Accelerate FPGA Routing with Parallel Recursive Partitioning

Minghua Shen[†] and Guojie Luo[†‡§]
[†]Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China
[‡]PKU-UCLA Joint Research Institute in Science and Engineering
[§]Collaborative Innovation Center of High Performance Computing, NUDT, China
Email: {msung, gluo}@pku.edu.cn

***Abstract*—FPGA routing is a time-consuming step in the EDA design flow. In this paper we present a coarse-grained recursive partitioning approach to exploit parallelism. The basic idea is to partition the nets into three subsets, where the first subset and the other two subsets consist of potentially conflicting nets and potentially conflicting-free nets, respectively. The two potentially conflicting-free subsets are routed in parallel after the first subset is routed. And all subsets are recursively partitioned in the same way. Furthermore, we point out that the estimated runtime using recursive bisection is close to the optimal estimated runtime using the optimal recursive partitioning, which we can find in polynomial time. The parallel router is implemented using the Message Passing Interface (MPI). Experimental results show that our parallel router ParRoute+ achieves a $7.06\times$ speedup compared to the VPR 7.0 router. This is a $3.36\times$ improvement over a recent coarse-grained parallel router.**

***Keywords*—FPGA routing, recursive partitioning, parallelization, Message Passing Interface (MPI).**

I. Introduction

As the density of FPGA devices keeps increasing, the associated computer-aided design (CAD) tools typically spend many hours synthesizing large designs. There are at least two directions to solve this issue: one attractive direction is hierarchical design reuse [3], and the other promising direction is to design efficient CAD tools by parallelization and hardware acceleration [2]. Though there are existing works on the parallelization of CAD algorithms, the parallelism and the currently available computational power are not fully exploited with a concrete conclusion. In this paper, we focus on the coarse-grained parallelism to reduce the runtime of FPGA routing.

FPGA routing is undoubtedly one of the most time-consuming steps in the CAD flow. The negotiation-based routing algorithm, a.k.a. the *PathFinder* algorithm, has been applied successfully in a variant of commercial FPGA routers [6]. The computational kernel of PathFinder is *maze expansion* - the algorithm used to find a tree to connect the individual pins of a net on the routing resource graph. During maze expansion, the algorithm temporally permits congestion between different nets, such that two different nets may be illegally routed using the same routing resource. To legalize the routing result, congestion penalties are imposed on the conflicting routing resources, and the nets are ripped-up and rerouted according to the increased penalties. The congestion penalties keep adjusted to encourage alternative routes after each iteration, until all the conflicts are resolved and the routing becomes feasible. This PathFinder algorithm enables the nets negotiate with each other to find a feasible routing, but it is naturally sequential and the parallelization is non-trivial.

Recently there have been several efforts in parallelizing the PathFinder algorithm [7], [11], [4]. Though the algorithm is inherently sequential, it is intuitive that a single net do not need to negotiate with a large number of other nets. We use message passing interface (MPI) [14] to exploit such coarse-grained parallelism by recursive partitioning. At each level a cutline partitions the routing resource graph into two subregions, so that the nets are partitioned into three subsets, S_0, S_- and S_+, where S_0 consists of the nets whose pins are across the two subregions, and S_- and S_+ consist of the nets whose pins are inside a single subregion. We first route the nets in S_0, which have higher chance to negotiate with each other, and then route the nets in S_- and S_+ in parallel, which have lower chance to compete for common routing resources. We observe that this dexterous and efficient approach has a positive influence to reduce the number of iterations. Furthermore, this approach is compatible with existing fine-grained parallelization approaches. We strongly believe that the proposed approach with recursive partitioning and routing convergence is a meaningful solution to reduce FPGA routing time.

Our work makes contributions in the following aspects:

- The proposed approach is able to partition the routing tasks into three subsets, S_0, S_- and S_+, where S_0 is a small subset to be routed with little sequential time, and S_- and S_+ are more-or-less balanced subsets to be routed in parallel.
- Empirical results show that there are few conflicts between S_- and S_+.
- We design a polynomial-time algorithm to find the optimal recursive partitioning. Then we point out that the recursive bisection results in a close-to-optimal estimated runtime empirically.
- Experimental results show the efficiency and efficacy of our parallelization approach, which is orthogonal to the existing fine-grained parallelization and can achieve further speedup.

The remainder of this paper is organized as follows. Section II summarizes the background on the PathFinder routing algorithm, as well as the previous efforts on its parallelization. The proposed recursive partitioning approach is described in Section III, and the parallel implementation is presented in Section IV. In Section V the detailed experimental results and comparisons are presented. Finally, conclusions and suggestions for future work are offered in Section VI.

