A Coordinated Synchronous and Asynchronous Parallel Routing Approach for FPGAs

Minghua Shen[1], Guojie Luo[2], and Nong Xiao[1]
School of Data and Computer Science, Sun Yat-sen University, China[1]
Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China[2]
Collaborative Innovation Center of High Performance Computing, NUDT, China[2]
Email: shenmh6@mail.sysu.edu.cn and gluo@pku.edu.cn

***Abstract*—Routing is a time-consuming process in the FPGA design flow. Parallelization is a promising direction to accelerate the routing. While synchronous parallelization can converge a feasible solution, the ideal speedup is rarely achieved due to excessive communication overheads. Asynchronous parallelization can provide an almost linear speedup, but it is difficult to converge in the limited number of iterations due to net dependency. In this paper we propose SAPRoute, which coordinates synchronous and asynchronous parallelism on distributed multiprocessing environment to accelerate the routing for FPGAs. The objective is to boost the more speedup of parallel routing algorithm under the requirement of convergence. To the best of our knowledge, this is the first work to study the impact of synchronization and asynchronization during parallelization. Experimental results show that our approach have negligible explicit synchronization overhead and achieves significant speedup improvement over a set of commonly used benchmarks. Notably, SAPRoute produces the speedup of 24.27× on average compared to the default serial solution.**

I. Introduction

FPGAs are increasingly popular for application-specific computing in datacenters, because they deliver high performance and energy efficiency compared to general purpose CPUs and GPUs, respectively. Moreover, FPGAs are an attractive design style that provides greater flexibility and shorter time-to-market than ASICs, because FPGAs can be modified to meet new requirements and have lower non-recurring engineering costs associated with manufacturing. However, as the density of FPGA design keep increasing, the associated CAD tool takes a very long time to synthesize the designs onto the underlying FPGAs device. This enlightens us to develop a fast and scalability design automation tool for FPGAs.

Routing is one of the most runtime-intensive steps in the FPGA design flow, which can take hours or days to complete a complex state-of-the-art design [1]. A promising direction to address the runtime challenge is to accelerate routing algorithm through parallel computing. Reducing the execution time of routing algorithm will improve the engineering productivity due to the shorter debug cycles. In addition, faster routing algorithm will increase the capacity of design space exploration. If routing is fast enough, it can also be integrated into the placement stage to provide more accurate timing information, which produces the better placement quality.

Routing is a complex NP complete problem that finds the disjoint paths in the graph to connect the pins of the source and sinks for each net, and whether a Quality of Results (QoR) goals are met. Routing a single net consists in assigning routing resources such that all the sinks are reachable from the source. When routing a set of nets in sequential, the order in which the nets are routed is critical since some routing resources needed by a net may be used by nets that are routed earlier [2]. Thus, the congestion avoidance mechanism is employed to resolve contention for routing resources.

Typical negotiation-based PathFinder algorithm [3] is the most versatile serial routing algorithm available in academic VPR framework [14]. Also, a variant of PathFinder is used in commercial FPGA CAD tools. The goal of PathFinder is to balance the performance and routability. PathFinder uses an iterative scheme that converges to a solution in which all nets are routed while achieving close to the optimal performance. Notice that routability is achieved by forcing nets to negotiate for a resource and thereby determine which net needs the resource most. In essence, nets negotiate with each other for which net keeps a shared routing resource. We explore the coarse-grained distributed parallel routing to improve the runtime in this iterative framework.

It is non-trivial to develop a parallel routing algorithm for FPGAs. This is because the dependency is always in a sequential routing process of several nets. Most of the previous attempts to exploits synchronous parallelism to accelerate the routing while maintaining the convergence and routing quality. However, they have lost sight of the fact that the vast communication overheads severely hinder the speedup of parallel routing algorithm, especially when we scale with more processor cores. Asynchronous parallelization may help overcome this obstacle, but it is difficult to converge a feasible solution with the limited number of iterations. These motivates the need to design a novel parallel router, which provides close to linear speedup and maintains the convergence.

In this paper we propose SAPRoute, a coordinated synchronous and asynchronous parallel approach to improve the speedup for FPGA routing. SAPRoute makes use of the dependency-aware rip-up and re-route based iterative scheme to eliminate the congestion between different nets until to find a feasible solution. In order to take full advantage of the parallelization, our design differs in multiple aspects from previous parallel routers in literature. The key contributions are summarized as follows:

978-1-5386-3093-8/17/$31.00 ©2017 IEEE

- We provide a quantitative study on the speedup and convergence of synchronous and asynchronous parallel routing algorithms.
- We propose a novel parallel router that coordinates the synchronous and asynchronous parallelism to provide significant speedup.
- We demonstrate promising improvements in speedup for a set of common benchmarks. Our source code will be publicly available on the authors' websites.

II. Background

A. Routing and Parallel Routing

The routing resources in an FPGA and their connections are represented by the directed graph. The set of vertices contains prefabricated wires in the FPGA architecture while the edges contains programmable switches that connect the wires together. There is a set of nets to be routed using the routing resources in the graph. The routing objective is to find a tree for each net such that all of the trees do not use the same routing resources and other requirements such as wirelength are satisfied.

