Serial-Equivalent Static and Dynamic Parallel Routing for FPGAs

Minghua Shen, *Member, IEEE*, Wentai Zhang, *Student Member, IEEE*, Guojie Luo, *Member, IEEE*, and Nong Xiao, *Senior Member, IEEE*

***Abstract*—Serial equivalency enables easier regression testing and customer support in production-grade parallel CAD tools. While existing parallel routing techniques have become sufficiently advanced to provide good speedup, support for serial equivalency still has been very limited or ignored because it was considered costly. In this paper, we present a serial-equivalent parallel router that not only provides significant speedup but also produces the same result as the serial router. This parallel router primarily leverages a dependency-aware scheduling algorithm to facilitate the serial equivalency. Moreover, regardless of how many processor cores are used, this scheduling algorithm also enables parallel router to have the same result as the serial router. In scheduling algorithm, according to the original net order of serial router, all of the nets are scheduled to a series of different stages. Specifically, the independent nets are scheduled to the same stage and they can be routed in parallel while the dependent nets are scheduled in different stages and they are processed in serial. Note that the parallel routing of independent nets can be explored in static and dynamic fashions, and the data synchronization between dependent stages is implemented in MPI-based message queue. Experimental evaluations using ten large designs from the academic VTR benchmark suite show that our parallel router can scale to 32 processor cores at least to provide an average 19.13× speedup compared to the state-of-the-art academic VPR router. And most importantly, our parallel router can maintain the serial equivalency which achieves the same results as the serial router. To the best of our knowledge, it is the first parallel router that provides significant speedup with a serial equivalency guarantee.**

***Index Terms*—Field-programmable gate array (FPGA), FPGA CAD, parallel routing, routing, serial equivalency.**

Manuscript received July 10, 2018; revised October 17, 2018; accepted November 30, 2018. Date of publication December 21, 2018; date of current version January 18, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003502, in part by the National Natural Science Foundation of China under Grant 61433019 and Grant 61802446, and in part by the Program for Guangdong Introducing Innovative and Entrepreneurial Teams under Grant 2016ZT06D211. This paper was recommended by Associate Editor P. Leong. *(Corresponding author: Minghua Shen.)*

M. Shen is with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510275, China, and also with the Key Laboratory of Machine Intelligence and Advanced Computing, Ministry of Education, Sun Yat-sen University, Guangzhou 510275, China (e-mail: shenmh6@mail.sysu.edu.cn).

W. Zhang and G. Luo are with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China.

N. Xiao is with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510275, China.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2887050

I. Introduction

In THE past 30 years, field-programmable gate arrays (FPGAs) have increased in speed by a factor of 100, meanwhile, cost and energy consumption per unit function have both decreased by more than a factor of 1000 [1]. These advancements are particularly promising which enables FPGAs to become the application accelerators deployed in datacenters. For example, Microsoft leverages the FPGAs to accelerate website search engine and software network functions [2], [3]. Unfortunately, the capacity of FPGA has alternately increased by more than a factor of 10 000 owing to the quantitative effects of Moore's Law. This change has also raised challenges to existing CAD compilation tools, which are used to synthesize the application designs onto underlying FPGA devices. Since the scale of the target application design increases, the compilation time of the CAD tool increases.

One possible direction to accelerate the compilation time of CAD tools is by using a faster processor. However, the failure of Dennard scaling has limited the maximum speed of a single processor. Considering that multicore processor has become prevalent today, parallelization has very strong attraction in many fields to address the runtime challenge. Specifically, parallel programs are preferred to be serial equivalency which must always give the same results as the serial version of the parallel algorithm. Meanwhile, this property is also emphasized by Altera for easier regression testing and customer support in industry [4]. This motivates the need to have a serial-equivalent parallel design implementation tool flow for FPGAs.

In FPGA CAD compilation flow, routing is probably the most tedious and time-consuming process. Obtaining a feasible routing solution is an NP-complete problem which finds a set of disjoint paths in a general graph. The heuristic PathFinder [6] algorithm works well in practice and it is also in the predominant use in academic research. This algorithm is usually divided into three nested iterations. In outermost iterations, it forces the nets to negotiate with each other to decide who will make a detour around the dependent routing resources, until all the dependencies are resolved to obtain a feasible solution. In middle iterations, it rips up an existing routing path and reroutes it by invoking maze expansion, which computes a path from the source node to each sink node in routing resource graph (RRG). In innermost iterations, the maze expansion employs the single source shortest path solver, which is usually implemented by Dijkstra's algorithm or A* search. However, there exist a dependency problem on routing a net one by one, this algorithm is serial in nature and it is nontrivial to perform the parallel routing of multiple nets for FPGAs.

Parallel routing for FPGAs has been studied extensively in the past few years. While existing work has witness the numerous efforts to explore the parallel routing, most of them have one or more of the following drawbacks which are not solved completely.

1) Quality of results (QoR) is a commonly used target in the modern routing study, because it directly affects maximum clock frequency and other design metrics, such as area, power, and routability. Some parallel routers [13], [14], however, have unacceptable degradation in the routing quality.
2) Determinism is very crucial in early design development and debugging, evidenced by vendors and users expect the same results to be produced each time when the routing tools are executed in parallel. However, some parallel routers [12], [15] cannot provide the deterministic results, where different results are produced each time the parallel router is executed. Thus, these parallel routers are impractical in an industrial context.
3) With the increasing logic capacities, scalability has become more important in modern parallel router. Several approaches [10], [11] try to partition the routing region into several subregions, the nets in their own respective subregions can be routed in parallel. However, some high fanout nets need to use some resources that distributed in other subregions and these nets cannot be routed in parallel. It is even worse that the number of such nets will increase when the number of partitions increases. Therefore, these parallel routers are not highly scalable.

Serial equivalency is an another indispensable feature to guarantee that parallel router always provides the same answer as the serial router. However, this property is rarely studied in prior parallel routers. Creating a faster and better router from scratch with equal capabilities and quality would be very difficult, and for this reason, we also explore the parallel router with a serial equivalency guarantee. Notably, serial equivalency is completely different from determinism, and it is an even stronger constraint we can apply to our proposed parallel router.

In this paper, we explore serial-equivalent multicore parallel routing for FPGAs. We leverage an optimal dependency-aware scheduling algorithm to guarantee serial equivalency of parallel router and explore the static and dynamic parallel techniques on multicore processor platform to expose a greater degree of parallelism. In our proposed parallel router, all of the nets are scheduled into several stages. The same stage has independent nets that can be routed in parallel while the different stages have dependent behaviors and they are processed in serial. Note that independent nets can be routed in static and dynamic parallelization, and data synchronization between dependent stages is implemented in MPI-based message queue. Our main contributions are summarized as follows.

1) To the best of our knowledge, it is the first serial-equivalent parallel router with a significant speedup guarantee for FPGAs.
2) We propose an optimal dependency-aware scheduling algorithm that enables the benefits of serial equivalency for parallel routing.
3) We explore the static and dynamic parallel routing with MPI-based messaging queue on multicore distributed-memory parallel platform.
4) We still provide significant speedup even with the constraint of serial equivalency, although it was considered expensive.

Experimental results using ten large designs from the academic VTR benchmark suite show that our parallel router can scale to 32 processing cores to provide an average 19.13× speedup and maintain serial-equivalent routing results comparing to the state-of-the-art academic VPR router.

The remaining part of this paper is organized as follows. In Section II, we describe the related work. In Section III, we give the motivation and overall design flow. In Section IV, we detail the scheduling algorithm. In Section V, we describe the static parallel approach. In Section VI, we present the dynamic parallel approach. In Section VII, we analyze the experimental results and evaluate the performance of the proposed parallel routing approaches. In Section VIII, we discuss the proposed parallel approaches and conclude this paper in Section IX.

II. Related Work

There is an abundance of work in the literature concerning FPGA routing, and we describe the most relevant works here, highlighting the elements that are beneficial for multicore parallel routing. Summaries include the works of process-level parallel routers [10], [11], [14]–[16] and thread-level parallel routers [12], [13], [17]. Specifically, the process-level parallel routers are based on MPI techniques while the thread-level parallel routers are based on mult*i*threading techniques.

Cabral *et al.* [10] partitioned entire RRG into several disjoint subgraphs and the nets in subgraph are routed in parallel. To generate the disjoint subgraphs, they require the FPGA architecture to have the disjoint switch box topology. Since the subgraphs are disjoint, there is no data synchronization between processor cores during parallel routing. As a result, they can achieve close to linear speedup for parallel routing. However, the disjoint switch topology is not suitable for modern FPGA architectures.

Gort and Anderson [11] partitioned all of the nets into different processor cores for parallel FPGA routing. Data synchronization among processor cores by using MPI-based message queue. Deterministic results are guaranteed by receiving the routing data in a blocking manner. However, this blocking scheme imposes expensive synchronization time and reduces the speedup. To improve the speedup, they consider the net dependency and only synchronize the dependent nets during parallel routing.