II. Background

The routing resources of an FPGA are represented by a directed graph $G = \langle V, E \rangle$, called the *routing resource graph.* A vertex $n \in V$ corresponds to an electrical pin or a wire segment, and an edge $e \in E$ represents a programmable connection point between an electrical pin and a wire segment, or a programmable routing switch between two wire segments.

The routing of a circuit is to map every net $N_i \in N$ onto the routing resource graph. A net N_i has one source node s_i and a few sinks t_{ij} that are logically connected to the source. Both sources and sinks are vertices in V, and thus the net N_i is a subset of V. The routing of net N_i is to find a subtree in graph G that includes all vertices in N_i, and this subtree is called the routing tree RT_i of net N_i. The source s_i is the root node of RT_i, and the sinks t_{ij} are the terminal nodes. The routing trees for different nets are disjoint in G, in order to prevent short circuits.

A. PathFinder Routing Algorithm

This section will review the congestion-driven PathFinder routing algorithm [10] that we will parallelize. Listing 1 shows a sketch of the PathFinder algorithm to find routes for all the nets. PathFinder routes one net at a time inside each iteration. Congestions are temporally allowed in the intermediate routing results, and the nets must negotiate with each other to determine which one occupies the congested resource in subsequent iterations, until all congestions are resolved and a complete legal routing result is obtained.

Listing 1 The sequential PathFinder algorithm

```
PathFinder(nets $\{N_i\}$, arch $G = \langle V, E \rangle$)
while routing incomplete or congestion exists do
    SeqMaze($\{N_i\}$)
    update history costs of the nodes in $V$
end while
end PathFinder

SeqMaze(nets $\{N_i\}$)
for each unrouted or congested net $N_i$ do
    rip-up $RT_i$ if exists
    $RT_i \leftarrow \{s_i\}$
    for each unconnected sink $t_{ij}$ do
        $P_j \leftarrow$ MazeExpand($RT_i, t_{ij}$)
        $RT_i \leftarrow RT_i \cup \{P_j\}$
    end for
    update present costs of the nodes in $RT_i$
end for
end SeqMaze
```

The PathFinder algorithm is based on the adjustment of congestions costs to resolve conflicts. The cost of a routing resource n is given by

$$c(n) = (d(n) + h(n)) \times p(n)$$

where $d(n)$ is the base cost which reflects the delay or the length of the routing resource, $h(n)$ contains the historical congestion and changes after each iteration, and $p(n)$ reflects the present congestion (the total number of nets that route through) in the current iteration.

B. Previous Parallel FPGA Routing

An early effort on the coarse-grained parallelization by Chan and Schlag [1] achieves a 2.5× speedup using 3 processors, targeting a distributed cluster. The disadvantage of their method is that the results are extremely sensitive to the order of the nets to be routed, due to the unbalanced workload among processors. It is still an open problem to determine the best net ordering [9]. Moreover, their results are not deterministic, and different runs produce different routing results.

Recently, Gort and Anderson [4] proposed a deterministic parallelization approach, which partitions the nets into disjoint subsets, and these subsets are routed in parallel with a proper synchronization scheme to maintain deterministic results while minimizes the overhead of idle time. Their approach achieves the speedups of 1.5×, 1.7×, 2.1× with 2, 3 and 4 cores, respectively.

Another fine-grained approach by Zhu et al. [12] partitioned high fanout nets into several low fanout subnets to be routed individually. Low fanout nets with non-overlapping bounding boxes are routed in parallel since they are unlikely to route in the same routing resource node. They achieved a speedup of 1.9× on a quad-core processor platform with 2.3% degradation in critical path delay.

The most recent work by Moctar and Brisk [7] explores the dynamic parallelism using Galois APIs. Galois is a graph-based parallelization framework to exploit the irregular and dynamic parallelism by speculative execution. For the routing problem, different nets are routed at the same time with speculation and an undo log, so that the misspeculated routing can be undone once conflicts are detected during execution. This dynamic approach exploits more fine-grained parallelism than the static approaches.

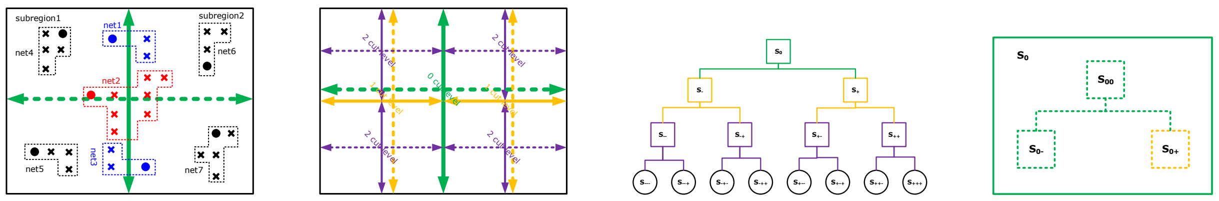
In this paper, we present a novel coarse-grained parallelization method by recursive partitioning. Our approach is compatible with the fine-grained parallelization methods for further acceleration.