The parallel routing is an extension of the serial routing problem. The parallel routing is also concerned with the speedup of serial routing algorithm when multiple process cores are available. A parallel routing algorithm generally divides the serial routing problem into several subproblems and conquers them concurrently. Ideally, every subproblem is the same size and there is no overhead, the speedup of parallel routing algorithm is equal to the number of processing elements. However, the ideal speedup is rarely obtained due to the overheads of synchronization and communication when executing the parallel routing algorithm. Moreover, the parallel routing algorithm is irregular [7].

Notice that since the amount of parallelism is determined by the input to the problem, it is *impossible* to determine the optimal scheduling at routing time to parallelize the nets.

B. Related Work

Existing parallel routing approaches can be classified into two categories. One is coarse-grained distributed-memory parallelization [4], [8], [6], [9], and the other is fine-grained shared-memory parallelization [10], [11], [12].

Chan and McMurchie [4] are first to leverage parallel techniques to accelerate the routing for FPGAs. They explore the update of congestion cost to improve the parallelism and ensure convergence. However, they do not guarantee the deterministic routing results due to the change of dependent net ordering. They provide the speedup of 2.5× with three processor cores.

Gort and Anderson [8] exploit the region-based partition to parallelize the net routing. Deterministic routing results are guaranteed by invoking the block version of MPI receive function. However, they are not scalable as evident in the diminishing speedup as the number of processors increases. This is due to the number of processors increases, the number of nets that across partitioning boundaries increases, requiring more inter-processor communication and reducing speedup. They achieve 2.8× speedup using eight processor cores.

Shen and Luo [6] accelerate the net routing using parallel recursive partitioning. They partition the nets into three subsets, where the first subset contains potentially conflicting nets, and the two remaining subsets consists of potentially conflicting-free nets. The latter two subsets are routed in parallel after routing the former subset. They achieve a speedup of up to 7.02× with 32 processes but has the same limitation as the parallel router [8].

Instead of directly partitioning, Cabral and Maculan [9] design a disjoint switch box topology, where the wires can only be connected to other wires on the same track. This restriction simplifies the parallel routing by allowing each processor to independently route a subset of nets. However, the disjoint architecture provides limited routability and is no longer used in state-of-the-art commercial FPGAs. As expected, they achieve an almost linear speedup because there is minimal communication overheads.

Moctar and Brisk [10] exploit the dynamic parallelism to parallelize the maze expansion for a single net routing. They leverage lock-based expansion operator and software transactional memory (STM) based priority queue to avoid the congestion in parallelization. However, the overhead of lock acquisition and rollback due to STM reduces speedup with the number of threads increases. They have a good speedup of 5.46× with eight threads, although they do not produce the deterministic routing results.

Hoo and Ha [11] develop a parallel FPGA router using Intel Threading Building Block techniques. They leverage a linear program to model the routing problem that can be decomposed into independent subproblem with Lagrangian relaxation. As a result, the nets can be parallelized, and good speedup of 7.05× is achieved. However, this method is only applied into the global routing problem and can not be extended to the detailed routing problem.

Shen and Luo [12] explore the GPU-accelerated parallel techniques for FPGA routing. They leverage both routing search space reduction and dynamic parallelism to accelerate the single net routing on GPU. Moreover, they also design an efficient multiple nets parallel routing method and achieve an average of 18.75× speedup with 3% degradation in routed wirelength.

While existing synchronous parallel works [4], [8], [6], [10], [12] have provided good speedup, none of these works focuses on the communication overhead reduction to increase the parallelism. In this paper, we explore the design of coarse-grained coordinated synchronous and asynchronous parallel routing on distributed multiprocessing environment.

III. Quantitative study

In this section we study the impact of synchronous and asynchronous on the convergence and speedup of parallel routing algorithms in the MPI-based parallel programming model. Our experimental methodology is to check whether the number of congestion nodes of parallel routing algorithm has

stopped decreasing monotonically in the iteration procedure, because this is a sign that the algorithm is having difficulty converging. More specifically, two different strategies are proposed to motivate the novel parallel router.

A. Asynchronous Parallel Routing

Asynchronous strategy can improve the efficiency of parallel execution by avoiding the communication overhead and synchronization cost between processor cores. This encourages us to have an asynchronous parallel implementation for FPGA routing.

Having partitioned the nets into subsets, the processes begin to route their respective net subsets. After a process routes a net, it do not synchronize the routing state and communicate with all of the other processes, and continues to route the next net until to finish the subset. Figure 1 shows that the nets are split with runtime load balancing between two processes and routed in asynchronous parallel at each iteration. Notice that we synchronize processes at the end of each iteration to ensure that all processes are working on the same iteration at the same time.

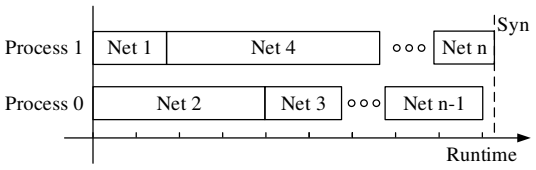


Fig. 1. Asynchronous parallel routing.

