_parallel_while
end procedure
```

[5], both phases of our algorithm route nets concurrently even if their bounding boxes overlap. Concurrent accesses to a RR node are synchronized with the use of mutexes. Due to how the route trees are updated, converging to a congestion-free result is an issue. Therefore, we also introduce methods to address this problem. The details of the methods are further elaborated in the following subsections.

A. ParaFRo

With the previous overview in mind, we now explain the pseudocode in Algorithm 1 in more detail. The algorithm starts by generating a RR graph that corresponds to the input FPGA architecture. Various variables ($prev\_n\_overused$, $routed$, p_i) that keep track of the state of routing are also initialized. Before the routing is started, the input list of nets, N is sorted in decreasing order of number of sinks. This is due to the fact that nets with a large number of sinks are generally harder to route when there is congestion.

Once the variables are initialized, the algorithm proceeds to route the nets in a while loop (Line 12) until the configurable maximum number of iterations or a congestion free state is reached. At the start of the loop, the variable $phase$ determines which phase the router is currently in. When the router is in phase one, **parallel_route** is called to route the nets in list N. When the router is in phase two, **partitioned_parallel_route** is called to route the nets in all the partitions in P. p_i is also passed into the function to keep track of how many iterations have passed since phase two started. It is then used by the function to determine whether to rip up all the nets in the partitions regardless of their congestion state. Details of this conditional rip up are explained in Section V-E. It is important to note that once the algorithm is in phase two, it does not transition back to phase one.

After routing with either **parallel_route** or **partitioned_parallel_route**, the number of congested or overused RR nodes is determined and stored into the variable $n\_overused$. Then, $n\_overused$ is checked and if it is equal to zero, $routed$ is set to true to indicate that routing is completed because there are no congested RR nodes. On the other hand, if $n\_overused$ is not zero, the condition highlighted in red in Algorithm 1 is checked. Basically, the condition checks whether the number of overused RR nodes has stopped decreasing monotonically, which is a sign that the algorithm is having difficulty converging. If the condition is true, the algorithm transitions into phase two by partitioning the nets into independent sets and setting $phase$ to $PHASE\_TWO$ and p_i to zero. The details of the partitioning are described in Section V-D.

Finally, the first and second order congestion costs [9] are updated in the same way as VTR [8] before moving on to the next iteration.

B. Phase One

parallel_route is the main driver of phase one, which routes the nets in N concurrently as shown by Line 9 of Algorithm 2. In our implementation, the threads responsible for routing the nets are spawned using Intel Threading Building Block (TBB) library's $parallel\_for$ loop. The number of threads can be controlled by using the $task\_scheduler\_init$ class of

Algorithm 3 Partitioned Parallel Router

```
1:  procedure PARTITIONED_PARALLEL_ROUTE($P$, $iter$)
2:      Input: Partitions of nets to route, $P$
3:      Input: Number of iterations since the start of phase two, $iter$.
4:
5:      parallel_for_all $P_i \in P$
6:          for all $net \in P_i$ do
7:              if $iter$ is mulitple of $RIP\_UP\_PERIOD$ then
8:                  Rip up the entire $net$                    ▷ Section V-E
9:              else
10:                 Rip up congested parts of $net$            ▷ Section V-B
11:             end if
12:             route_one_net($net$)                          ▷ Algorithm 4
13:         end for
14:     end_parallel_for_all
15: end procedure
```

Algorithm 4 Net Router

```
1:  procedure ROUTE_ONE_NET($net$)
2:      Input: Net to route, $net$
3:
4:      $min\_heap \leftarrow \{\}$
5:      $route\_tree \leftarrow \{$source of $net\}$
6:      for all $sink \in net.sinks$ do
7:          Add $route\_tree$ to $min\_heap$
8:          while !$min\_heap$.empty() and !$found\_sink$ do
9:              $current \leftarrow min\_heap$.pop()
10:             if $current == sink$ then
11:                 $found\_sink \leftarrow$ true
12:             else if $current$.cost $<$ $state[current]$.cost then
13:                 $state[current]$.cost $\leftarrow$ $current$.cost
14:                 for all $n \in current$.neighbors do
15:                     $c\_cost \leftarrow$ get_congestion_cost($n$)
16:                     $t\_cost \leftarrow$ get_timing_cost($current$, $n$)
17:                     $n$.cost $\leftarrow$ get_total_cost($c\_cost$, $t\_cost$)
18:                     $min\_heap$.push($n$)
19:                 end for
20:             end if
21:         end while
22:         Add path to $sink$ to $route\_tree$
23:         for all $node \in$ path to $sink$ do
24:             lock mutex of $node$
25:             Update first order congestion cost of $node$
26:             unlock mutex of $node$
27:         end for
28:         $min\_heap \leftarrow \{\}$
29:         for all $node \in$ modified RR nodes do
30:             $state[node]$.cost $\leftarrow \infty$
31:         end for
32:     end for
33: end procedure
```

the TBB library. The main goal of this phase is to achieve maximum parallelism by minimizing imbalance. We explored two scheduling policies to achieve it: greedy and static. In the greedy policy, each thread will try to get a net from the list N as long as N is not empty. In this case, none of the threads waste time idling. To allow lightweight acquisition of a net from N, an atomic variable cur_net is used to keep track of the first unrouted net in N. In the static policy, each thread routes every $(tid+k*num\_thread)$-th net from N where tid is the thread identifier, which ranges from 0 to $num\_thread-1$. This policy is a simple way of getting a reasonably balanced set of nets for each thread to route.