Moctar and Brisk [12] exploited speculative parallelism for mult*i*threaded parallel routing. They allow each thread to process multiple iterations at once and use locks to implement the thread-level synchronization. If a lock can be acquired, parallel runtime detects the dependency and rolls back one of the dependent activities. This requires having always a copy of the data before modifications. Unfortunately, this speculative parallelism can not guarantee the deterministic results.

Hoo *et al.* [13] formulated parallel routing as a linear programming problem and partition the problem into independent subproblems that can be solved in parallel. Parallel routing is done by using the Intel threading building block (TBB)

library. They can achieve good speedup when the overhead of task creation and context switching with Intel TBB are amortized. Unfortunately, such speedup will degrade the critical path delay dramatically.

Shen and Luo [14] employed recursive partitioning to parallelize the multinet routing for FPGAs. All the nets are partitioned into three subsets, where the subset containing dependent nets is routed in serial and the remaining two subsets containing independent nets are routed in parallel. This partitioning can be solved in dynamic programming. Unfortunately, their approach needs to take much longer wirelength for parallel routing.

Hoo and Kumar [15] exploited speculative parallelism and path encoding for parallel FPGA routing. They accelerate the routing in speculative parallelism and iteratively reduce the number of processes to guarantee the convergence of parallel router. The parallel routing time can be reduced by encoding the routing paths to sinks in a space-efficient manner. While they achieve good speedup, their parallel router can not produce the deterministic results.

Shen *et al.* [16] explored synchronous and asynchronous parallel routing for FPGAs. They enable asynchronous parallelism for parallel routing the independent nets to maintain good speedup without requiring communication overhead. They employ synchronous parallel routing for dependent nets to guarantee parallel router has same convergence to serial router. Moreover, they optimize the rip-up and reroute process to reduce the parallel routing time.

Hoo and Kumar [17] presented a parallel deterministic router based on spatial partitioning. To improve speedup, nets within a partition are scheduled to be routed in parallel. Multisink nets are decomposed into single-sink nets, and their bounding boxes are shrunk to increase the number of nets that can be routed in parallel. With these enhancements, they can reduce the routing time significantly.

In this paper, we focus on serial-equivalent parallel router and explore static and dynamic parallel approaches for FPGA routing.

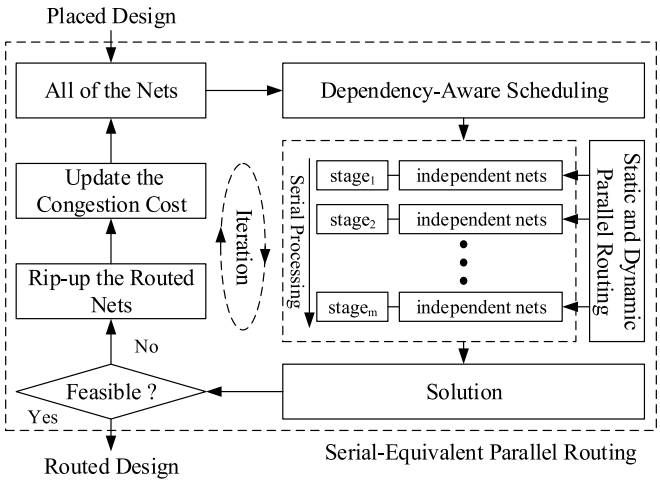

Fig. 1. Serial-equivalent parallel routing flow.

III. Motivation and Design Flow

A. Motivation

Parallelization has become a very attractive direction to reduce the routing time significantly. However, the dynamic scheduling of multiple threads or processes will result in different routing solutions across different runs. Serial-equivalent results across different runs and different platforms are crucial in a commercial context for three reasons.

1) When a bug is reported, we must be able to reproduce it. And different results to serial version will result in extremely difficult debugging, even if the bug is not caused by a parallel algorithm.
2) When we perform thousands of regression tests prior to each release of CAD tools, such as Quartus and Vivado, it would be extremely difficult to diagnose the failing tests whose results changed randomly.
3) When vendors and customers evaluate the performance of CAD tools and correctly use this product, it is impractical in customer support if parallel version can not produce the same result as the serial version.

There exists a latest parallel routing work [17] with emphasis on deterministic results. However, their parallel routing results are different from the result of serial version when running on machines with various numbers of cores. It is also important to parallelize the routing across different platforms. Otherwise, we cannot benefit from a more powerful computing platform with more processor cores, or we will lose the benefits of serial equivalency as mentioned earlier. When a parallel router provides the serial-equivalent results across different runs and different platforms, it is obvious that this algorithm produces the deterministic routing results as well [9], because it will generate identical routing solutions as running on a single-core platform. Serial equivalency is a very strong constraint that seems difficult to satisfy without degrading the performance.

In this paper, we explore the serial equivalency of parallel routing techniques without any significant performance degradation. It is based on an optimal scheduling algorithm to maintain the serial equivalency. And it also relies on the bounding box of each net to determine the dependency between nets in routing parallelization techniques.

B. Overall Design Flow

The overall design flow of our serial-equivalent parallel router is shown in Fig. 1, and notably, we still adopt the negotiation-based rip-up and reroute framework [6] to iteratively reduce the routing congestion until the feasible solution finds. In this flow, each iteration has two major phases: 1) scheduling and 2) parallel routing. The scheduling phase is to assign all of the nets to a series of different stages according to the original net order. Specifically, the nets in the same stage are independent and the dependent nets are distributed in different stages for serial-equivalent parallel routing. The details of scheduling will be revealed in Section IV. The parallel routing phase enables the independent nets are routed in parallel and the dependent stages are processed in serial. Note that we explore the static and dynamic parallel routing of the independent nets whereby there are no clear Internet constraints or relationships in same stage. The implementation details will be described in Sections V and VI. Combining this two phases, we can effectively parallelize the multinet routing with a serial equivalency guarantee.

When the iteration proceeds, the net dependency will change due to the relative scarcity of routing resources, both wires and connection points. Thus, before each parallel routing iteration, we will reschedule the nets into different stages considering such change in order to obtain the large degree of parallelism and ensure parallel algorithm has the same number of iterations as the serial version. Moreover, all the routing resources are accessible by every net during parallelization. Since the serial equivalency is guaranteed by the scheduling algorithm, multinet parallelization is supported to accelerate the routing stages on multicore distributed-memory system.

IV. Dependency-Aware Scheduling

In this section, we analyze and formulate the problem of parallel routing with sequential equivalency and present a scheduling algorithm to guarantee such property. Notably, the scheduling algorithm is provably optimal.

A. Problem Statement

Serial router routes a net at a time and it depends heavily on the net order. Thus, in serial router, net order is a very important factor to the routing results [7] and moreover, the perturbation of net order will result in divergence [8]. To ensure that parallel router has the same convergence as the serial router and avoid the degradation in the routing quality, we need to maintain the original net order and route the nets in parallel only if these nets are independent and they do not affect the results of each other. Thereby, we have the following definitions.

Definition 1 (Net Order): The routing nets n_i are the key elements in the scheduling. The set $T = (n_1, n_2, n_3, \ldots, n_t)$ is the collection of all the nets, where the nets are scheduled by the increasing order as $n_1, n_2, n_3, \ldots, n_t$.

Note that the increasing order of parallel router is the same as the original net order of serial router. Parallel router with serial equivalency provides identical and deterministic results as the corresponding serial router. We now define the concept of serial equivalency in the following context of our parallel routing approach. Initially, the definition of scheduling stages, as the routing tasks, is introduced.

Definition 2 (Scheduling Stages): Given a serial order of the nets $T = (n_1, n_2, \ldots, n_t)$ to be routed, the parallel router schedules a routing iteration into a sequence of scheduling stages $M = (p_1, p_2, \ldots, p_m)$, where $\bigcup p_i = T$ and $i \neq j \iff p_i \cap p_j = \varnothing$.

Definition 3 (Serial Equivalency): Given the scheduling stages, the parallel router routes the nets in p_1 concurrently at the first stage, and then after synchronization, it routes the nets in p_2 at the second stage, and so on. We call this parallel router *Serial equivalency*, if the routing results are equivalent to the serial routing results, where the nets $n_1, n_2, \ldots, n_t$ are routed one by one.

We regard such parallel router serial equivalency because it achieves the same results as the serial router for various numbers of cores. Thus, it shares the benefits from serial equivalency as mentioned in Section III-A.