With the iteration proceeds, this approach continues to route the nets until the configurable maximum number of iterations[1] or a congestion free state is reached. At the meanwhile, the number of congested nodes is checked in each iteration, and if the value is zero, it indicates that routing is completed because there are no congested nodes. However, it is difficult to converge a feasible solution with the limited number of iterations due to the nets exists the dependency and no synchronization performs in asynchronous parallel routing.

Basically, whether the number of congested nodes has stopped decreasing monotonically, which is a sign that the algorithm may not converge a congestion-free state. With this in mind, a promising approach for assuring convergence is to decrease the number of active processes towards the end of routing when the algorithm is not making adequate progress. When the number of congested nodes between nets has stopped monotone decreasing, we reduce the number of active instances to one. It means that we use a single process to route the nets, and this is sequential routing. By transforming the asynchronous parallel routing into the sequential routing, we reduce the possibility of convergence problems.

As described above, while such this approach overcomes the limitation of asynchronous parallel routing, the speedup is inefficient due to the proportion of execution time that the part benefiting from parallelization is very little. This motivates the need to explore the design of synchronous parallel routing for FPGAs.

[1]We set the maximum number of iteration is 50.

B. Synchronous Parallel Routing

Synchronous parallel techniques are commonly used to accelerate the routing for FPGAs. Similar with the existing works [4], [8], [6], we also use the MPI messages to communicate the intermediate results and synchronize the respective states between the processes.

In synchronous parallel routing, each process must know which routing resources are used by other nets and must avoid using such these resources. When a process routed a net, it synchronizes the update with all other processes, and the update message contains the routing results and congestion information of the net. Meanwhile, this process must wait until the update is available from other processes. And once the update is obtained by this process, the next net will be route until it has no more nets to route. Figure 2 provides an example of synchronous parallel routing at every iteration. While it is very costly to stall to synchronize the intermediate results and states due to the dependency of nets, it is easy to find a feasible solution.

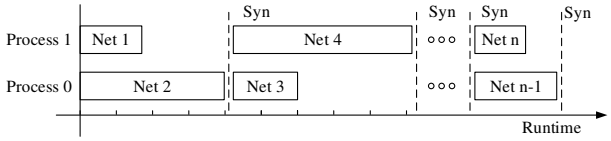


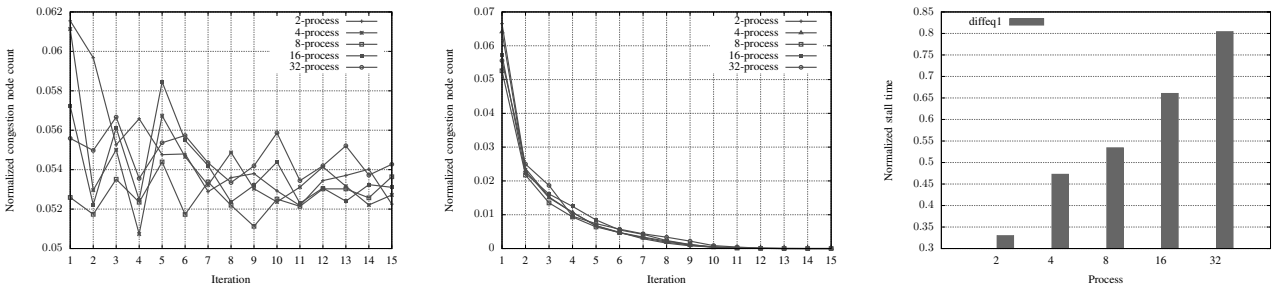
Fig. 2. Synchronous Parallel Routing.

The pseudocode for our synchronous parallel routing is shown in Algorithm 1. Note that it only shows the parallel portion of implementation. The Algorithm 1 is the perspective of one of N processes participating in the routing. The variable i represents the process index, and $local$ is the index of the master process when the algorithm executes. $subset[i]$ is the subset of nets assigned to process i for routing. The array $net[i]$ holds which net is currently being routed by each process. The first_net() function extracts the first net from the subset of process i to route. The next_net() function returns the next net to be routed by process i.

Algorithm 1 Synchronous parallel routing

```
1:  while congestion exists or routing incomplete do
2:    partition the nets into N subsets
3:    for  i such that i ≠ local do
4:      net[i] = first_net(subset[i])
5:    end for
6:    for net_local ∈ subset[local] do
7:      route the net_local
8:      synchronize the update for the net_local
9:      for i such that i ≠ local do
10:       the update is obtained from process i for net[i].
11:       net[i] = next_net(subset[i])
12:     end for
13:   end for
14: end while
```

The while loop is the iteration of routing algorithm, and we partition the nets into N subsets, where N is the number of



(a) (b) (c)

Fig. 3. Analysis of convergence and stall time of asynchronous and synchronous parallel routers.

processes. Each process is aware of the subset of nets being in every other process. For each process i, except the $local$ process, we use the first net of *subset*[i] to initialize the $net[i]$ variable. In the coming outer loop, we route once for each net in the subset of nets of local process. We first route the local net net_local, and then synchronize the update message to all other processes containing the routing result and congestion state of net_local. In the inner loop, we execute once for each process, i, apart from $local$. This loop is responsible for receiving any update messages from other processes that have already been sent. We receive an update message, and then extract the next net from *subset*[i] to update the $net[i]$ that to be routed.

