Towards Serial-Equivalent Multi-Core Parallel Routing for FPGAs

Minghua Shen and Nong Xiao
School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
Email: {shenmh6|xiaon6}@mail.sysu.edu.cn

***Abstract*—In this paper, we present a serial-equivalent parallel router for FPGAs on modern multi-core processors. We are based on the inherent net order of serial router to schedule all the nets into a series of stages, where the non-conflicting nets are scheduled in same stage and the conflicting nets are scheduled in different stages. We explore the parallel routing of non-conflicting nets on multi-core processors for a significant speedup. We perform the data synchronization of conflicting stages using MPI-based message queue for a feasible routing solution. Note that load balance is always used to guide the multi-core parallel routing. Experimental results show that our parallel router provides about 19.13× speedup on average using 32 processor cores comparing to the serial router. Notably, our parallel router generates exactly the same wirelength as the serial router satisfying serial equivalency.**

I. Introduction and Motivation

FPGA is a reconfigurable circuit able to be programmed by a designer. Programming an application onto FPGAs depends heavily on EDA tools. Routing is one of the most critical steps in FPGA EDA flow and the existing routers always take a very long time to complete, significantly hindering designer productivity [1]. Multi-core processor is very prevalent today enabling parallel router to be explored to accelerate the routing time. It is non-trivial to parallelize the routing as it exhibits an inherent sequential net processing order [2].

There have been several recent efforts in exploring parallel routing for FPGAs [3], [4], [5], [6]. However, these parallel routers do not have the serial equivalency, which cannot generate the identical results as the serial router. Parallel programs are preferred to be serial equivalency, which must always give exactly the same results as the serial version of the algorithm [7], [8]. This feature has been emphasized by Altera for easier regression testing and customer support in parallel EDA tools. It is a regret that support for the serial equivalency is very limited or ignored as it was considered costly in parallel EDA tools. In addition, serial equivalency is obtained at the cost of speedup, especially for parallel routers [6].

Serial equivalency is very crucial in the production-grade parallel EDA tools for three reasons. (1) When a bug is reported by parallel tools, we must have the ability to reproduce it. It is extremely difficult to debug the different results between parallel and serial programs, especially that the parallel program does not cause this bug. (2) When we perform thousands of regression tests prior to each release of EDA 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 EDA tools and use this product, it is impractical in customer support if parallel version cannot generate the same result as the serial version.

There exists a latest parallel router [6] with emphasis on parallel determinism. However, their parallel results are different from the results of serial router when running on a multi-core processor platform. It is important to parallelize the routing on the platforms with various numbers of cores. Otherwise, we cannot benefit from a more powerful computing platform with more processor cores, or we will lose the benefits of serial equivalency. In addition, when a parallel router has serial-equivalency, it is obvious that this parallel router has determinism as well [9], because it will generate the identical solutions as running on a single-core processor platform. Serial equivalency is a very strong requirement and it is rarely studied in prior parallel routing works.

In this paper, we present a serial-equivalent parallel router for FPGAs running on multi-core processors. Benefiting from conflict-aware multi-net scheduling, our parallel router always generates exactly the same results as serial router, regardless of how many processor cores are used. Notably, our scheduling algorithm is provably optimal and it resorts to the bounding box of each net to determine the conflicts between nets in parallel routing. Further, load balance is always used to drive the parallel routing for significant speedup. Our contributions are summarized as follows:

- We propose a serial-equivalent multi-core parallel router that provides significant speedup generating the identical results as serial router.
- We propose a conflict-aware scheduling algorithm that enables the benefits of serial equivalency for parallel routing.
- We propose a load balanced dynamic parallel approach that exposes a high degree of parallelism for parallel routing when running on multi-core processors.
- We present experimental evaluations adopting commonly used benchmark suite to demonstrate the effectiveness of our proposed parallel router.

Experimental results show that our parallel router scales to 32 processor cores to provide about 19.13× performance speedup on average, comparing to the state-of-the-art serial VPR 7.0 router. Further, our parallel router always generates the identical wirelength results as serial router satisfying serial

978-3-9819263-4-7/DATE20/©2020 EDAA

equivalency. To our knowledge, it is the first work emphasizing on serial-equivalent parallel routing with a significant speedup guarantee for FPGA routing.

II. Preliminaries

The net order is a critical factor in FPGA routing [10], [11] and for serial equivalency, we are based on the inherent net order of serial router and give the following definitions.