After an unrouted net is obtained and stored in net, its route tree is traversed starting from the root. As soon as a congested route tree node is found, the node and its leaves are recursively ripped up. We have also considered the same rip up mechanism as VTR [8] where the entire route tree is ripped up regardless of the congestion state of the nodes. In fact, we show empirically in Section VI-A that this mechanism is necessary to achieve convergence under high-stress condition.

After ripping up the net, the function **route_one_net** is called to route it. The function is also the place where synchronization among threads is implemented.

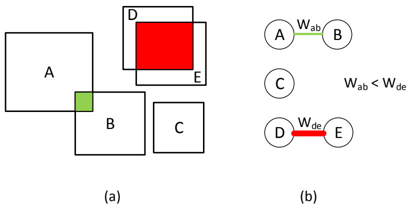


Fig. 4: (a) Bounding boxes (b) The associated dependency graph

C. Net Router

The **route_one_net** function shown in Algorithm 4 is essentially the same as that of VTR's [8] except for the differences highlighted in red. Since multiple threads can write to the congestion state of the same RR node at the same time, the writes need to be protected by a mutex to prevent race conditions. The choice of the type of mutex is important in determining the scalability of the parallel router because the introduction of a mutex basically serializes the execution of the router, which reduces speedup.

In our implementation, the mutexes are realized using TBB's $spin\_mutex$ class, which allows for lightweight acquisition and release of a lock. A $spin\_mutex$, as its name implies, only spins in a tight loop instead of transitioning into kernel mode when the mutex cannot be locked. Therefore, in the case where contention on the mutex is light, very little time is wasted spinning and the overhead of transitioning into kernel mode is avoided. However, when there is heavy contention, $spin\_mutex$ is less scalable than other types of mutexes. Fortunately, we have found empirically that the contention is light in most of the benchmarks.

The $spin\_mutex$ is used in the red highlighted parts of Algorithm 4 where synchronization among threads is required because the shared RR congestion state is accessed by multiple threads concurrently.

D. Transition to Phase Two

Since the congestion cost of the RR nodes are updated only after finding a path to the sink (Line 25, Algorithm 4), it is possible for the neighbor exploration stage to read outdated congestion state (Line 15, Algorithm 4). This is especially the case when nets with overlapping bounding boxes are being routed simultaneously. As a result, converging to a congestion-free state can be a problem.

TABLE I: Summary of benchmarks used in the experiments

Benchmark	Total nets	Total blocks	Channel width Min	120%	140%
stereovision1	11,078	1,205	104	126	146
LU8PEEng	16,276	2,373	114	138	160
stereovision2	34,473	2,939	154	186	216
LU32PEEng	54,217	7,544	174	210	244
neuron	55,434	3,473	206	248	290
stereo_vision	62,824	3,457	228	274	320

When the number of overused nodes stops decreasing, the congested nets are partitioned in such a way that the bounding box overlap of nets that are in different partitions are minimized, and the algorithm transitions into phase two. Minimization of overlap has a direct impact on enhancing convergence because it reduces the probability of reading outdated congestion state.

Before partitioning starts, a bounding box dependency graph is built. An example of the graph is shown Figure 4. The vertices of the graph represent the congested nets while the edges indicate bounding box overlaps between the associated vertices. Each vertex has a weight that measures the amount of work required to route the net, and it is used for a metric for load balancing. Since load balancing is not the main priority in phase two, the vertex weights are set to the bounding box size to approximate the workload. Each edge also has a weight that measures the size of overlaps between the bounding boxes of the nets (vertices) that the edge connects to.