The advantage of such a mechanism synchronous message handling is efficient and convenient, but also overcome the shortcoming of asynchronous parallel routing. With the iteration proceeds, the number of congestion nodes reduces to zero, and that, this parallel router converges a feasible solution. Unfortunately, due to the synchronization is costly, this approach can not provide high speedup.

C. Analysis of Convergence and Stall Time

Here we evaluate one representative design *diffeq1* from the VTR benchmark suite [14], and use both convergence and stall time as two important evaluation metrics of parallel routing approach. We leverage the two aforementioned approaches to parallelize the net routing in the negotiation-based iterative framework [3]. Notably, we use the number of congestion nodes to demonstrate the convergence, and the maximum number of iteration is set to 50. Moreover, stall time is measured using the hardware counters internal to an Intel Xeon processor. Results are given for 2, 4, 8, 16, and 32 processes.

Figure 3 analyzes the normalized congestion node count and stall time of asynchronous and synchronous parallel routers, respectively. Figure 3(a) shows the congestion node count of asynchronous parallel router is oscillatory and can not converge a congestion-free state with the limited number of iterations. Figure 3(b) gives the decrease of congestion node count of synchronous parallel router coincides with the reduction of sequential router during the iteration. While this approach is convergence, the communication overhead is very costly, as shown in Figure 3(c). While we are only presenting one benchmarks here due to space limitation, we observe from our experiments very similar trends to Figure 3 across a broad range of designs, which motivates us to propose SAPRoute that will be detailed in the next section.

IV. SAPRoute Techniques

SAPRoute partitions the nets into processor cores, and applies an iterative fashion to eliminate the congestion in parallel. This fashion integrates commonly used negotiation-based policy to progressively balance the goals of performance and routability. The coordinated synchronous and asynchronous parallelism are exploited to accelerate the routing. In this section we describe the SAPRoute techniques in detail and mainly focus on the speedup optimization. We also show that SAPRoute can easily be extended to handle the timing-driven router.

A. Dependency-Based Coordination

Dependency may occur when several nets are near to each other, when the nets are routed concurrently. To avoid to dependency, processes must recognize when a resource is also used by other routing paths and synchronize the routing states to maintain the convergence in parallel. Two important observation are as follows.

1) Routing resource graph is sparse.
2) Most of the nets are low fan-out[2].

These enlightens that we coordinate the asynchronous and synchronous parallelism to accelerate the potential independent and dependent nets, respectively. Notice that the independent nets are parallelized in asynchronous to strive for more speedup, and the dependent nets are parallelized in synchronous to maintain the convergence.

It is non-trivial to determine the net dependent state due to it is dynamic with the routing iteration proceeds. An promising method is to use the previous dependent state to predict the current net dependency such that we choose the optimal parallel strategy. Notably, the dependent state is involved to the net bounding box before the first iteration, but it is related to the routing resource nodes of net during the iteration. Figure 4 coordinates the synchronous parallelism for the dependent net routing and the asynchronous parallel routing for the independent nets. This approach reduce the communication

[2]We observe that the about 85% nets have only one or two sinks from the VTR benchmarks.

overhead to improve the speedup with the requirement of convergence.

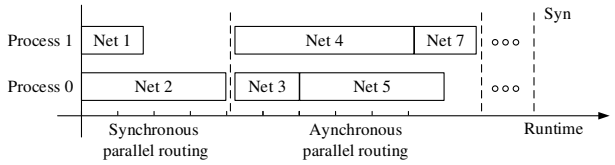


Fig. 4. Coordinated Parallel Routing.

While this approach has the potential to determine the optimal parallel strategy to accelerate the routing, it is challenge to balance the amount of workloads between processes in the multi-core distributed environment. To guarantee the load balance among the processors, we must estimate the runtime used to route a net. The aforementioned method leverages the number of routing resource nodes visited during maze routing in previous iteration to predict the routing time needed for the net in the current iteration. While the exiting work [8] has also demonstrated the effectiveness of this method, they do not provide the full load balance due to there exists difference between the node count and routing time. Moreover, since the net dependency state will change during the routing iterations, the static partitioning is not an optimal method for parallelization. In the next section, we introduce a novel dynamic partitioning approach to explore the maximum parallelism available by routing nets while minimizing the load imbalance.

B. Task-Based Dynamic Partitioning

The task-based dynamic partitioning strategy boosts the parallelism of multi-net routing on multi-core distributed environment. The idea is to maintain all processes working with almost full load during parallelization. Thus, this method can obtain more parallelism than the region-based static partitioning strategy [4], [8], [6]. The algorithmic flow is as follows. A net routing is defined as a task, and a task queue is used to preserve all tasks. The task queue is updated at the beginning of each iteration, and each process repeatedly acquires a task from the task queue when the process finishes the task. Because all tasks are dynamically partitioned into available processes, load variation between processes can be minimized.