Definition 1. (net order). 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 $n_1, n_2, n_3, \ldots, n_t$.

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

Definition 3. (serial equivalency). For the given 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 parallel routing results are equivalent to the serial routing results, where the nets $n_1, n_2, ..., n_t$ are routed one by one in serial router.

Scheduling the nets into a series of stages for serial-equivalent parallel routing depends heavily on the dependencies between different nets. Thus we have the concept of independent net.

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 has no overlap, i.e.,

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

Note that each net n_i has an unique bounding box b_i and 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). 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.

Definition 5. (conflict graph). For 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 minimizing total parallel routing time, our objective is to minimize the total number of stages to increase the degree of parallelism and reduce the synchronization overheads between cores. Based on the definitions above, we formulate the scheduling problem for serial-equivalent parallel routing on multi-core processors.

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

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

III. Methodologies

A. Conflict-Aware Scheduling

According to the bounding box size of previous iteration and the default box expansion, we can calculate the size of expanded box at each iteration, further determining the dependencies between different nets in current iteration. We thus schedule the nets into multiple stages. Further according to the inherent net order of serial router, we consider the conflicts between nets to explore the scheduling of all the nets to obtain the serial-equivalent parallel routing results.

In a sense that the goal of this scheduling is to find a set $M = (p_1, p_2, ..., p_m)$ 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 conflicting collection, and we aim to schedule the conflict graph G' into conflicting stages as few as possible.

According to Definition 5, we have two observations:

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 propose Algorithm 1 to solve the scheduling problem.

Algorithm 1 The Scheduling Algorithm

```
Require: G' = (V', E')
Ensure: M
 1: M ⇐ ()
 2: K ⇐ 1
 3: while |V'| > 0 do
 4:     p_K ⇐ ∅
 5:     for every node n_i in V' do
 6:         if in-degree of n_i is zero then
 7:             p_K ⇐ p_K ∪ {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' ⇐ E' − {e'_uv}
13:         end for
14:     end for
15:     V' ⇐ V' − p_K
16:     M.append(p_K)
17:     K ⇐ K + 1
18: end while
```

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 conflicting 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 line 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 are 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 $n_v \in p_{k+1}$. □

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

COROLLARY 1. In any solution $P = (p_1, p_2, ..., p_m)$ given by Algorithm 1, there exists a path $n_{k_1} \to n_{k_m}$ 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 exists 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 $n_{k_1} \to n_{k_m}$ 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 $q(k_1) < q(k_2) < ... < q(k_m)$. 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. □

B. Parallel Routing Exploration

1) Dynamic Parallel Model: To fully exploit the multi-core 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.

We consider the same stage of nets $S = \{s_1, s_2, \ldots, s_a\}$ to be scheduled to a multi-core processor system $C = \{c_1, c_2, \ldots, c_b\}$ for dynamic parallel routing, where a is the number of independent nets in the stage and b is the total 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. As our parallel router has the iterative feature, we are encouraged to select the routing time of previous iteration to represent the routing time of current iteration for each net. Thus we can calculate the worst-case execution time including a start time and an end time to route a net. Then we calculate time range (deadline) to route a net on a processor core when performing the parallel routing on a multi-core processor system.

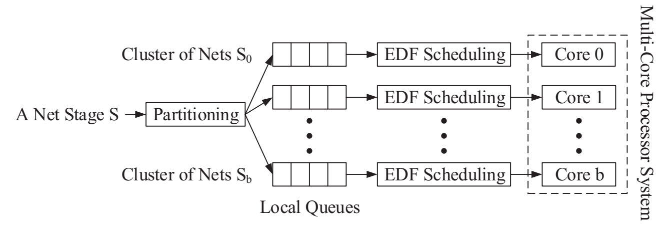

Fig. 1. The independent nets are partitioned and scheduled to multiple processor cores for dynamic parallel routing.

According to these values above, 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 multi-core system can be calculated by $K_{tot} = \sum_{i=1}^{a} k_i$. Note that the total utilization must be smaller than the number of processor cores ($K_{tot} \leq b$) to obtain a feasible scheduling solution[2] for all the independent nets on multi-core 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 multi-core processor if all of its instances finish no smaller than their time deadlines to route a net.