Considering that the nets in same stage are routed in parallel and the stages M are processed in serial, thus we will require analyzing the dependent relationships between different nets to validate the serial equivalency. It is very critical to determine the net independency before the routing in the next iteration. Here, we analyze the dependency between nets based on the previous iteration and consider net bounding box to enable an efficient analysis. For each net n_i, we consider the unique bounding box b_i, artificially restricting the maze expansion of net n_i. For each bounding box b_i, the width and height are w_i and h_i, and the lower-left cornet position is at (x_i, y_i). With this assumption, we introduce the independent concept.

Definition 4 (Independent Net): The nets in a stage p_k are independent, if the bounding box of every pair of nets in p_k have no overlap, i.e.,

$$\begin{aligned}&\forall b_i, b_j \in p_k\\&(x_i + w_i < x_j) \vee (y_i + h_i < y_j)\\&\quad \vee (x_j + w_j < x_i) \vee (y_j + h_j < y_i).\end{aligned}$$

It is easy to see that if the nets in the same stage are independent, we can route these independent nets in parallel and still generate the same results as in a serial routing. By restricting every net in its bounding box, the net dependency can be efficiently analyzed before the next rerouting.

According to the concepts and assumptions above, it is natural to introduce the conflict graph $G'(V', E')$.

Definition 5 (Conflict Graph): Given the conflict graph $G'(V', E')$, V' represents the set of all the nets. A directed edge $e'_{ij} \in E'$ represents the dependency between nets n_i and n_j, while n_i must be scheduled before n_j to maintain serial equivalency.

Instead of directly minimizing the total parallel routing time, we minimize the number of stages to increase the degree of parallelism and reduce the synchronization overhead of parallel processes on multicore system. Based on the above definitions, we can formulate the serial-equivalent parallel routing problem as a scheduling problem.

Problem: Given a conflict graph $G'(V', E')$ of all the nets, the objective is to find a scheduling $M = (p_1, p_2, \ldots, p_m)$, such that the number of stages m is minimized.

Here, a stage p_k is a collection of nets, and all the nets in p_k are independent in the conflict graph G'.

B. Scheduling Algorithm

Net bounding box and its expansion approaches are widely explored to improve the serial routers. Also, they can provide the potential to determine the dependency between nets when we perform the parallel routing of multiple nets at each iteration. In original serial router, the initial bounding box is slightly larger than the minimum bounding box enclosing the terminal pins of routed net. The inability to find a legal routing path within the current bounding box results in expansion of the current bounding box, and then the net is rerouted again in expanded bounding box during the next iteration. This default box expansion enables each box to be expanded in four directions and the expanded size is set to one unit by default. This enlightens us to refer whether the boxes are overlap to determine the net dependency.

According to the box size of previous iteration and the default box expansion, we have the ability to calculate the size of expanded box at each iteration, and further determine the dependencies between nets in the current iteration. And thus we can schedule the nets into multiple stages to explore serial-equivalent parallel routing. Note that we follow the original

Algorithm 1 Scheduling Algorithm

Require: $G' = (V', E')$
Ensure: M
1: $M \Leftarrow ()$
2: $K \Leftarrow 1$
3: **while** $|V'| > 0$ **do**
4: $p_K \Leftarrow \varnothing$
5: **for** every node n_i in V' **do**
6: **if** in-degree of n_i is zero **then**
7: $p_K \Leftarrow p_K \cup \{n_i\}$
8: **end if**
9: **end for**
10: **for** every element n_u in p_K **do**
11: **for** every edge e'_{uv} **do**
12: $E' \Leftarrow E' - \{e'_{uv}\}$
13: **end for**
14: **end for**
15: $V' \Leftarrow V' - p_K$
16: M.append(p_K)
17: $K \Leftarrow K + 1$
18: **end while**

increasing net to avoid the quality degradation and to maintain the same convergence as the serial router. Therefore, we are inspired to focus on the net order to explore dependency-aware scheduling algorithm for the nets.

In a sense that the goal of this scheduling is to find a set $M = (p_1, p_2, \ldots, p_m)$ of V' from the conflict graph G', where the elements in the same stage p_k are disconnected with each other. Every stage p_k is a dependent collection, and we aim to schedule the conflict graph G' into dependent stages as few as possible.

According to the Definition 5, we provide two observations as follows.

1) Conflict graph G' is directed, and $\forall n_i, n_j \in p_k, e'_{ij} \notin E'$.
2) If $n_i \in p_k$, $n_j \in p_l$, and $e'_{ij} \in E'$, we have $i < j$ and $k < l$ (a necessary condition for serial equivalency).

These observations inspire us to develop a topological-like traversal to solve the scheduling problem, and its pseudocode is shown in Algorithm 1.

In this algorithm, every vertex and edge are accessed once and only once. Thus, the algorithm has a complexity of $O(|V'| + |E'|)$ as a conventional traversal flow. The cost to print the solution is at least $O(|V'|)$, and to read the dependency E' is $O(|E'|)$. Therefore, it is an asymptotically optimal algorithm.

In the following demonstrations, we will prove that M is the optimal solution with the minimum m. Obviously, this solution is legal because no two elements in p_k are connected according to lines 5–9 in Algorithm 1. Every batches p_K of nodes have no in-edge, so that $\forall n_i, n_j \in p_k, e'_{ij} \notin E'$.

On the other hand, while no edges inside one stage p_k, there must exist edges between two consecutive stages p_k and p_{k+1}, which is Theorem 1.

Theorem 1: $\forall n_v \in p_{k+1}, \exists n_u \in p_k$, such that $e'_{uv} \in E'$.

Proof (Proof by Contradiction): Suppose $\nexists n_u \in p_k$ with e'_{uv}, and it means that $n_v \in p_{k+1}$ has a zero in-degree while we process p_k. Thus, n_v will be lifted into p_k instead of p_{k+1}. This contradicts $n_v \in p_{k+1}$. ■

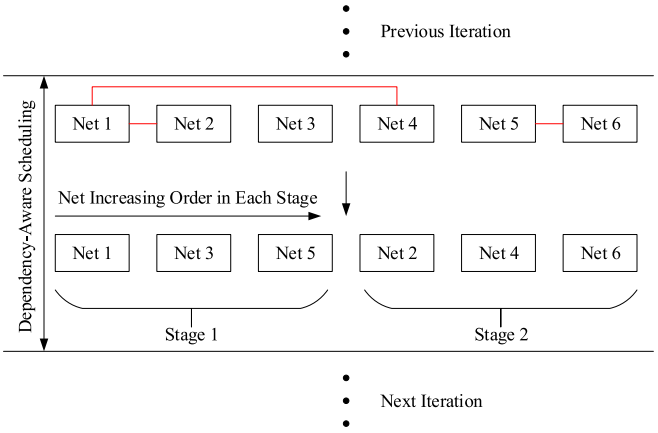

Fig. 2. Dependency-aware scheduling. Red line denotes that there is a conflict between nets.

When we can find a path from vertex a to vertex b, we denote $a \rightarrow b$. From Theorem 1, we induce a corollary, which is obvious.

Corollary 1: In any solution $P = (p_1, p_2, \ldots, p_m)$ given by Algorithm 1, there exists a path $n_{k_1} \rightarrow n_{k_m}$ of length m, consisting of the vertices $\{n_{k_1}, n_{k_2}, \ldots n_{k_m}\}$ with $n_{k_t} \in p_t$.

Theorem 2: The number of stages m in M calculated from Algorithm 1 is minimum.

Proof (Proof by Contradiction): Suppose there exist a better solution $M' = \{p'_1, p'_2, p'_3, \ldots, p'_{m'}\}$ with $m' < m$.

From Corollary 1, we have already found a path $n_{k_1} \rightarrow n_{k_m}$ with vertices $\{n_{k_1}, n_{k_2}, n_{k_3}, \ldots n_{k_m}\}$ in the solution M. For any element n_k on this path, we denote $p'_{q(k)}$ as the stage that it belongs to in the better solution M'. According to observation 1 and 2, we have $q(k_1) < q(k_2) < \cdots < q(k_m)$. By Pigeonhole principle, since $m' < m$, there exist two elements n_{k_u} and n_{k_v} such that $q(n_{k_u}) = q(n_{k_v})$. It contradicts the previous inequality. ■

Fig. 2 shows an example of dependency-aware scheduling for serial equivalency. According to the net dependent state of previous iteration and the default net bounding box expansion, we can determine the dependency between nets in the next iteration. Thus, the independent nets can be collected and their parallel execution will not influence on the routing results when comparing to the serial router. And this collection process is also successful because the scheduling algorithm can efficiently and precisely determine the dependency between the nets in each iteration. By using net scheduling, we start to route the first stage of independent nets concurrently and then route the next stage until all the stages are processed. Moreover, the data synchronization among dependent stages is implemented in MPI-based message queue. It is evident that each stage can be routed in parallel without affecting the routing results, because it benefits the net dependent detection during routing iteration.