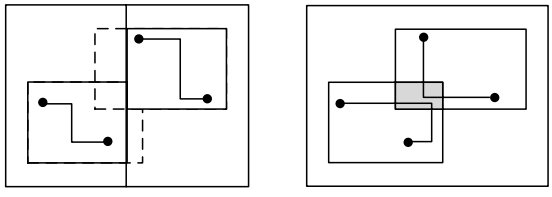


Fig. 5. (a) The region-based partitioning; (b) the task-based partitioning

Although this method overcomes the obstacles of the load imbalance, the competition of routing resources between processes is a critical challenge. Figure 5(a) shows that in the region-based partitioning strategy, the boundary of each subregion constrains the routing search space of each net. The region restriction degrades the routing quality, but it avoids the usage of the same routing resource by more than one process. Figure 5(b) shows the task-based partitioning, one process may compete with other processes for the same routing resource, and this competition probably needs more iterations.

An efficient approach is to roll back an earlier state and start over with software transactional memory when there is a dependency on a shared resource. While the previous work [10] has also demonstrated the effectiveness of this method, they do not produce high parallelism because the rollback process has an expensive overhead. Moreover, this approach do not provide the deterministic results. In the following section, we present a heuristic dependency-aware approach to solve this problem.

C. Dependency-Aware Rip-Up and Re-Route

The original rip-up and re-route strategy takes the most computation time, which the entire routing tree is ripped-up regardless of the congestion state of the nodes. However, we observe two important phenomena by analyzing the rip-up and re-route a net.

1) A low-fanout net often only has a few routing resource nodes.
2) A net bounding box is incrementally relaxed such that the routing search space and the identified routing path of a net change only slightly.

The first tells that the low-fanout net needs only a few detours to avoid the dependent resources. The second implies that the process may reuse most routing resource nodes of the original path in the new routing path[3]. These motivates us to stop new routing path sharing with the resources from the original routing path of other routed nets in the previous iteration.

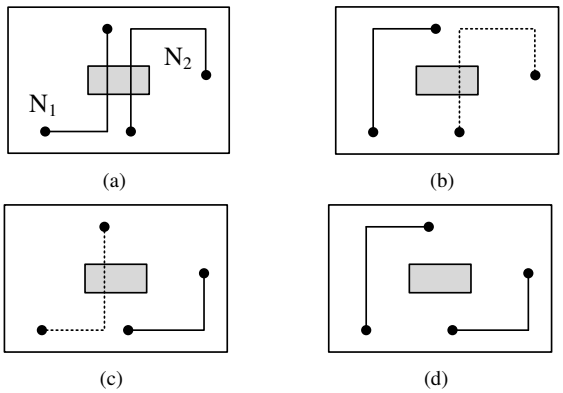


Fig. 6. Dependency-aware rip-up and re-route.

A promising approach is to leverage dependency-aware rip-up and re-route to enable each process to inform other processes, if there exits nodes have been occupied by other nets. Consequently, each process avoids to use these nodes and detour to find other paths. The implementation details are as follows. Prior to the current routing iteration, each process labels these nodes that have been used by other nets in

[3]We observe that the about 20% routing resource nodes are different between the new and original routing path.

the previous iteration, and then cleans these labels until new routing path is constructed. By assigning extra routing cost to theses labelled nodes, this approach encourages alternative routes to be explored to avoid the dependency.

Figure 6 shows that two processes 0 and 1 parallel re-route nets N_1 and N_2, respectively. Initially, two routing paths are given in Figure 6(a). However, two paths are ripped-up due to they are dependent. And at the meanwhile, the original routing path are labelled with dotted lines in Figure 6(b) and Figure 6(c), respectively. Then, the nets N_1 and N_2 attempt to explore the alternative paths because extra routing cost is assigned to the labelled nodes. At last, two legal paths are produced in Figure 6(d). If a routing path uses the labelled nodes, it suggests that the routing quality is still acceptable, although the dependency exists on the path. This is because the capacity is larger than the demand for these nodes.

SAPRoute dramatically reduces the routing time with the dependency-aware rip-up and re-route approach. Moreover, this approach excels at routing the high-stress benchmarks as well.

D. Overall Design Flow

We summarize the overall design flow of SAPRoute using the techniques in Section IV-A, IV-B and IV-C. Figure 7 shows the overall flow of SAPRoute.

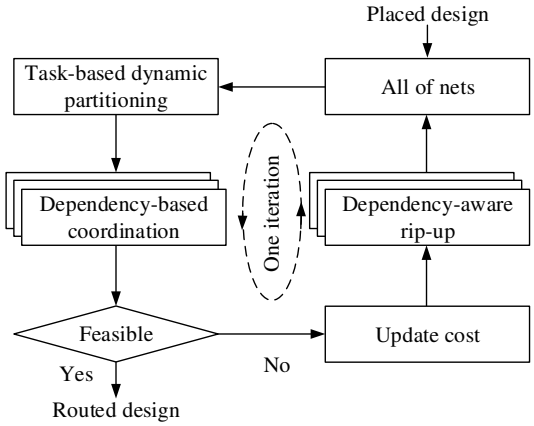


Fig. 7. Overall design flow of SAPRoute.