Partitioning is performed using the METIS tool [7]. As explained in Section IV-A, METIS has a parameter $ubvec$ that sets the maximum amount of load imbalance among the partitions. In our case, we set $ubvec$ to be a large value of 10^6 to ensure that nets in each partition are as independent as possible with respect to nets in other partitions albeit at the cost of worse load balancing.

E. Phase Two

Phase two is executed by **partitioned_parallel_route** shown in Algorithm 3 after the congested nets are partitioned. It is similar to phase one but with two main differences. The first difference is that it only routes nets that are the result of partitioning described in Section V-D. The second difference is in terms of how the nets are ripped up. In addition to ripping up the route tree starting at the first congested node, the whole route tree is ripped up every RIP_UP_PERIOD iterations. The motivation behind this is to prevent a net from hogging a path that is required by other nets to route successfully. In other words, it allows the algorithm to break free from a local minimum. Similar to phase one, we have also experimented with ripping up the route tree unconditionally.

VI. Results

Our experiments were performed on a machine with dual Intel(R) Xeon(R) CPU E5-2670 v3 running at 2.3 GHz and 128 GB of RAM. The operating system is Ubuntu 14.04, and the kernel version is 4.2.0-30. Our parallel router was written in C++14 and compiled using gcc version 5.3.0 with the optimization flag set to -O3. For comparison purposes, VTR [8] was compiled using the same gcc version and optimization flag.

TABLE II: Notations

Notation	Meaning
120%	Channel width is 20% higher than the VTR minimum
140%	Channel width is 40% higher than the VTR minimum
all	All nets are rerouted regardless of their congestion status
cong.	Only congested nets are rerouted

TABLE III: Execution time (in seconds) of VTR and single threaded ParaFRo for different benchmarks under different configurations

Benchmarks	VTR 120%	VTR 140%	ParaFRo, all 120%	ParaFRo, all 140%	ParaFRo, cong. 120%	ParaFRo, cong. 140%
stereovision1	14	12	19	17	10	7
LU8PEEng	68	61	91	81	25	22
stereovision2	105	89	126	102	50	32
LU32PEEng	561	462	640	648	204	225
neuron	913	825	433	541	152	141
stereo_vision	473	486	351	397	108	118

Table I shows a summary of the benchmarks used for the experiments. They were circuits from VTR [8] and Titan [11]. The chosen VTR benchmarks were among the biggest in the package. MCNC [14] benchmarks were not evaluated because even the largest MCNC benchmark took less than a second to route. The architecture files used for VTR and Titan circuits were k6_frac_N10_mem32K_40nm.xml and stratixiv_arch.timing.xml respectively. The benchmarks were packed and placed using VTR before being routed by ParaFRo and VTR with different routing parameters.

A. Speedup

In this subsection, we study the effect of four factors on the speedup of ParaFRo: scheduling policy, channel width, net reroute condition and number of threads. We refer to the combination of the factors as a configuration, and the notations used to specify it are summarized in Table II.

The channel width factor is specified relative to the minimum channel width required to successfully route each benchmark with VTR. In our experiments, ParaFRo was executed with channel widths that are 20% and 40% higher than the minimum. Since ParaFRo is non-deterministic, five runs of ParaFRo were executed for each configuration, and the average was calculated. The results are shown in Table III, Figure 5 and 6. Since the execution times of ParaFRo for more than one thread can be calculated easily from Figure 5 and 6, they are not included in Table III.

1) Static scheduling: Figure 5 shows the normalized speedup of ParaFRo when static scheduling is employed. The speedups are normalized to the single threaded variant of ParaFRo for each configuration. An interesting result is that ParaFRo achieved super-linear speedup with certain configurations. The most significant examples are neuron and stereo_vision where super-linear speedup is achieved with 2 and 4 threads. This is due to the fact that these benchmarks are more difficult to route, and ParaFRo transitions into phase two earlier. As described in Section V-E, only congested nets from phase one are routed in phase two. Therefore, the workload in

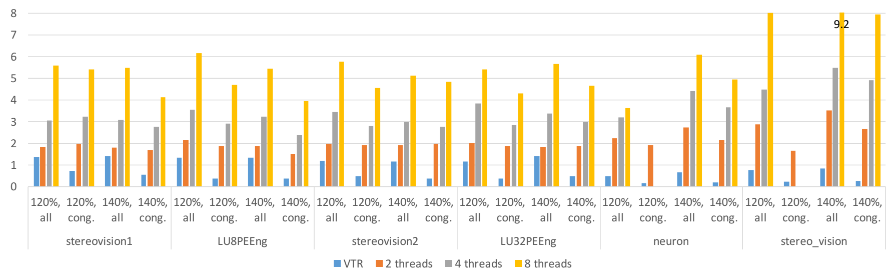


Fig. 5: Normalized speedups achieved by ParaFRo and VTR under different configurations with **static scheduling**. The results are normalized to the their respective single threaded ParaFRo configurations.