V. Static Parallel Routing

In this section, we present a static parallel routing approach for FPGAs. Combined with dependency-aware scheduling in static parallel router, the independent nets in same stage can be parallelized in static fashion and the dependent stages are

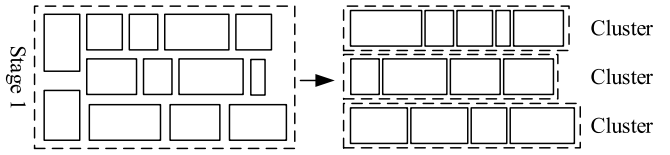

Fig. 3. Load balanced partitioning of multiple independent nets in same stage.

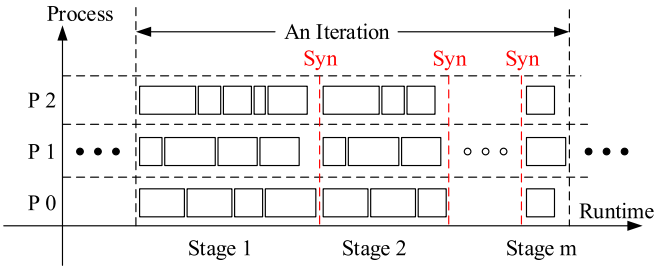

Fig. 4. Multinet static parallel routing on multicore distributed platform.

processed in serial to produce the serial equivalent routing results as the serial router.

A. Load Balanced Partitioning

While the independent nets of same stages can be routed in parallel, how to partition the nets into different processor cores for significant speedup is a nontrivial problem. As with any parallel program, it is desirable to reduce the stall time between different instances on the multicore processor systems. In our static parallel router, this goal translates into balancing the workloads between processor cores.

To enable the workloads between processor cores having a good balance, we require to estimate the runtime needed to route a single net. We consider the number of nodes visited in routing a single net in the previous iteration for predicting a single-net routing time based on two experimental observations.

1) During rip-upping and rerouting a single net, most of the nodes visited in the previous iteration will be reused in the next iteration. Analysis shows that the reuse rate of new routing path can up to 85% nodes of the original routing path.
2) Both bounding box and number of sinks are two other metrics for comparisons. Number of sinks is also used as the prediction metric in [14]. We speculate that a net with a large bounding box has a long distance between sinks, implying a long routing time. Analysis shows that number of explored nodes is the best metric and can lead to the lowest amount of stall time.

Therefore, it is useful to predict the runtime of routing a single net by using the number of nodes explored in the prior iteration. And therefore, we use the number of explored nodes as a load balancing metric to partition the nets into processor cores.

Fig. 3 shows the process of load balanced partitioning for multiple nets located in same stage. The independent nets of same stage are partitioned into different clusters, where the number of clusters is the same to the number of processor cores. Partitioning into net clusters is finished such that the sum of the number of explored nodes for the nets in each cluster is approximately equal. Each cluster is assigned to one processor core. Note that the number of explored nodes is no available in the first iteration and thus, we assign each cluster an equal number of nets in that iteration.

B. Multinet Static Parallel Routing

MPI is widely used to develop the distributed parallel programs because it enables separate MPI processes to communicate with one another. In distributed parallel platform, we focus on the multinet parallel routing approach. Specifically, with partitioning in each stage, the nets of each stage can be assigned into the different clusters, each of which can be routed concurrently with separate MPI process provided by a corresponding processor core.

Fig. 4 shows the multinet parallel routing process that the independent nets of each stage are partitioned into the corresponding MPI processes. Each MPI process only routes its own cluster of nets and maintain its own data structures, including routing path and associated congestion cost information. When the nets of first stage are routed in parallel, each MPI process needs to use MPI messages to communicate the intermediate results with other MPI processes and to synchronize their own respective results of overall routing state.

MPI have a mechanism that all of the MPI processes are invoked simultaneously. When dependent tasks are performed in parallel, synchronization seems to be particularly important. During our parallelization, synchronization is to determine which routing resources have been occupied by other nets and must avoid sharing the same resources. Because the dependent nets are distributed in different stages, we must synchronize the congestion information before routing the next stage such that this parallel router converges a feasible solution.

Since we only perform the parallel routing of independent nets in same stage and synchronize the dependent information between different stages, the total routing time is reduced with the similar convergence as the serial router. Mostly important, with dependency-aware scheduling, this static parallel router is able to give the same routing results as the serial version of parallel router.

VI. Dynamic Parallel Routing

In this section, we explore dynamic parallel routing for independent nets in a stage. Notably, this approach is different from the static parallel routing approach and it can obtain better speedup under the constraints of serial equivalency.

A. Dynamic Parallel Model

To fully exploit the multicore processor, parallel routing in dynamic fashion is expected to provide a greater degree of parallelism, where the parallel routing of multiple independent nets can utilize multiple processor cores at the same time. This is a foundation to provide an improvement about the available parallel performance over traditional single-core processor.

We consider the same stage of nets $S = \{s_1, s_2, \ldots, s_a\}$ to be scheduled to a multicore processor $C = \{c_1, c_2, \ldots, c_b\}$ for dynamic parallel routing, where a is the number of independent nets in the stage and b is the number of available processor cores. Given a net s_i, we have (r_i, d_i), where r_i is

the worst-case CPU time to route a net s_i and d_i is the CPU time range[1] to route the net s_i. Since router has the iterative feature and two experimental observations in Section V-A, we are encourage to select the routing time of previous iteration to represent the routing time of current iteration for each net in parallel routing. Thus, we can calculate the worst-case execution time including a start time and an end time to route a net. Further we can calculate the time range (deadline) to route a net on a processor core when performing the parallel routing on a multicore processor system. According to these available results, the utilization to route a net s_i on a processor core can be calculated by $k_i = r_i/d_i$. Thus, the total utilization of the stage S on multicore system can be calculated by $K_{\text{tot}} = \sum_{i=1}^{a} k_i$. Note that the total utilization must be smaller than the number of processor cores ($K_{\text{tot}} \leq b$) to obtain a feasible scheduling solution[2] for all the independent nets on multicore processor systems.

In addition, the nets assigned to a processor core c_i is denoted by P_i and given a net to the assignment of processor core, the utilization of processor core c_i is represented by $K_i = \sum_{s_e \in P_i} k_e$ $(1 \leq e \leq a)$. A net is considered to be schedulable for multicore processor if all of its instances finish no smaller than their time deadlines to route a net.

B. Quantifiable Load Balancing

Load balancing is very important to provide good speedup for dynamic parallel routing on multicore processor systems. The goal of load balancing is to evenly assign the nets such that every processor core has the same amount of nets. By balancing the nets between processor cores, computing resources provided by multicore systems can be efficiently used without over-provisioning and wasting potential.

We quantify the load balancing between processor cores by using the coefficient of variation, which is defined as the ratio of the standard deviation of the workload between the processor cores to the average workload. The imbalanced measure can describe the deviation of the current load balancing scheme with regard to a perfect balance. Note that a perfect balance indicates a perfectly uniform distribution. The standard deviation of an assignment is denoted by $\sigma = (1/b)\sqrt{(K_i - \mu)^2}$, where $\mu = \sum_{i=1}^{b} K_i/b$ and it means the utilization value of core. Note that a higher value of load imbalance indicates a lower effective load balancing scheme. In dynamic parallel routing, load balance is always used to guide the assignment of multiple nets toward multicore processor systems.

Lemma 1: There exists load imbalance between processor cores if there are two cores c_i and c_j such that $k_e < K_i - K_j$, where $s_e \in P_i$

C. Multinet Dynamic Parallel Routing

Fig. 5 shows the independent nets of the same stage can be scheduled to multicore processor system for dynamic parallel routing. This dynamic parallel approach consists of two steps: 1) partitioning and 2) scheduling. In partitioning, each net is assigned to a processor core, where the net is able to satisfy its deadline to complete the net routing. In scheduling, the assigned nets are scheduled within the time line of each processor core. Note that partitioning occurs before parallel runtime and scheduling occurs during parallel runtime. The per-core scheduling algorithm adopts the earlier deadline first arithmetic by default and in this paper, we only focus on the partitioning approach of multiple nets.

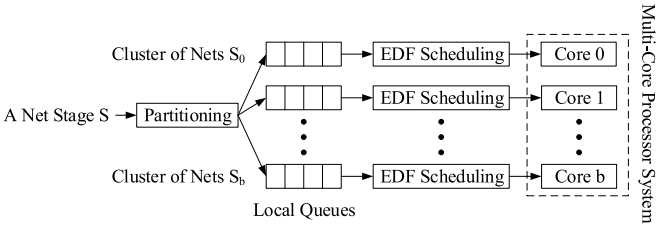

Fig. 5. Independent nets are partitioned and scheduled to multiple processor cores for dynamic parallel routing.