2) Quantifiable Load Balance: Load balance is to provide great speedup for parallel routing on multi-core processor systems. The goal of load balance is to evenly assign the nets so that every core has the same amount of nets. By balancing the nets between cores, computing resources provided by multi-core systems can be used efficiently.

We quantify the load balance 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 balance 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 $\sigma = \frac{1}{b}\sqrt{(K_i - \mu)^2}$, 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 balance scheme. In parallel routing, load balance is always used to guide the assignment of multiple nets towards multi-core processor systems.

3) Load Balanced Dynamic Parallel Routing: Fig. 1 shows the independent nets of the same stage are scheduled to multi-core processor system for dynamic parallel routing. This dynamic parallel approach consists of two steps: partitioning and scheduling. In partitioning, each net is assigned to a processor core where the net is able to satisfy its deadline

[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.

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 (EDF) by default and here we study the partitioning of multiple nets.

Given a schedulability test in a multi-core processor system, our partitioning problem is to assign the nets to multiple cores and this is a variation of the bin packing problem which is NP-complete. There are 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 balance between multiple processor cores, while worst-fit has good load balance but relatively poor schedulability when comparing to first-fit. We propose a new partitioning approach that not only provides a good schedulability but also maintains a good load balance when performing the parallel routing of independent nets between processor cores in multi-core parallel system.

The proposed approach, named load balancing-aware partitioning (LBAP), is implemented in the partitioning step of Fig. 1. And the LBAP consists of three critical steps:

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 for independent nets, each net is assigned to 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 balance test and repartitioning needs to continue until further reassignment cannot implement an improvement in terms of load balance. Note that at the repartitioning step, it is unnecessary to re-test the schedulability for each net routing. This is because a new scheduling solution generated by the repartitioning step is feasible.

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 core and this is a basic requirement in multi-core parallel system. If FFDU does not generate a feasible scheduling solution for the given stage of independent nets, it returns a value used to indicate the partitioning of independent nets is failed. Further we also cannot perform the repartitioning step.

In repartitioning step, we adopt worst-fit increasing utilization (WFIU) to obtain good load balance. 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 have weaker schedulability and load balance when comparing

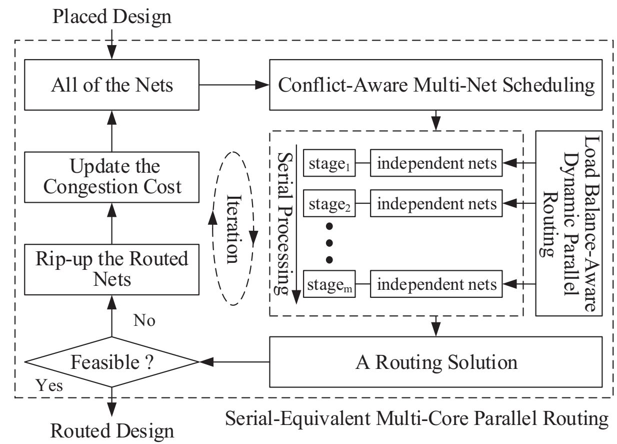

Fig. 2. The overall design of our parallel router.

to worst-fit decreasing utilization (WFDU) that has good load balance. But in this work, WFIU is based on FFDU assignment and as a repartitioning approach and thus, it can provide good schedulability and load balance and reduce the overheads of retesting the schedulability. The repartitioning stops when there is no additional net reassignment to improve load balance.

C. The Overall Design of Parallel Router

Our parallel router adopts the negotiation-based rip-up and re-route iterative engine [2] to progressively eliminate the resource congestions between different nets until finding a feasible routing solution. The overall design of our parallel router is shown in Fig. 2 and note that scheduling and parallelization are imposed in each iteration. In scheduling, we are based on inherent net order to schedule all nets to several stages so that the same stage has the non-conflicting (independent) nets and the conflicting (dependent) nets are distributed in different stages. The details of scheduling will be revealed in Section III-A. In parallelization, we perform load balance-aware dynamic parallel routing for each stage which consists of independent nets, and perform the synchronization between different stages using MPI-based message queue on a multi-core processor system. The implementation details will be described in Section III-B.