later iterations of the ParaFRo when routing these benchmarks is actually lower than that of single threaded ParaFRo. Due to the same reason, stereo_vision is able to achieve a speedup of close to 8X with 8 threads with the (120%, all) configuration. However, there is no super-linear speedup because the increase in the number of iterations required for convergence outweighs the advantage gained from a lower workload.

In addition, it can be seen from Figure 5 that when only congested nets are ripped up, the speedup of ParaFRo is generally lower than when all nets are ripped up at the start of every iteration. The reason is that routing only congested nets causes higher load imbalance in later iterations of the router.

Some configurations of ParaFRo also fail to successfully route larger benchmarks as indicated by speedup on 0 in Figure 5. It can be observed that when the channel width is 20% higher than the minimum and only congested nets are rerouted, the neuron and stereo_vision benchmarks fail to route with 4 and 8 threads. This is because some non-congested nets utilize RR nodes that are critical to successfully route the congested nets. The problem is resolved when the channel width is increased to 40% higher than the minimum. In addition, the same phenomenon is observed in Section VI-A2 when greedy scheduling is employed. Therefore, we conclude that all nets have to be rerouted in every iteration in order for ParaFRo to converge in a high-stress condition where the number of wires in the routing channel is limited.

The single threaded *all* variant of ParaFRo is algorithmically identical to VTR, and it can be seen in Figure 5 that ParaFRo is slightly slower than VTR for smaller benchmarks. On the other hand, the single threaded *cong.* variant of ParaFRo is significantly faster than VTR especially for larger benchmarks because not rerouting large and non-congested nets in ParaFRo significantly reduces execution time.

Another observation that can be made from Figure 5 is that the normalized speedup is relatively consistent across all benchmarks.

2) Greedy scheduling: Intuitively, greedy scheduling would achieve better speedup than static scheduling because of lower load imbalance since the threads do not spend time idling. However, as shown in Figure 6, this is generally not the case. In fact, only some benchmarks such as stereovision2 and LU32PEEng have higher speedup than static scheduling in certain ParaFRo configurations. This counter-intuitive observation can be explained with the result from [3] that the nets have to be routed in the same order in every iteration to achieve convergence. Greedy scheduling causes the order in which the nets are routed to be different across iterations. Therefore, the number of iterations required for convergence is actually higher in greedy scheduling. As a result, even though each routing iteration in greedy scheduling is shorter due to better load balancing, this gain is canceled out by the higher number of iterations required for convergence. An interesting extension that will be addressed in future work is to combine both greedy and static scheduling to obtain the best of both policies.

Another observation that can be made from Figure 6 is that in addition to the benchmarks that failed to route in Figure 5, stereo_vision fails to route with 120%, cong., and 2 threads. The reason behind this is the same as the aforementioned reasons [3].

B. Critical Path Delay

ParaFRo's critical path delay is also compared to VTR's to see if the quality of result is compromised as a result of parallelization. The bar chart in Figure 7 and 8 show the average critical path delay normalized to that of VTR's for static and greedy ParaFRo respectively. Anything that is below the red line indicates an improvement over VTR and vice versa.

It is clear from the figures that most benchmarks except stereovision1, neuron and stereo_vision have similar critical path delay as compared to VTR. neuron is even routed with a lower critical path delay than VTR. On average, the normalized critical path delay is 1.0008. Unfortunately, Moctar [10] and Shen [13] did not report the normalized critical path delay so we are not able to compare with them.

Since ParaFRo is non-deterministic, we also investigate the variations in terms of critical path delay. They are shown as error bars on top of the bar chart in Figure 7 and 8. Except for stereovision1 (Figure 8), neuron and stereo_vision (Figure 7 and 8), which have significant variations for some configurations, most benchmarks have very consistent critical path delay across different runs of the algorithm despite the non-deterministic nature of ParaFRo. In addition, it can be seen that greedy ParaFRo yields more variations as compared to static ParaFRo. However, despite the variations, it is important to note that the worst critical path delay is only 3% more than that of VTR's (stereovision1).