SAPRoute takes the netlist and routing resource graph as the input, and first leverages a task-oriented scheme to dynamically partition the nets into the processes detailed in Section IV-B. SAPRoute then uses the dependency-oriented fashion detailed in Section IV-A to coordinate the synchronous and asynchronous parallelism to accelerate the routing. Unless converging a congestion-free state, these congestion nets are subsequently ripped-up and parallel re-routed with the dependency-aware technique detailed in Section IV-C, which generates the optimized version of the maze expansion. And then, SAPRoute re-partitions the nets into the processor cores to parallelize the routing.

We define an iteration as the three steps including multi-net partitioning, coordinated synchronous and asynchronous parallel routing, and rip-up and re-route optimization. At each iteration, SAPRoute re-partitions the nets into the different processor cores using the techniques in Section IV-B. In the rare case where the re-partitioned results are identical to the ones from the previous iteration, we discard the current partitioning and re-partition the nets again. This re-partitioning scheme is more effective than a static pre-partitioning, as it allows the coordinated strategy to uncover more parallelism.

In SAPRoute, the overall flow contains a user-specified number of iterations, and the overall flow terminates when it reaches the maximum number of iterations or the routing time limit. A legal routed design is generated as the final result.

E. Extension to Timing-Driven Router

SAPRoute can be extended to support timing-driven delay optimization router with only a few modifications[4] to the parallel design flow. In timing-driven routing algorithm, we assume that the input routing resource graph and netlist are the same as the negotiation-based routing algorithm. And SAPRoute will attempt to reduce the delay of nets that the congestion does not increase. To handle timing-driven routing algorithm, we modify the SAPRoute flow in the following aspects:

1) We replace the negotiation-based router with a timing-driven router so that the delay constraint is likely to be met.
2) During each iteration, after re-partitioning the nets, we add a delay optimization step that re-routes the nets using the delay-oriented routing techniques.
3) After the delay optimization step, we reject the result of the current iteration if the design exceeds the user-defined iterative constraint or the delay of the design increases compared to the previous iteration. If a current iteration is rejected, we reuse the netlist from the previous iteration.

The above extensions ensure that the delay constraint is satisfied throughout the design flow, while the SAPRoute techniques are able to handle the wirelength optimization under the congestion constraint.

V. Experimental Results

In this section, we measure the impact of the novel parallel routing algorithm introduced in the previous section on the quality of results and speedup of SAPRoute. We also compare them with the original VPR 7.0 router and the state-of-the-art parallel FPGA routers.

A. Experimental Setup

We implement SAPRoute techniques in C/C++ and use the VTR 7.0 CAD flow [14] in these experiments. This flow takes as input a benchmark Verilog circuit and an FPGA architecture description file. The flow maps the circuit to the architecture described in that file then outputs statistics about that final mapping. We use Odin II for elaboration, ABC for logic

[4]It is also based on the negotiation-congestion routing flow.

synthesis, AAPack for packing, and VPR 7.0 for placement and routing. VPR is left at default values.

We use the well-known VTR 7.0 benchmarks to evaluate the effectiveness of SAPRoute. The VTR 7.0 benchmarks are a standard set of Verilog circuits that come from a variety of different applications. MCNC benchmarks were not evaluated due to even the largest benchmark spent several minutes to route. Table I shows a summary of the 10 largest benchmarks used for the experiments.

TABLE I
SUMMARY OF RESULTS USING VTR 7.0 ROUTER ACROSS 10 LARGEST BENCHMARKS.

Circuit	Nets	Widt.	Iter.	Wire.	Time(s)
blob.	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
stere.0	9312	96	9	115870	217.62
stere.1	13523	154	9	199814	840.54
LU8PEE.	16278	160	12	426520	2170.24
bgm	27853	116	11	152096	1843.27
stere.2	36479	182	11	702836	14159
mcml	81282	196	11	1542736	29640.2

Experiments were performed on Linux servers, where each node has two 6-core Intel Xeon E5-2430 processors at 2.2GHz and 32GB shared memory. We ran our parallel router using 2, 4, 8, 16 and 32 processes, and use four networked nodes when the number of processes exceeds 8. The baseline for comparison is the serial VPR router, which was implemented in C without any parallelization overhead.

Table I shows the absolute values of running the VTR flow using VPR 7.0 router. The leftmost column lists the circuit used. After that, from left to right, the columns are as follows: 1) The number of nets used in each benchmark; 2) The channel width is the $1.4\times$ minimum channel width [10], [6] needed by VPR router. 3) The number of iterations needed to route the circuit. 4) The routed wirelength when the circuit is routed for the current flow; and 5) The time needed to route the circuit in seconds; This table serves as the baseline values from which the later relative comparisons are made.