IV. Evaluation

We integrate the proposed parallel router into the commonly used VTR 7.0 CAD flow [12]. We use the large VTR 7.0 benchmarks to evaluate the effectiveness of our parallel router. We evaluate the experiments on Linux servers, each of which has a 8-core Intel Xeon processor with 2.2GHz and 32GB memory. We run our parallel router using 2, 4, 8, 16 and 32 cores, and use two nodes to provide 16 cores and use four nodes to provide 32 cores. Notably, most of the previous works also chooses the VPR 7.0 router as the baseline for comparisons. Moreover, this academic VPR 7.0 router is faster than commercial router [13].

To demonstrate the effectiveness of our proposed parallel router, we implement two static parallel routers, StPaRo and

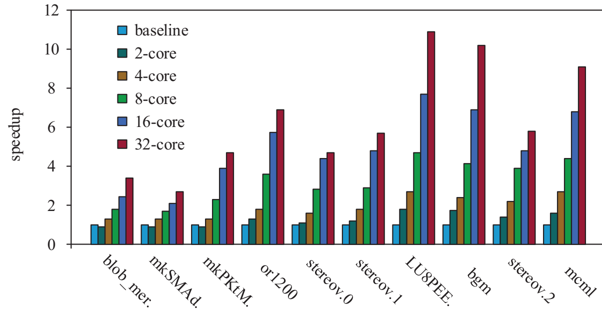

Fig. 3. Speedups of StPaRo with 2, 4, 8, 16, and 32 processor cores.

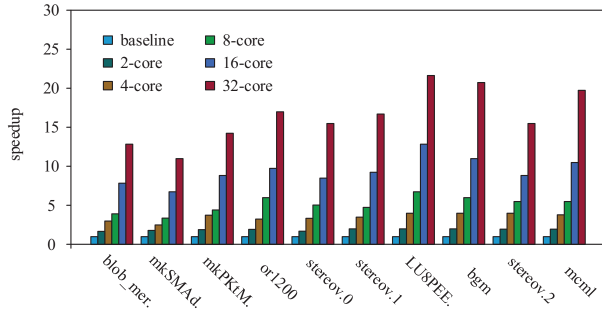

Fig. 4. Speedups of SE-StPaRo with 2, 4, 8, 16, and 32 processor cores.

SE-StPaRo. StPaRo is a static parallel router while SE-StPaRo is a serial-equivalent static parallel router, both of which are also performed on multi-core processors. The proposed serial-equivalent multi-core parallel router is dynamic and it is denoted by SE-DyPaRo.

In StPaRo, all of the nets are partitioned into subsets, each of which is assigned to its own processor core for parallel routing. Note that the number of subsets are equal to the number of processor cores. Each processor core routes a net and then synchronizes with other processor core using MPI-based message queue and then routes the next net, and so on, until completing the parallel routing. In SE-StPaRo, we first adopt conflict-aware scheduling algorithm mentioned in Section III-A to generate a series of stages and each stage has the independent nets. We then perform the parallel routing of independent nets using the same approach as the StPaRo. Note that in StPaRo and SE-StPaRo, we do not consider the multi-core processor scheduling problem, thus both of them are static parallel routers.

A. Experimental Results

Fig. 3 shows the speedups of StPaRo with the various number of processor cores. On average, StPaRo has 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 StPaRo is practicable multi-core parallel approach to accelerate the routing time. Please note that since all the nets are not scheduled to independent and dependent nets, when parallelizing the multi-net routing, StPaRo frequently performs synchronization to avoid to the congestions. Since synchronization overhead is very expensive, StPaRo cannot

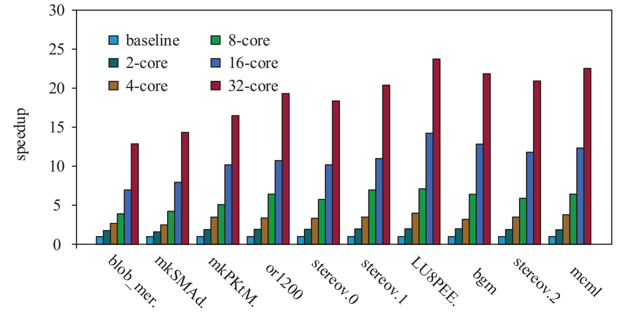

Fig. 5. Speedups of SE-DyPaRo with 2, 4, 8, 16, and 32 processor cores.

provide a high degree of parallelism and its scalable speedup is very low. Moreover, without conflict-aware scheduling, StPaRo does not have the serial equivalency and thus, it cannot provide the same result as the serial router.