Given a schedulability test in the multicore processor system, our partitioning problem is to assign the nets to multiple cores effectively and this is a variation of the bin packing problem which is NP-complete. There have several heuristic algorithms to solve this problem, such as next-fit, first-fit, best-fit, and worst-fit. In general, first-fit has good schedulability but poor load balancing between multiple processor cores, while worst-fit has good load balancing but relatively poor schedulability when comparing to first-fit. We propose a new partitioning approach that not only provides a good schedulability but also maintain a good load balancing when routing the nets between processor cores in multicore parallel system.

The proposed approach, named load balancing-aware partitioning (LBAP), is implemented in the partitioning step of Fig. 5. And the proposed LBAP consists of three critical steps as follows.

1) *Sorting:* Given a stage, all the independent nets need to be sorted in a order of decreasing utilization.
2) *Partitioning:* With this sorted order about independent nets, each net is assigned into the first processor core, where it can be routed on the processor core while satisfying the constraints of all deadlines.
3) *Repartitioning:* Using the order of increasing utilization to sort the nets, each net is reassigned into a processor core with the minimum utilization.

The repartitioning step begins only when a feasible scheduling solution can be generated in the partitioning step. Moreover, the repartitioning step has a load balancing test and repartitioning needs to be continue until further reassignment cannot implement a improvement in terms of load balancing. Note that at the repartitioning step, it is unnecessary to retest the schedulability for each net routing on multicore processor system. This is because a new scheduling solution generated by the repartitioning step is feasible as well based on our theorem (see Theorem 3).

In partitioning step, we adopt first-fit decreasing utilization (FFDU) to obtain good schedulability. Employing FFDU to finish a single net routing will not miss deadline at each processor core and this is a basic requirement in multicore parallel system. If FFDU cannot generate a feasible scheduling solution for the given stage of independent nets, it returns

[1]We define that the time range is equal to the deadline of routing a net on a processor core.

[2]Nets are referred to be schedulable if they are scheduled to be routed on a processor core while satisfying their deadline constraints.

a value used to indicate that the partitioning of independent nets is failed.[3] If FFDU partitioning cannot generate a feasible scheduling solution, we also cannot perform the repartitioning step.

In repartitioning step, we adopt worst-fit increasing utilization (WFIU) to obtain good load balancing. Note that the nets are reassigned in order of increasing utilization and this is in contrast to the partitioning step. This order is used to maintain a feasible scheduling solution without retesting the schedulability, although a new solution generated by repartitioning may be different from the scheduling solution generated by FFDU. Moreover, using the WFIU only may has weaker schedulability and load balancing when comparing to worst-fit decreasing utilization that has good load balancing. But in this paper, WFIU is based on FFDU assignment and as a repartitioning approach and thus, it can provide good schedulability and load balancing and reduce the overheads of retesting the schedulability. The repartitioning stops when there is no additional net reassignment to improve load balancing.

Lemma 2: Given a current assignment and there is a single net considered for repartitioning, the WFIU-based repartitioning stops if the utilization of the single net routing on a processor core is equal to or larger than $K_{\max} - K_{\min}$.

Proof: When considering a single net for repartitioning at the multicore processor systems, we are based on the current assignment to calculate the minimum and maximum utilization values of processor cores, $K_{\min}$ and $K_{\max}$. This current assignment is different from FFDU assignment due to that the assignment is changed in the repartitioning process. Based on Lemma 1, the maximum utilization of a potential net for repartitioning will be smaller than the difference of the minimum and maximum utilization values of processor cores. If the utilization of the potential net for repartitioning is equal to or larger than the difference of the utilization values, the repartitioning process stops. ■

Since the nets considering repartitioning are sorted in the order of increasing utilization, this process can stop when the above condition is satisfied without requiring to further explore repartitioning of nets to reduce the load imbalance between processor cores. In fact, if a single net with a smaller utilization on multicore processor system is difficult to implement any improvement in dynamic parallel routing, there is no chance to make any improvement in load balancing for any net with the larger utilization than the utilization of current net considered for repartitioning.

Theorem 3: The proposed LBAP can generate a feasible scheduling solution if FFDU can generate a feasible scheduling solution.

Proof: Given a feasible scheduling solution generated by FFDU assignment, the utilization of each processor core c_i is less than or equal to 1 in order to make all the nets meet all the deadline constraints. The net s_e that considers for repartitioning is currently assigned to processor core c_i and thus its utilization is less than or equal to 1 since all the deadline constraints are satisfied. Thus, we have

$$K_i \leq 1, \text{ where } s_e \in P_i. \tag{1}$$

[3]In general, heuristic FFDU can generate a feasible scheduling solution and if it fails in the experiments, we will adopt static parallel approach for routing.

Assuming that there is another processor core c_j with the minimum utilization value and it is used as a target for a net s_e used to reassignment, thus the utilization of the processor core c_j is less than 1 since all the given assignment satisfies all deadline constraints. Thus, the utilization value of the processor core c_j that considers for repartitioning is not equal to 1 which forms

$$K_j < 1, \text{ where } c_j \text{ subject to } \min_{c_e \in C} K_e. \tag{2}$$

Assuming that the utilization of net s_e is smaller than the difference between its currently assigned core's utilization and the minimum core's utilization. Then the assignment is regarded to be load imbalance based on Lemma 1

$$k_e < K_i - K_j, \text{ where } s_e \in P_i. \tag{3}$$

In this situation, LBAP has least utilization by removing the net s_e from core c_i and reassigning the net s_e to core c_j. Thus, we have

$$K_j = K_j + k_e < K_i. \tag{4}$$

Although the utilization of core c_j is increased when adding the utilization value of the net s_e, its value is still smaller than 1 as follows, according to (1), (3), and (4):

$$K_j < 1. \tag{5}$$

Also, the utilization value of core c_i is smaller than 1 since the utilization value of the net s_e is reduced. Note that the utilization of the net s_e is k_e and its value is larger than 0. Meanwhile, the modified utilization value of processor core c_i are not equal to 1 as well

$$K_i = K_i - k_e < 1, \text{ where } K_i \neq 1 \text{ and } k_e > 0. \tag{6}$$

From (5) and (6), the utilization value of these two processor cores are smaller than 1 and thus their assignment still satisfies deadline constraints after repartitioning a net. According to Lemma 2, the repartitioning process stops when the next net s_e that considered for reassignment satisfies the following equation: $k_e \geq (K_{\max} - K_{\min})$. This is conflicted with (3). During repartitioning, all of the above equations are satisfied and thus, if FFDU assignment meets all the deadline constraints, the proposed LBAP assignment satisfies all the deadlines as well. ■

VII. Experimental Evaluation

The proposed serial-equivalent parallel routing approach is implemented in C++ programming language, as well as it has been incorporated into VTR 7.0 CAD flow [18]. We use the large-scale VTR 7.0 benchmarks to evaluate the effectiveness of our proposed parallel router. Notably, the VTR benchmark suite is very popular in parallel routing research. Experiments were performed on Linux servers, where each node has a 8-core Intel Xeon processor operating at 2.2 GHz and 32 GB memory. We run our parallel router using 2, 4, 8, 16, and 32 processor cores, and use two nodes to provide 16 processor cores and use four nodes to provide 32 processor cores.

Table I shows a summary of ten largest VTR benchmarks routed by serial VPR 7.0 router. Notably, most of the previous parallel routers also chooses the VPR 7.0 router as the baseline for comparisons. Moreover, this academic VPR 7.0 router is faster than commercial router [5]. The serial VPR 7.0 router

TABLE I
SUMMARY OF RESULTS USING VPR 7.0 ROUTER ACROSS TEN LARGEST VTR BENCHMARKS

Name	Nets	Width	Iteration	Wirelength	Time(s)
blob_mer.	6606	68	16	119927	544.73
mkSMAd.	7154	56	15	108553	360.03
mkPKtM.	7474	52	15	109980	275.73
or1200	8078	68	16	133856	858.66
stereov.0	9312	96	9	115870	217.62
stereov.1	13523	154	9	199814	840.54
LU8PEE.	16278	160	12	426520	2170.24
bgm	27853	116	11	152096	1843.27
stereov.2	36479	182	11	702836	14159
mcml	81282	196	11	1542736	29640.2

has two versions, one is routability-driven router and the other is timing-driven router. The former optimizes the total routed wirelength and the latter optimizes the critical path delay. In this paper, we only parallelize the routability-driven router and evaluate the total routed wirelength and available speedup. Note that our proposed parallel approach is also suitable for timing-driven router since these two serial routers have the same data structure and algorithmic flow.

To analyze the impacts of scheduling on achieved speedup and routed quality of the proposed static and dynamic parallel routing approaches, we first implement a simple parallel router (SiPaRo) for comparisons. This SiPaRo is static and employs the same partitioning and parallelization methods described in Section V. In SiPaRo, all of the nets are partitioned into clusters, each of which is assigned to the corresponding processor core for parallel routing. Each processor core routes a net and then synchronizes with other processor core and then routes the next net, and so on, until to finish the parallel routing.