B. Speedup and Quality

Figure 8 shows the speedup provided by increasing the number of processes, normalized to single-process VPR 7.0 execution. On average, this synchronous parallel router achieves $1.42\times$, $2.07\times$, $3.85\times$, $5.63\times$, and $6.58\times$ speedup with 2, 4, 8, 16, and 32 processes, respectively. These results indicates that this parallel router scales up to at least 32 processes, although the parallelism is inefficient. The perfectly linear speedup is rarely achieved due to excessive communication overheads and synchronization cost between processes. That being said, these experiments shows that this approach is feasible, and motivates us to explore the asynchronous parallelism for the ideal speedup.

Figure 9 shows the normalized speedup of SAPRoute when we coordinate the synchronous and asynchronous parallelism using dynamic partitioning and dependency-aware rip-up and re-route. On average, speedups of $1.67\times$, $3.35\times$, $6.57\times$, $12.82\times$, and $24.27\times$ are achieved with 2, 4, 8, 16, and 32 processes, respectively. Notably, SAPRoute achieves close to ideal speedup using 2, 4, and 8 processes due to there is very little communication overhead on a single machine. While the perfect speedups are not expected in the multi-server network, SAPRoute still achieves a promising speedup of $24.27\times$ with 32 processes. Compared with the synchronous parallel router, SAPRoute has advantages of high parallelism, and highly scalable. Moreover, it is compatible with the fine-grained parallel router, such as the work of Moctar and Brisk [10], and can obtain much further speedup.

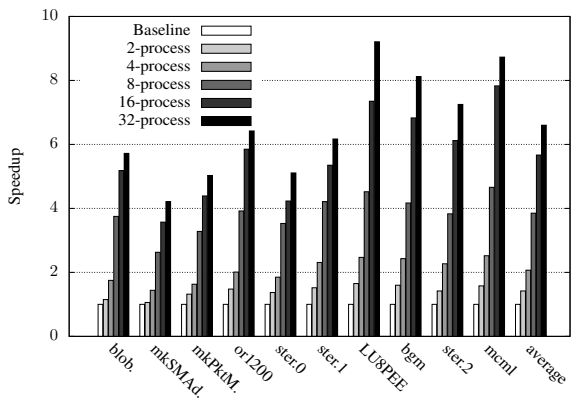


Fig. 8. Speedup of synchronous parallel router using 2, 4, 8, 16 and 32 processes.

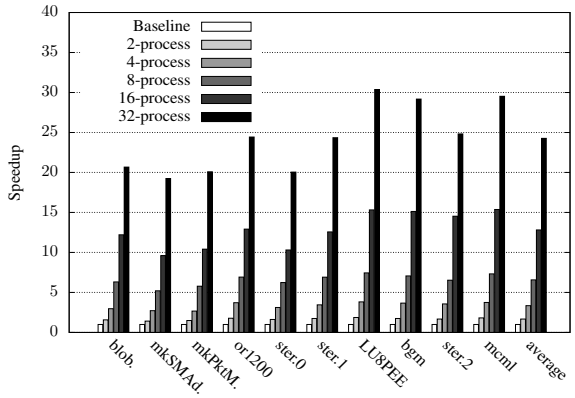


Fig. 9. Speedup of SAPRoute using 2, 4, 8, 16 and 32 processes.

Table II shows the normalized wirelength of synchronous parallel router and SAPRoute using various processes. Observe that in all circuits, the average change to the routed wirelength is less than 0.4% for the synchronous parallel router. This approach do not significantly impact the routed wirelength. Unfortunately, the wirelength is increased by 3.2% using SAPRoute with 32 processes, on average. This is due to the dependency-aware rip-up and re-route fashion requires extra detours to explore the longer routing paths to avoid the congestion. Still it is meaningful to trade 3.2% quality to achieve $24.27\times$ speedup for many FPGA-based applications, especially for the custom computing and logic emulation [13].

As the number of used processes increases, the number of used routing resources increases, and the total size of

TABLE II
IMPACTS ON THE ROUTED WIRELENGTH

Quality	Wirelength									
Name	synchronous parallel router					SAPRoute				
Circuit	2-proc	4-proc	8-proc	16-proc	32-proc	2-proc	4-proc	8-proc	16-proc	32-proc
blob.	119933	119972	120065	120142	120221	120217	120554	121123	122195	123113
mkSMAd.	108546	108572	108637	108783	108962	108843	109128	109635	110573	111727
mkPKtM.	109972	110125	110234	110327	110420	110397	110834	111201	112247	113252
or1200	133867	133892	133956	134321	134434	134395	134672	135313	137205	138334
ster.0	115883	115910	116108	116214	116525	116372	116750	117230	118606	119425
ster.1	199886	199924	200195	200312	200465	200343	200952	201527	203773	205595
LU8PEE	426543	426885	426957	427251	427564	427375	428761	431754	435904	441017
bgm	152125	152157	152371	152540	152665	152603	153245	154136	155312	156568
ster.2	702854	703241	703383	705537	705650	705625	708243	713282	721718	729536
mcml	1542839	1542957	1545243	1547435	1547682	1547564	1553526	1562214	1581572	1596547
Avger.	1.000	1.001	1.001	1.003	1.004	1.003	1.007	1.012	1.023	1.032

used caches by processors increases. It means that SAPRoute needs more cache accesses and less memory accesses. Thus, SAPRoute is highly scalable and can provide further speedup with sufficient resources support.