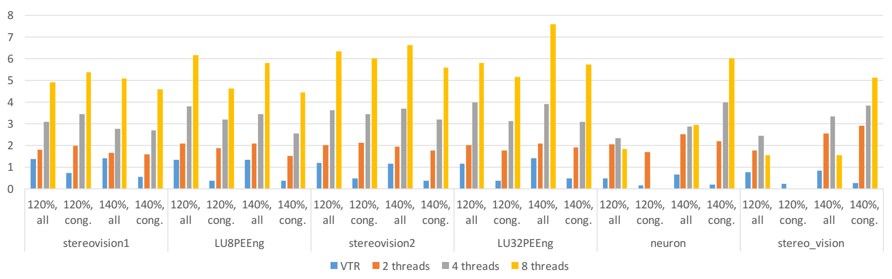


Fig. 6: Normalized speedups achieved by ParaFRo and VTR under different configurations with **greedy scheduling**. The results are normalized to the their respective single threaded ParaFRo configuration.

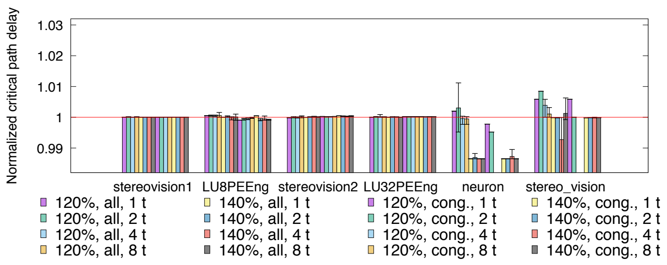


Fig. 7: Average critical path delay of **static** ParaFRo and its variation normalized to VTR's for various benchmarks

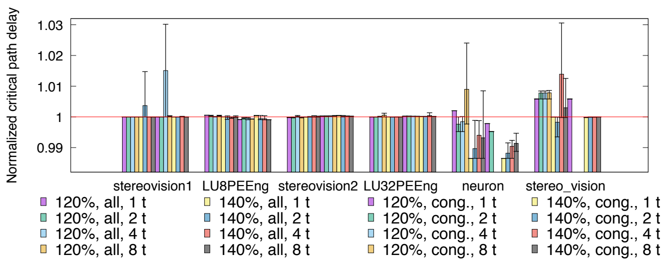


Fig. 8: Average critical path delay of **greedy** ParaFRo and its variation normalized to VTR's for various benchmarks

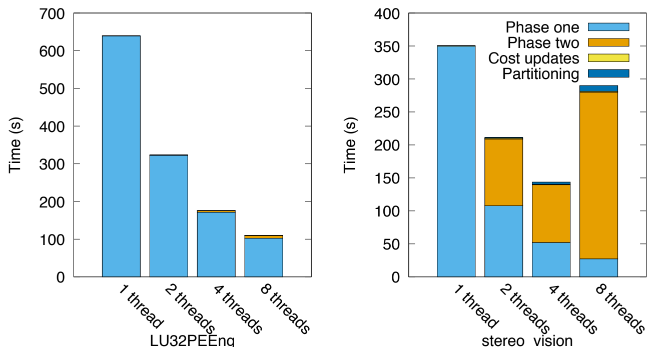


Fig. 9: Execution time profile of LU32PEEng and stereo_vision

TABLE IV: Summary of large Titan benchmarks used in the experiments

Benchmark	Total nets	Total blocks	Channel width		
			Min	120%	140%
cholesky_mc	65,672	5,328	238	286	334
des90	90,434	4,977	310	372	434
segmentation	125,589	9,047	292	352	410
stap_qrd	144,408	16,017	270	324	378
sparcT2_core	176,232	13,738	302	364	424
denoise	257,423	18,600	310	372	434

In addition, the critical path delay is relatively independent of the number of threads in most benchmarks. In other words, quality of result is not sacrificed in favor of higher speedup.

C. Execution Profile

Figure 9 shows two different execution profiles of ParaFRo (greedy, 120%, all). The two benchmarks are chosen to illustrate the need for phase two of ParaFRo. It can be seen that stereo_vision spends significantly more time in phase two than LU32PEEng. A possible reason is that the number of overlapping bounding boxes in stereo_vision is higher than that of LU32PEEng. Therefore, the probability of threads reading outdated congestion state in stereo_vision is higher as the number of threads increases. This prevents the number of congested RR nodes from decreasing monotonically, which ParaFRo detects as a signal to transition into phase two.