Note that to analyze the experimental results, the static parallel routing based on dependency-aware scheduling is denoted by StPaRo-DAS and, the dynamic parallel routing based on dependency-aware scheduling is denoted by DyPaRo-DAS.

A. Routing Time and Speedup

In the first experiment, we evaluate the three proposed parallel techniques (SiPaRo, StPaRo-DAS, DyPaRo-DAS) for FPGA routing, respectively. We primarily study the execution time and available speedup when leveraging these parallel techniques to accelerate the original routing process.

Table II shows the execution time and achieved speedup of SiPaRo with the increasing number of processor cores, comparing to the original serial VPR 7.0 router. On average, SiPaRo achieves about 1.22×, 1.86×, 3.23×, 5.08×, and 6.49× speedups using 2, 4, 8, 16, and 32 processor cores, respectively. These results denote that SiPaRo is practicable parallel approach to accelerate the routing time for FPGAs on multicore processor systems. Please note that since all of the nets are not scheduled to independent and dependent nets, when parallelizing the routing of multiple nets, SiPaRo needs to frequently perform the data synchronization between processor cores to avoid to the congestions of routing resources, further converging a feasible routing solution. Since the frequent net-level synchronization overhead is very expensive, SiPaRo cannot provide a high degree of parallelism in parallel routing and its scalable speedups is very low. Moreover, without dependency-aware scheduling, SiPaRo does not have the serial equivalency and thus, it cannot provide the same result as the serial router.

Table III describes the execution time and achieved speedup of StPaRo-DAS with the different number of processor cores. On average, speedups of about 1.56×, 2.83×, 5.31×, 9.26×, and 16.58× can be achieved using 2, 4, 8, 16, and 32 processor cores, respectively. These results denote that StPaRo-DAS is a more feasible parallel routing approach and it is faster than the previous SiPaRo approach in terms of achieved average speedups. StPaRo-DAS exploits dependency-aware scheduling algorithm to enable all of the nets to be scheduled to several stages. In StPaRo-DAS, the independent nets are scheduled to the same stage and the dependent nets are scheduled to the different stages. Thus, StPaRo-DAS performs the parallel routing in same stage and performs the serial processing for different stages due to that there is a dependent behavior between different stages. Therefore, StPaRo-DAS enables the independent nets to be routed in parallel and minimizes the number of stages to reduce the expensive synchronization overheads between processor cores. Benefiting from the net-dependency-aware scheduling algorithm, the achieved speedup of StPaRo-DAS is improved significantly. When adopting 32 processor cores, StPaRo-DAS is about 2.5 times faster than SiPaRo. In addition, it is a pity that the ideal speedup is difficult to obtain in StPaRo-DAS because the dependent stages are processed in serial and the synchronization overhead is still exist in parallel routing. And nevertheless, our StPaRo-DAS can scale to 32 processor cores at least, leading to about 17× speedup for FPGA routing. Moreover, StPaRo-DAS has serial equivalency and produces the same result as the serial router.

Table IV reports the routing time and achieved speedup of DyPaRo-DAS using different number of processor cores. Considering that there is a dependent behavior on the scheduled stages, the data synchronization must be performed to ensure that DyPaRo-DAS has the same number of iterations as the serial router to find a feasible routing solution. It is difficult to optimize the synchronization overheads in DyPaRo-DAS under the constraints of serial equivalency and convergency. Therefore, we consider the parallel optimization and explore dynamic parallel routing to the independent nets of same stage on multicore processor system. Benefiting from the dynamic parallel routing approach, our DyPaRo-DAS can provide about 1.75×, 3.48×, 6.39×, 11.20×, and 19.13× speedups on average using 2, 4, 8, 16, and 32 processor cores, respectively. In terms of 32 processor cores, there are about 1.13× and 2.95× improvements over the proposed StPaRo-DAS and SiPaRo approaches, respectively. These improves come from the implementation of dynamic parallel routing for independent nets. Therefore, we conclude that DyPaRo-DAS has the ability to expose a higher degree of parallelism than the above StPaRo-DAS approach, further enabling the highly scalable parallel routing on multicore processor system, especially with the increasing number of processor cores. Benefiting from dependency-aware scheduling, DyPaRo-DAS also has serial equivalency, resulting in the same routing result as the serial router.

From Tables II–IV, we summarize the proposed parallel routing techniques and present some intuitive observations. From the horizontal point of view, we observe that the achieved speedup increases significantly with the increasing number of processor cores, especially for the proposed

TABLE II
ROUTING TIME OF SIPARO ACROSS TEN LARGEST BENCHMARKS

Summary	The speedups of SiPaRo using 2, 4, 8, 16, and 32 processor cores.									
Number	2-core		4-core		8-core		16-core		32-core	
Name	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup
blob_mer.	555.85	0.98	425.57	1.28	302.63	1.80	222.34	2.45	162.61	3.35
mkSMAd.	404.53	0.89	333.36	1.08	235.31	1.53	164.40	2.19	131.88	2.73
mkPKtM.	270.32	1.02	195.55	1.41	104.84	2.63	68.76	4.01	56.62	4.87
or1200	692.47	1.24	451.93	1.90	243.25	3.53	147.03	5.84	121.28	7.08
stereov.0	211.28	1.03	132.70	1.64	76.90	2.83	47.10	4.62	43.44	5.01
stereov.1	661.84	1.27	469.58	1.79	285.90	2.94	167.44	5.02	144.67	5.81
LU8PEE.	1400.15	1.55	851.07	2.55	475.93	4.56	284.06	7.64	199.10	10.90
bgm	1245.45	1.48	808.45	2.28	445.23	4.14	255.30	7.22	180.54	10.21
stereov.2	10335.04	1.37	6585.58	2.15	3575.51	3.96	2760.04	5.13	2407.99	5.88
mcml	21478.41	1.38	11856.08	2.50	6736.41	4.40	4463.89	6.64	3275.16	9.05
Average	—	1.22	—	1.86	—	3.23	—	5.08	—	6.49

TABLE III
ROUTING TIME OF STPARO-DAS ACROSS TEN LARGEST BENCHMARKS

Summary	The speedups of StPaRo-DAS using 2, 4, 8, 16, and 32 processor cores.									
Number	2-core		4-core		8-core		16-core		32-core	
Name	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup
blob_mer.	412.67	1.32	242.10	2.25	140.39	3.88	73.31	7.43	41.93	12.99
mkSMAd.	292.71	1.23	175.62	2.05	99.73	3.61	57.15	6.30	33.55	10.73
mkPKtM.	202.74	1.36	115.85	2.38	58.54	4.71	34.04	8.10	19.24	14.33
or1200	543.46	1.58	299.18	2.87	153.06	5.61	86.47	9.93	48.73	17.62
stereov.0	158.85	1.37	83.38	2.61	44.32	4.91	24.99	8.71	13.99	15.55
stereov.1	522.07	1.61	304.54	2.76	167.44	5.02	92.27	9.11	51.41	16.35
LU8PEE.	1148.28	1.89	616.55	3.52	326.84	6.64	185.02	11.73	101.22	21.44
bgm	1012.79	1.82	567.16	3.25	296.35	6.22	162.98	11.31	88.83	20.75
stereov.2	8280.12	1.71	4538.14	3.12	2344.21	6.04	1535.68	9.22	862.30	16.42
mcml	17232.67	1.72	8541.84	3.47	4574.10	6.48	2762.37	10.73	1513.03	19.59
Average	—	1.56	—	2.83	—	5.31	—	9.26	—	16.58

TABLE IV
ROUTING TIME OF DYPARO-DAS ACROSS TEN LARGEST BENCHMARKS

Summary	The speedups of DyPaRo-DAS using 2, 4, 8, 16, and 32 processor cores.									
Number	2-core		4-core		8-core		16-core		32-core	
Name	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup	Runtime	Speedup
blob_mer.	358.38	1.52	204.02	2.67	115.16	4.73	65.63	8.30	39.70	13.72
mkSMAd.	223.62	1.61	115.39	3.12	69.91	5.15	40.32	8.93	24.61	14.63
mkPKtM.	162.19	1.70	78.33	3.52	42.62	6.47	27.09	10.18	15.93	17.31
or1200	485.12	1.77	230.20	3.73	120.43	7.13	77.08	11.14	43.72	19.64
stereov.0	129.54	1.68	66.15	3.29	35.97	6.05	20.30	10.72	12.39	17.57
stereov.1	480.31	1.75	241.53	3.48	127.16	6.61	68.06	12.35	41.10	20.45
LU8PEE.	1136.25	1.91	563.70	3.85	300.17	7.23	157.84	13.75	92.51	23.46
bgm	985.71	1.87	502.25	3.67	270.67	6.81	151.34	12.18	84.79	21.74
stereov.2	7822.65	1.81	3999.72	3.54	2135.60	6.63	1208.11	11.72	670.41	21.12
mcml	16021.73	1.85	7619.59	3.89	4192.39	7.07	2335.71	12.69	1371.60	21.61
Average	—	1.75	—	3.48	—	6.39	—	11.20	—	19.13

DyPaRo-DAS approach. From the vertical point of view, we observe that the achieved speedup continues to increase when increasing the size of the application designs, and we can provide a better speedup when performing the parallel routing of the larger application designs. It is obvious that there is a design named stereov.2 having a slightly large speedup when comparing with other larger designs. Our analysis points out that this design is relative congestion due to the difference of its own architecture. But the overall looks like, we route the larger design to achieve larger speedup based on the proposed parallel techniques.