C. Comparison with existing works

Figure 10 compares the average speedup of SAPRoute to existing parallel routers. The speedup is normalized to VPR router and averaged across the ten benchmarks that we evaluated such that a fair comparison with existing works can be made. Although we use 1.4× the minimum channel width for our experiments, Gort [5], Moctor [10], Shen [6], and Shen [12] use 1.3×, 1.4×, 1.4×, and 1.3×, respectively.

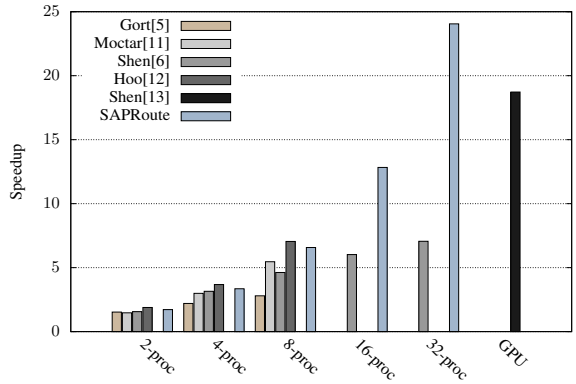


Fig. 10. Speedup of SAPRoute compared to existing work.

TABLE III
COMPARISONS OF ROUTED QUALITY.

Name	Gort	Shen	Hoo	Shen	SAPRoute
Quality	1%	10%	7.5%	2.7%	3.2%

It can be seen that with only 40% higher than minimum channel width, SAPRoute outperforms the state-of-the-art parallel routers. Notably, SAPRoute runs 3.43× and 1.29× faster than the most recent coarse-grained distributed parallel router [6] and fine-grained GPU-accelerated parallel router [12], respectively. While SAPRoute achieves much better performance for maximum parallelism, there still has room for improvement. Moreover, SAPRoute is integrated with fine-grained parallel method [12], making it potential to get much further speedup.

Finally, it can be seen from Table III that our SAPRoute outperforms the parallel router of both Shen [6] and Hoo [11] in terms of quality of results. It is acceptable to have 3.2% degradation on the routing quality for SAPRoute.

VI. CONCLUSION

In this paper we present a coordinated synchronous and asynchronous parallel routing approach based on dynamic partitioning and dependency-aware rip-up and re-route. The advantages of synchronous and asynchronous parallelism are used to parallelize the routing with convergence. Moreover, dynamic partitioning guarantees the full load balance, and dependency-aware rip-up and re-route optimizes the routing time. With these three techniques, this approach provides 24.27× speedup and increases 3.2% wirelength.

VII. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported by NSFC61520106004, NSFC61433019, and 2016YFB1000302.

REFERENCES

[1] K. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz. *"Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap Between Academic and Commercial CAD."* in ACM Transactions on TRETS 2015.
[2] R. Rubin and A. Dehon. *"Timing-driven pathfinder pathology and remediation:quantifying and reducing delays noise in VPR-pathfinder."* in ACM International Symposium on FPGA 2011.
[3] L. McMurchie and C. Ebeling. *"Pathfinder: A negotiation-based performance-driven router for FPGAs."* in ACM International Symposium on FPGA 1995.
[4] P. Chan, M. Schlag, C. Ebeling, and L. McMurchie. *"Distributed-memory parallel routing for field-programmable gate arrays."* in IEEE Transactions on TCAD 2000.
[5] M. Gort and J. Anderson. *"Deterministic multi-core parallel routing for FPGAs."* in IEEE International Conference on FPT 2010.
[6] M. Shen and G. Luo. *"Accelerate FPGA routing with parallel recursive partitioning."* in IEEE/ACM International Conference on ICCAD 2015.
[7] K. Pingali et al. *The tao of parallelism in algorithms.* in ACM Conference on PLDI 2011.
[8] M. Gort and J. Anderson. *"Accelerating FPGA routing through parallelization and engineering enhancements."* in IEEE Transactions on TCAD 2012.
[9] L. Cabral, J. Aude, and N. Maculan. *"TDR: A distributed-memory parallel routing algorithm for FPGAs."* in IEEE International Conference on FPL 2002.
[10] Y. Moctar and P. Brisk. *"Parallel FPGA routing based on the operator formulation."* in ACM Conference on DAC 2014.
[11] C. Hoo, A. Kumar, and Y. Ha. *"ParaLaR: A parallel FPGA router based on lagrangian relaxation."* in IEEE International Conference on FPL 2015.
[12] M. Shen and G. Luo. *"Corolla: GPU-accelerated FPGA routing based on subgraph dynamic expansion,"* in ACM International Symposium on FPGA 2017.
[13] C. Mulpuri and S. Hauck. *"Runtime and Quality Tradeoffs in FPGA Placement and Routing,"* in ACM International Symposium on FPGA 2001.
[14] J. Luu et al. *"VTR 7.0: Next Generation Architecture and CAD System for FPGAs,"* in ACM Transactions on TRETS 2014.