In addition, Figure 9 also explains the reason why stereo_vision has worse speedup than LU32PEEng in Figure 6. As the number of threads increases, the proportion of time spent in phase two for stereo_vision is higher than LU32PEEng. Since phase two focuses on convergence instead of load balancing, the speedup achieved in phase one is outweighed by the poor load balancing in phase two. Besides, the amount of time spent in phase two for stereo_vision with 8 threads is higher than that with 4 threads. As a result, the speedup is worse for 8 threads as compared to 4 threads as shown in Figure 6.

D. Large benchmarks

We have also experimented with larger Titan benchmarks shown in Table IV. The results are reported separately in this subsection because the minimum channel widths were determined differently where a heuristic was added into VTR's binary search to significantly reduce the time required. The heuristic came from the observation that the execution time of routing iterations increases exponentially when trying to route a benchmark with channel width that is too small. Such an increase in execution time is detected, and the current

TABLE V: Execution time (in seconds) of VTR and single threaded ParaFRo for large Titan benchmarks under different configurations

	VTR		ParaFRo, all		ParaFRo, cong.	
Benchmarks	120%	140%	120%	140%	120%	140%
cholesky_mc	1,224	1,270	762	188	755	169
des90	2,250	2,906	3,551	434	1,592	403
segmentation	2,317	2,617	1,541	227	1,303	213
stap_qrd	2,852	2,533	2,138	598	1,750	508
sparcT2_core	2,454	2,204	1,659	309	1,239	271
denoise	7,066	5,889	3,365	424	2,505	393

routing is terminated. The binary search then proceeds to route with the next possible channel width. This early termination significantly reduces the time required to find the minimum channel width albeit with slight overestimation in some cases. The minimum channel widths in Table IV is obtained by setting the threshold at which the routing is terminated to be 5 times the execution time of the first routing iteration.

1) Speedup: Similar to Section VI-A, the execution time of VTR and single-threaded ParaFRo is listed in Table V while the speedup of ParaFRo is shown in Figure 10 and 11.

It can be seen in Figure 10 and 11 that single-threaded ParaFRo is faster than VTR in all benchmarks except one scenario of des90 (120%, all). This is because we have added an enhancement that was proposed by Gort [5]. We observed that the Logic Array Block (LAB) in stratixiv_arch.timing.xml has equivalent data output pins. The equivalence causes convergence issues because the packer assumes that each net will use only one LAB output pin but nets with multiple sinks tend to use more than one of those pins during routing. In order to solve the problem, the enhancement that we have added forces the router to use the same output pin to route the rest of the sinks once the first sink of the net has been routed.

In addition, the speedup of static ParaFRo is relatively consistent across all benchmarks except for cholesky_mc (120%, all & 140%, all), des90 (120%, all) and stap_qrd. des90 (120%, all) achieved superlinear speedup because the single-threaded ParaFRo took twice as many iterations to route as compared to multi-threaded ParaFRo. cholesky_mc (120%, all & 140%, all) and staq_qrd have worse speedups compared to other benchmarks because our simple load balancing mechanism does not work well for the two benchmarks. Static ParaFRo also managed to route all the benchmarks in Table IV.

On the other hand, the speedup of greedy ParaFRo is generally less consistent and lower than that of static ParaFRo as shown in Figure 11. Greedy ParaFRo also failed to route some benchmarks under certain configurations. These observations are due to the same reason explained in Section VI-A2 that nets have to be routed in the same order in every iteration to ensure convergence.

2) Critical Path Delay: As shown in Figure 12[1], static ParaFRo achieved lower critical path delay than VTR in almost all benchmarks, and the critical path delay is relatively independent of the number of threads. Greedy ParaFRo also has lower critical path delay than VTR in most benchmarks as shown in Figure 13 but the worst degradation is around 5% higher than that of static ParaFRo.

[1]Due to time constraint, we managed to run ParaFRo only once for each configuration and benchmark. Therefore, the variations in terms of critical path delay are not shown.

E. Comparison with existing works

In Figure 14, we compare the average speedup of ParaFRo to existing parallel routers. The speedup is normalized to VTR and averaged across the twelve benchmarks that we evaluated so that a fair comparison with existing works can be made. While we used 20% and 40% higher than the minimum channel width for our experiments, Gort [5], Moctar [10] and Shen [13] used 30%, 40% and 40% respectively.