B. Total Routed Wirelength

In the next experiment, we evaluate the quality results generated by the proposed parallel routers. Note that with the results, we also evaluate the serial equivalency of the proposed parallel routers. Consider that the StPaRo-DAS and DyPaRo-DAS approaches are based on dependency-aware scheduling algorithm, thus they have serial equivalency and they have the same results as the serial router. Here, we only evaluate the StPaRo-DAS results, as well as the SiPaRo results.

Table V presents the QoR between SiPaRo and DyPaRo-DAS compared to serial VPR 7.0 router regarding the total routed wirelength. No difference of total routed wirelength between serial VPR router and the proposed DyPaRo-DAS parallel router is observed for all circuit designs. This is because DyPaRo-DAS schedules all the nets into a series of different stages according to the original net order of serial router. This enables the independent nets to be located in same stage and the dependent nets to be distributed in different stages. The former encourages us to perform the parallel

TABLE V
SUMMARY OF QUALITIES BETWEEN SIPARO AND DYPARO-DAS ACROSS TEN LARGEST BENCHMARKS

Summary	Impacts on the total routed wirelength.									
Name	SiPaRo					DyPaRo-DAS				
Circuit	2-core	4-core	8-core	16-core	32-core	2-core	4-core	8-core	16-core	32-core
blob_mer.	120121	120356	120573	121275	122708	119927	119927	119927	119927	119927
mkSMAd.	108614	108943	109230	109815	110952	108553	108553	108553	108553	108553
mkPKtM.	110256	110612	110877	111540	112815	109980	109980	109980	109980	109980
or1200	134275	134561	134653	135418	137564	133856	133856	133856	133856	133856
stereov.0	116173	116425	116718	117339	118853	115870	115870	115870	115870	115870
stereov.1	200197	200715	200953	201692	204581	199814	199814	199814	199814	199814
LU8PEE.	427154	427837	428652	431523	438730	426520	426520	426520	426520	426520
bgm	152463	152917	153142	154228	155119	152096	152096	152096	152096	152096
stereov.2	704452	706074	708182	713765	724117	702836	702836	702836	702836	702836
mcml	1546841	1551652	1554517	1563751	1584876	1542736	1542736	1542736	1542736	1542736
Average	1.002	1.005	1.007	1.013	1.025	1.000	1.000	1.000	1.000	1.000

routing in same stage due to that there is no dependent behavior between nets. The latter enlightens us to perform the serial processing for different stages due to that there exist a dependent behavior between different stages. This guarantees that our DyPaRo-DAS parallel router has serial equivalency which generates the same results as the serial router.

The SiPaRo parallel router does not schedule all the nets into independent and dependent nets for parallel routing exploration. Thus, it cannot generate the same results as the serial router. It means that SiPaRo does not have the serial equivalency. Moreover, the SiPaRo parallel router does not maintain the original net order, resulting in a slightly higher total routed wirelength than the serial router.

From the results of total routed wirelength, we conclude that dependency-aware scheduling enables DyPaRo-DAS to have serial equivalency, thereby generating the same routing results as the serial router. Notably, this is first work to implement serial-equivalent parallel results with significant speedup for FPGA routing. In summary, the proposed scheduling algorithm not only can provide significant speedup but also maintain serial equivalency of parallel router.

C. Scalability With Titan Benchmark Suite

In the third experiment, we only evaluate the scalability of the proposed DyPaRo-DAS parallel router. This is due to that DyPaRo-DAS has a better speedup and it is much faster than the proposed SiPaRo and StPaRo-DAS parallel routers. We conduct the experiments using the large-scale and heterogeneous Titan benchmark suite [5]. Table VI shows the details of ten benchmarks used for evaluations in our scalable experiments. We still select the VTR 7.0 CAD tool to synthesize these application designs adopting Altera's Stratix IV FPGA architecture. Note that all of the designs are routed by original VPR 7.0 router which forms our baseline for comparisons.

Fig. 6 shows the available speedups of the DyPaRo-DAS parallel router when routing the ten heterogeneous Titan designs using 2, 4, 8, 16, and 32 processor cores. When adding the number of processor cores, DyPaRo-DAS still has a good speedup to route the heterogeneous designs. Note that with the increasing size of application designs, DyPaRo-DAS has also the ability to continue to provide a significant speedup. It is obvious that our DyPaRo-DAS has the potential to route the very-large-scale application design to provide a good speedup when adding the number of processor cores. In terms of using 32 processor cores, our DyPaRo-DAS can achieve about

TABLE VI
DETAILS OF TEN HETEROGENEOUS TITAN BENCHMARKS

Design Name	Abbr.	Nets	Blocks	LUTs	FFs
sparcT1_core	SC	92673	92060	42257	45272
neuron	NE	105541	86619	24413	59822
stereo_vision	SV	124670	93617	38206	51258
des90	DE	125831	108791	62070	30244
cholesky_mc	CM	127399	105998	27209	74051
SLAM_spheric	SS	149334	127872	108849	18342
segmentation	SE	201709	167001	153487	7477
bitonic_mesh	BM	210367	188767	107277	49570
dart	DA	223411	201708	103069	87386
openCV	OP	258030	212207	108042	86529

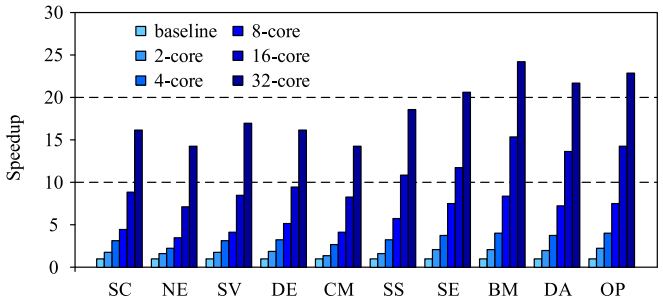

Fig. 6. Available speedups of the DyPaRo-DAS parallel router when routing the heterogeneous Titan designs using 2, 4, 8, 16, and 32 processor cores.

$15\times \sim 25\times$ speedups. Thus, we believe that DyPaRo-DAS has the ability to provide the highly scalable parallel routing for very-large-scale and heterogeneous FPGA designs.

In addition, benefiting from dependency-aware scheduling algorithm, our DyPaRo-DAS still generates the same results as the serial router when routing these Titan designs. With scheduling and dynamic parallel approaches, we accelerate the serial router for large-scale Titan designs on multicore processor systems enabling the significant speedup and a serial equivalency guarantee. Thus, it is great meaningful to integrate our proposed parallel routing approach into the commercial CAD compilation tools.

D. Comparisons With State-of-the-Art Parallel Router

We compare the proposed parallel routers with the state-of-the-art parallel router ParaDro [17] in terms of average speedup. Note that we do not emphasize the difference of other factors, such as benchmarks and experimental platforms, but focus on the general observations based on the parallel

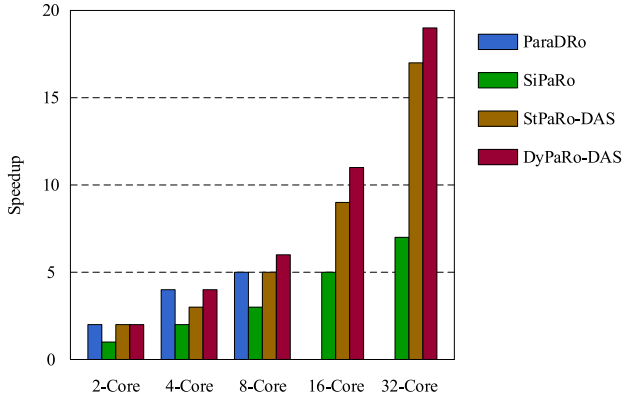

Fig. 7. Comparisons between the state-of-the-art ParaDRo and the proposed parallel routers.

approaches and the available speedups. This comparison fashion is the same as the previous parallel routers, including the state-of-the-art ParaDro.