Fig. 4 shows the speedups of SE-StPaRo. On average, speedups of 1.56×, 2.83×, 5.31×, 9.26×, and 16.58× are achieved using 2, 4, 8, 16, and 32 cores, respectively. This denotes that SE-StPaRo is a more feasible parallel routing approach and it is faster than StPaRo in terms of achieved speedups. SE-StPaRo exploits conflict-aware scheduling to enable all of the nets to be scheduled to several stages. In SE-StPaRo, the independent nets are scheduled to the same stage and the dependent nets are scheduled to the different stages. Thus SE-StPaRo performs the parallel routing in same stage and performs the serial processing for different stages due to that there is a dependent behaviour between different stages. Therefore, SE-StPaRo enables the independent nets to be routed in parallel and minimizes the number of stages to reduce the expensive synchronization overheads between cores. Benefiting from conflict-aware scheduling, the speedup of SE-StPaRo is improved significantly. Further SE-StPaRo has serial equivalency and generates exactly the same results as the serial router.

Fig. 5 shows the speedups of SE-DyPaRo. As there is a dependent behavior, synchronization between stages is used to ensure a feasible solution. It is difficult to optimize synchronization overheads under the constraints of convergency and serial equivalency. Thus, we consider parallel optimization and explore dynamic parallel routing for independent nets. Benefiting from dynamic parallel routing, SE-DyPaRo provides about 1.75×, 3.48×, 6.39×, 11.20×, and 19.13× speedups on average using 2, 4, 8, 16, and 32 cores, respectively. In terms of 32 cores, there are about 1.13× and 2.95× speedup improvements over SE-StPaRo and StPaRo, respectively. These improvements come from the implementation of dynamic parallel routing for independent nets. Therefore we conclude that SE-DyPaRo has the ability to expose a higher degree of parallelism than the above approaches. In addition, SE-DyPaRo also has serial equivalency generating exactly the same results as the serial router.

Table I shows the impacts of StPaRo, SE-StPaRo, and SE-DyPaRo on the total routed wirelength. There is no difference between serial router, SE-StPaRo, and SE-DyPaRo for all

TABLE I
SUMMARY OF QUALITIES BETWEEN STPARO, SE-STPARO, AND SE-DYPARO ACROSS TEN LARGE BENCHMARKS ($\times 10^5$).

Summary	Wirelength	Impacts on the total routed wirelength.														
Name	VPR Router	StPaRo					SE-StPaRo					Our Parallel Router SE-DyPaRo				
Circuit	Baseline	2-core	4-core	8-core	16-core	32-core	2-core	4-core	8-core	16-core	32-core	2-core	4-core	8-core	16-core	32-core
blob_mer.	1.199	1.201	1.203	1.205	1.212	1.227	1.199	1.199	1.199	1.199	1.199	1.199	1.199	1.199	1.199	1.199
mkSMAd.	1.085	1.086	1.089	1.092	1.098	1.109	1.085	1.085	1.085	1.085	1.085	1.085	1.085	1.085	1.085	1.085
mkPKtM.	1.099	1.102	1.106	1.108	1.115	1.128	1.099	1.099	1.099	1.099	1.099	1.099	1.099	1.099	1.099	1.099
or1200	1.338	1.342	1.345	1.346	1.354	1.375	1.338	1.338	1.338	1.338	1.338	1.338	1.338	1.338	1.338	1.338
stereov.0	1.158	1.161	1.164	1.167	1.173	1.188	1.158	1.158	1.158	1.158	1.158	1.158	1.158	1.158	1.158	1.158
stereov.1	1.998	2.001	2.007	2.009	2.016	2.045	1.998	1.998	1.998	1.998	1.998	1.998	1.998	1.998	1.998	1.998
LU8PEE.	4.265	4.271	4.278	4.286	4.315	4.387	4.265	4.265	4.265	4.265	4.265	4.265	4.265	4.265	4.265	4.265
bgm	1.520	1.524	1.529	1.531	1.542	1.551	1.520	1.520	1.520	1.520	1.520	1.520	1.520	1.520	1.520	1.520
stereov.2	7.028	7.044	7.060	7.081	7.137	7.241	7.028	7.028	7.028	7.028	7.028	7.028	7.028	7.028	7.028	7.028
mcml	15.42	15.46	15.51	15.54	15.63	15.84	15.42	15.42	15.42	15.42	15.42	15.42	15.42	15.42	15.42	15.42
Average	1.000	1.002	1.005	1.007	1.013	1.025	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