We can see in Figure 14 that with only 20% higher than the minimum channel width and rerouting all nets, static ParaFRo outperforms all existing parallel routers except Gort's enhanced router. On the other hand, when only congested nets are rerouted, ParaFRo has significantly higher speedup than all existing works. It is also faster than Gort's enhanced router, which also reroutes only congested nets.

Compared to VTR, static ParaFRo with 8 threads is 6.2X faster when all nets are rerouted with 20% higher than the minimum channel width. When only congested nets are rerouted, static ParaFRo with 8 threads is 26.2X faster than VTR with the same channel width (20% higher than the minimum). Given more routing resources (40% higher than the minimum), static ParaFRo with 8 threads is able to achieve a speedup of 6.8X and 27.6X relative to VTR when all and only congested nets are rerouted respectively.

VII. Future works

Currently, one of the limitations of ParaFRo is that it is non-deterministic. This is because the congestion costs seen by each thread is dependent on the time at which the thread discovers a RR node. Since threads do not run at exactly the same speed, the result of ParaFRo is non-deterministic due to race conditions.

Fortunately, we believe that it is possible to achieve determinism in ParaFRo, and we will continue to explore the idea in the future. In order to achieve deterministic results upon successive routing runs, it is imperative that all the nets see the same cost of resources as in the previous runs. This requires two conditions to be satisfied – all the nets should be routed in the same order, and the cost of the resources while a net is being routed should be the same as in previous runs.

While the first condition can be satisfied with a static scheduling algorithm and barriers to ensure the same order of nets during each run, the second condition requires a copy of the congestion state to be stored before routing a net in each thread. While routing a net, cost of using a RR node is calculated based on the stored congestion state. In this case, the congestion costs seen by a thread is no longer dependent on the execution speed of the thread. Once a net has been routed by a thread, the first order congestion costs are updated before waiting for other threads to complete the same using a barrier. The process is repeated until all the nets are routed. Although this approach guarantees determinism, the speedup may be

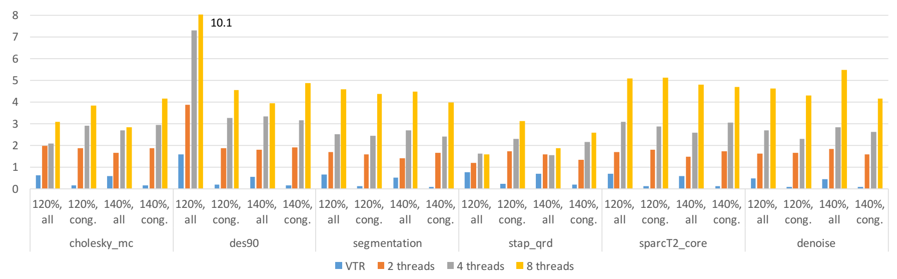


Fig. 10: Normalized speedups achieved by ParaFRo and VTR under different configurations with **static scheduling** for large Titan benchmarks. The results are normalized to the their respective single threaded ParaFRo configuration.

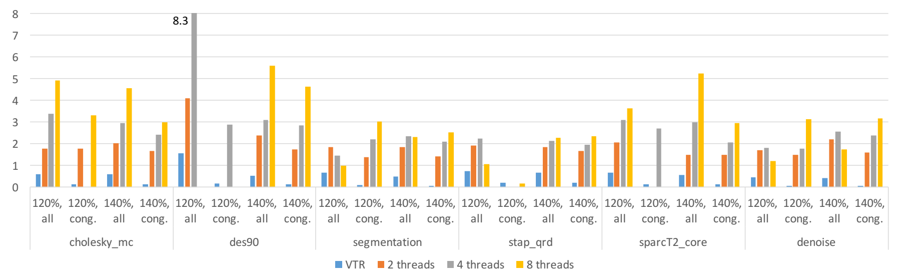


Fig. 11: Normalized speedups achieved by ParaFRo and VTR under different configurations with **greedy scheduling** for large Titan benchmarks. The results are normalized to the their respective single threaded ParaFRo configuration.