Fig. 7 shows the normalized speedups of ParaDRo and our parallel routers. Our parallel routers can scale to 32 processor cores at least and our DyPaRo-DAS can provide about 19× speedup. ParaDRo achieves a maximum speedup of about 5× with 8 processor cores. In terms of the maximum speedup, our DyPaRo-DAS is about four times faster than the ParaDRo and thus, our parallel approach is very suitable to accelerate the FPGA routing. In addition, it can be seen from Fig. 7 that with the increasing number of processor cores, the achieved speedup is improved significantly, especially for the proposed DyPaRo-DAS parallel router. Therefore, combining the scheduling and dynamic parallel approaches is able to provide a significant speedup which enables the highly scalable parallel routing for FPGAs. Moreover, our DyPaRo-DAS is serial equivalency able to give exactly the same results as the corresponding serial router. ParaDRo has deterministic parallel routing results but this result is different from the serial router. ParaDRo can be repaired to fulfill the serial-equivalent requirement but at the cost of speedup [17].

From the comparisons, we conclude that our DyPaRo-DAS parallel router can achieve much more speedup than the state-of-the-art ParaDRo router. And most importantly, our DyPaRo-DAS has the serial equivalency which achieves the same results as the serial router. These conclusions demonstrate the effectiveness of our scheduling algorithm and dynamic parallel approach. This further shows the efficiency and effectiveness of our serial-equivalent parallel router.

VIII. Discussion and Possible Limitation

The experimental results show that our parallel router can be a desirable choice in the context of FPGA CAD compilation tools. Dependency-aware scheduling algorithm enables our parallel router to generate the same results as the original serial router, further implementing the serial equivalency of parallel router. Static parallel approach demonstrates the effectiveness of our parallel router which provides a good speedup and serial-equivalent results. Dynamic parallel approach enables our parallel router to provide the significant speedup with the different number of processor cores comparing with the serial router, further demonstrating the high scalability of parallel router. When comparing with the state-of-the-art parallel router, our parallel router also performs better in terms of available speedup and QoR. In addition, our parallel router has serial equivalency which produces the same results as the serial router. It is convincing that our parallel router is an attractive choice for FPGA routing.

There is a possible limitation about our serial-equivalent parallel router running on the multicore processor systems. To maintain the serial equivalency of parallel router, we require to generate the conflict graph based on the dependencies between different nets. The complexity of conflict graph generation is $O(N^2)$, where N is the total number of nets. With the number of nets become large, the time to generate the conflict graph is a possible limitation in parallel routing. But in practice, it is very fast to generate the conflict graph due to that the total number of nets is very small for existing application designs including large-scale Titan designs. Actually, the time of conflict graph generation can be ignored when comparing to the routing time. Moreover, this limitation is trivial probably and an easy and intuitive way is to partition the conflict graph into several subgraphs for parallel routing. We believe that our serial-equivalent parallel router will be more powerful than the previous parallel routers and it is particularly suitable for FPGA routing.

IX. Conclusion

Serial equivalency is a very important requirement in parallel CAD tools. In this paper, we attempt to explore serial-equivalent parallel routing for FPGAs. We primarily employ a dependency-aware scheduling algorithm to make all the nets are scheduled into several stages. The same stage consists of independent nets and can be routed in parallel while the different stages are processed in serial. Note that we adopt dynamic parallel routing for independent nets to expose the large degree of parallelism. Experimental evaluations using ten large designs from the academic VTR benchmark suite show that our serial equivalent parallel router can achieve about 19× speedup on average using 32 processing cores. This is the first work to maintain serial-equivalent parallel routing with significant speedup for FPGAs.

Acknowledgment

The authors would like to thank the insightful comments and feedbacks from anonymous reviewers.

References

[1] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proc. IEEE* , vol. 103, no. 3, pp. 318–331, Mar. 2015.

[2] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2016, pp. 1–13.

[3] B. Li *et al.*, "ClickNP: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2016, pp. 1–14.

[4] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *Proc. Int. Symp. Field Program. Gate Arrays (FPGA)*, 2008, pp. 14–23.

[5] K. Murray *et al.*, "Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial CAD," *J. ACM Trans. Reconfig. Technol. Syst.*, vol. 8, no. 2, Apr. 2015, Art. no. 10.

[6] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *Proc. Int. Symp. Field Program. Gate Arrays (FPGA)*, 1995, pp. 111–117.

[7] R. Y. Rubin and A. M. Dehon, "Timing-driven pathfinder pathology and remediation: Quantifying and reducing delay noise in VPR-pathfinder," in *Proc. Int. Symp. Field Program. Gate Arrays (FPGA)*, 2011, pp. 173–176.
[8] P. K. Chan, M. D. F. Schlag, C. Ebeling, and L. McMurchie, "Distributed-memory parallel routing for field-programmable gate arrays," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 8, pp. 850–862, Aug. 2000.
[9] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proc. USENIX Conf. Hot Topics Parallelism (HotPar)*, 2009, p. 4.
[10] L. Cabral, J. S. Aude, and N. Maculan, "TDR: A distributed-memory parallel routing algorithm for FPGAs," in *Proc. IEEE Int. Conf. Field Program. Logic Appl. (FPL)*, 2002, pp. 263–270.
[11] M. Gort and J. H. Anderson, "Accelerating FPGA routing through parallelization and engineering enhancements special section on PAR-CAD 2010," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 1, pp. 61–74, Jan. 2012.
[12] Y. O. M. Moctar and P. Brisk, "Parallel FPGA routing based on the operator formulation," in *Proc. 45th Annu. Design Autom. Conf. (DAC)*, San Francisco, CA, USA, 2014, pp. 1–6.
[13] C. H. Hoo, A. Kumar, and Y. Ha, "ParaLaR: A parallel FPGA router based on Lagrangian relaxation," in *Proc. IEEE Int. Conf. Field Programmable Logic Appl. (FPL)*, London, U.K., 2015, pp. 1–6.
[14] M. Shen and G. Luo, "Accelerate FPGA routing with parallel recursive partitioning," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, 2015, pp. 118–125.
[15] C. Hoo and A. Kumar, "ParaDiMe: A distributed memory FPGA router based on speculative parallelism and path encoding," in *Proc. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2017, pp. 172–179.
[16] M. Shen, N. Xiao, and G. Luo, "A coordinated synchronous and asynchronous parallel routing approach for FPGAs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Irvine, CA, USA, 2017, pp. 577–584.
[17] C. H. Hoo and A. Kumar, "ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling," in *Proc. Int. Symp. Field Program. Gate Arrays (FPGA)*, 2018, pp. 67–76.
[18] J. Luu *et al.*, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *J. ACM Trans. Reconfig. Technol. Syst.*, vol. 7, no. 2, Jun. 2014, Art. no. 6.

Minghua Shen (M'18) received the Ph.D. degree in computer science from Peking University, Beijing, China, in 2017.

He is currently an Associate Researcher with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China. His current research interests include FPGA synthesis, heterogeneous and parallel computing, and cyber-physical systems.

Dr. Shen is a member of ACM.

Wentai Zhang (S'16) received the B.S. degree from Peking University, Beijing, China, in 2014, where he is currently pursuing the Ph.D. degree with the Center for Energy-Efficient Computing and Applications.

He has published papers in several conferences, such as ICCAD and DATE. His current research interests include electronic design automation, heterogeneous accelerators, and medical imaging analytics.

Guojie Luo (M'12) received the B.S. degree in computer science from Peking University, Beijing, China, in 2005 and the M.S. and Ph.D. degrees in computer science from the University of California at Los Angeles, Los Angeles, CA, USA, in 2008 and 2011, respectively.

He is currently an Associate Professor with the School of EECS, Peking University. His current research interests include electronic design automation, heterogeneous computing with FPGAs and emerging devices, and medical imaging analytics.

Dr. Luo was a recipient of the 2013 ACM SIGDA Outstanding Ph.D. Dissertation Award in Electronic Design Automation and the ten-year Retrospective Most Influential Paper Award at ASPDAC 2017. He is a member of ACM.

Nong Xiao (SM'18) received the B.S. and Ph.D. degrees in computer science from the College of Computer, National University of Defense Technology, Changsha, China, in 1990 and 1996, respectively.

He is currently a Professor with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China. He has over 160 publications to his credit in journals and international conferences, including the IEEE TRANSACTIONS ON SERVICES COMPUTING, the IEEE TRANSACTIONS ON MULTIMEDIA, the *Journal of Parallel and Distributed Computing*, the *Journal of Computer Science and Technology*, HPCA, ICCAD, MIDDLEWARE, MSST, IPDPS, CLUSTER, SYSTOR, and MASCOTS. His current research interests include network parallel computing, large-scale storage system, and computer architecture.

Dr. Xiao is a member of ACM.