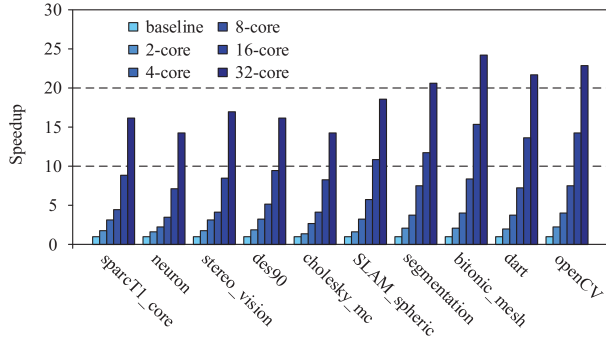

Fig. 6. The scalable speedups of our SE-DyPaRo parallel router when routing the heterogeneous Titan designs using 2, 4, 8, 16 and 32 processor cores.

circuit designs. This is because these two parallel routers schedule all the nets into a series of different stages according to the original net order of serial router. They enable 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 routing in same stage due to that there is no dependent behaviour between nets. The latter enlightens us to perform the serial processing for different stages due to that there exists a dependent behaviour between different stages. These guarantee that they have serial equivalency which generates the identical results as the serial router.

Fig. 6 shows the scalable speedups of the SE-DyPaRo 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, SE-DyPaRo still has a good speedup to route the heterogeneous designs. Note that with the increasing size of application designs, SE-DyPaRo has also the ability to continue to provide a significant speedup. It is obvious that our SE-DyPaRo 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 SE-DyPaRo can achieve about $15\times \sim 25\times$ speedups. Thus we believe that SE-DyPaRo has the ability to provide the highly-scalable parallel routing for very-large-scale and heterogeneous FPGA designs.

At last, we compare SE-DyPaRo with the state-of-the-art parallel router ParaDRo [6] in terms of maximum speedup and total wirelength. ParaDRo achieves about $5.4\times$ speedup on average using 8 processor cores with about 8% degradation on total wirelength comparing to the serial VPR 7.0 router. In addition, ParaDRo does not have serial equivalency and it satisfies serial equivalency at the cost of achieved speedup.

V. CONCLUSION

Serial equivalency enables easier regression testing and customer support in production-grade parallel EDA tools. In this paper, we propose a serial-equivalent multi-core parallel router for FPGAs. It is the first work to study serial-equivalent parallel routing for FPGAs.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive comments. This paper is supported by National Key Research and Development Program of China under grant No. 2018YFB1003502, National Natural Science Foundation of China under grant No. 61433019 and 61802446, and other grants from 2016ZT06D211, 2018B030312002, 19lgpy215.

REFERENCES

[1] S. Trimberger. *Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology.* in IEEE Journal, 2015.

[2] L. McMurchie et al. *Pathfinder: A negotiation-based performance-driven router for FPGAs.* in FPGA, 1995.

[3] Y. Moctar et al. *Parallel FPGA routing based on the operator formulation.* in DAC, 2014.

[4] M. Shen et al. *Accelerate FPGA routing with parallel recursive partitioning.* in ICCAD, 2015.

[5] M. Shen et al. *A coordinated synchronous and asynchronous parallel routing approach for FPGAs.* in ICCAD, 2017.

[6] C. Hoo et al. *ParaDRo: A parallel deterministic router based on spatial partitioning and scheduling.* in FPGA, 2018.

[7] J. Dennis et al. *Determinacy and repeatability of parallel program schemata.* in DFM, 2012.

[8] M. Song et al. *Realizing Reproducible and Reusable Parallel Floating Random Walk Solvers for Practical Usage.* in DATE, 2019.

[9] R. Bocchino et al. *Parallel programming must be deterministic by default.* in HotPar, 2009.

[10] R. Rubin et al. *Timing-driven pathfinder pathology and remediation:quantifying and reducing delays noise in VPR-pathfinder.* in FPGA, 2011.

[11] P. Chan et al. *Distributed-memory parallel routing for field-programmable gate arrays.* in TCAD, 2000.

[12] J. Rose et al. *VPR 7.0: Next generation architecture and CAD system for FPGAs.* in TRETS, 2014.

[13] K. Murray et al. *Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD.* in TRETS, 2015.