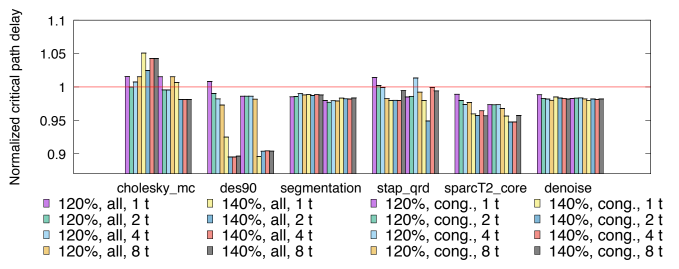


Fig. 12: Critical path delay of **static** ParaFRo normalized to VTR's for large Titan benchmarks

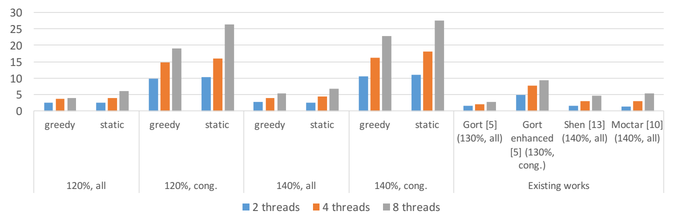


Fig. 14: Comparison of ParaFRo's average speedup with existing works

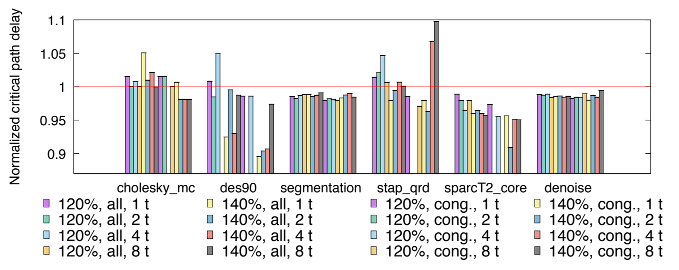


Fig. 13: Critical path delay of **greedy** ParaFRo normalized to VTR's for large Titan benchmarks

reduced due to the overhead of waiting in the case where workloads among threads are unbalanced. Therefore, we also plan to investigate more effective load balancing mechanisms so that speedup is not dramatically reduced when achieving determinism.

VIII. Conclusions

This paper presents ParaFRo – a hybrid parallel approach to route nets in an FPGA design. The algorithm has two phases: the first phase tries to minimize load imbalance in order to achieve maximum parallelism. Once the congestion stops decreasing monotonically, the second phase of the algorithm is triggered where the remaining congested nets are partitioned and rerouted. This hybrid approach provides a scalable methodology while ensuring convergence. The approach has been evaluated with benchmarks of various sizes from VTR and Titan. When only congested nets are rerouted, ParaFRo with 8 threads is 26.2X faster than VTR.

IX. Acknowledgments

This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed) at the Technische Universität Dresden.

REFERENCES

[1] V. Betz and J. Rose. VPR: A new packing, placement and routing tool for FPGA research. In *FPL*, pages 213–222, 1997.

[2] L. A. Cabral, J. S. Aude, and N. Maculan. TDR: A distributed-memory parallel routing algorithm for FPGAs. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 263–270. Springer, 2002.

[3] P. K. Chan, M. D. Schlag, C. Ebeling, and L. McMurchie. Distributed-memory parallel routing for field-programmable gate arrays. *IEEE TCAD*, 19(8):850–862, 2000.

[4] X. Chen, J. Zhu, and M. Zhang. Timing-driven routing of high fanout nets. In *FPL*, pages 423–428. IEEE, 2011.

[5] M. Gort and J. H. Anderson. Accelerating FPGA Routing Through Parallelization and Engineering Enhancements, Special Section on PAR-CAD 2010. *IEEE TCAD*, 31(1):61–74, 2012.

[6] C. H. Hoo, A. Kumar, and Y. Ha. Paralar: A parallel fpga router based on lagrangian relaxation. In *FPL*, pages 1–6. IEEE, 2015.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[8] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed, et al. VTR 7.0: Next generation architecture and CAD system for FPGAs. *TRETS*, 7(2):6, 2014.

[9] L. McMurchie and C. Ebeling. PathFinder: a negotiation-based performance-driven router for FPGAs. In *ACM/SIGDA FPGA*, 1995.

[10] Y. O. M. Moctar and P. Brisk. Parallel FPGA Routing based on the Operator Formulation. In *DAC*, pages 1–6. ACM, 2014.

[11] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial cad. *TRETS*, 8(2):10, 2015.

[12] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The tao of parallelism in algorithms. *ACM Sigplan Notices*, 46(6):12–25, 2011.

[13] M. Shen and G. Luo. Accelerate fpga routing with parallel recursive partitioning. In *ICCAD*, pages 118–125. IEEE, 2015.

[14] S. Yang. *Logic synthesis and optimization benchmarks user guide: version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991.