III. Routing Task Partitioning

Partitioning plays an important role in attacking the complexity issue of circuit and system design. In this section, we propose a strategy to generate a physical hierarchy for task partitioning.

A. Location-based Partitioning

Instead of directly partition the routing resource graph into subregions, we recursively partition the nets into



(a) Subregions and nets after partitioning (b) Recursive partitioning (c) A tree structure for task assignment (d) Parallelize the nets across subregions

Fig. 1: Parallel recursive partitioning

subsets based on the locations of their pins, while the nets in each subset are still routed in the complete routing resource graph. In this way, all the routing resources are available to every net during routing.

The subsets are defined indirectly by the partitioning of the routing resource graph. We can assign locations to the routing resource nodes, which could be the placement coordinates of the logic blocks or routing tiles that these nodes belong to. Given these locations, we use the median cutline to bi-partition the routing resource region into two subregions, and thus define three subsets of nets: the nets S_0 across two subregions, the nets S_- in the left/lower subregion, and the nets S_+ in the right/upper subregion. An example of the partitioning by the solid green line at level 0 is shown in Figure 1(a), where the black nets represent S_- and S_+ in separate subregions and both the red and blue nets represent S_0 across different subregions.

The nets in subsets S_- and S_+ continue to be bi-partitioned, resulting in a recursive partitioning scheme. An example with three levels of recursive partitioning is shown in Figure 1(b). The nets in S_- (or S_+) are partitioned by the solid yellow line at level 1 into three subsets, S_{-0}, S_{--} and S_{-+} (or S_{+0}, S_{+-} and S_{++}). The nets in S_{-0} and S_{+0} remain at level 1, and the subsets S_{--}, S_{-+}, S_{+-} and S_{++} are further partitioned by the solid purple line at level 2, respectively.

To enable further partitioning, the nets in subset S_0 can be partitioned in the same way. As shown in Figure 1(a), S_0 are partitioned by the dashed green line into three subsets, where the red nets represent S_{00} and the blue nets represent S_{0-} and S_{0+}.

B. Optimal Partitioning

Besides the simple task partitioning by the median cutline introduced above, we formulate a general problem of routing task partitioning, and present a polynomial-time algorithm to find the *optimal* solution of this problem. Though the polynomial is super-quadratic, the size only depends on the number of partitions not the number of nets. This size is user-defined, and the optimization itself can also be parallelized.

The *routing task partitioning problem* is formulated as follows with related notations in Table I. Given

- an FPGA routing architecture and a netlist,
- the coordinate of every pin in every net,
- the minimum unit and the maximum level of partitioning, and
- the runtime estimation to route a net,

find a hierarchical bi-partitioning such that the runtime estimation for the parallelization under this partitioning scheme is minimized.

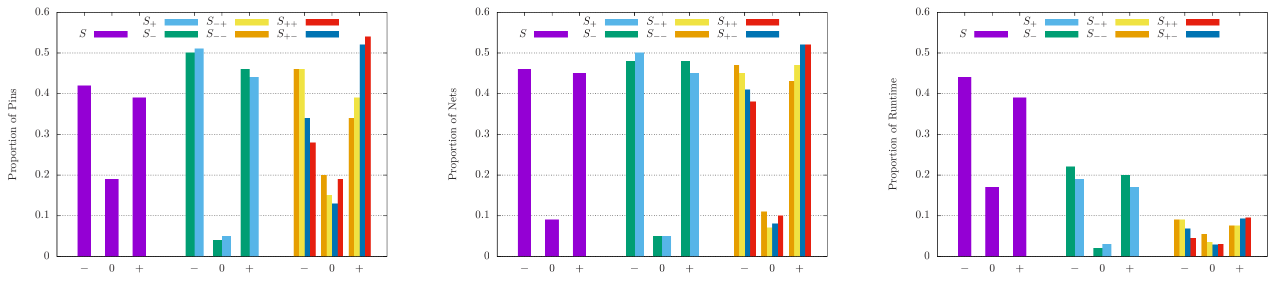
Here we assume the runtime estimation $t(n)$ of a net is a constant during routing. We further assume that if we route the net subset S_1 sequentially, and then route the net subsets S_2 and S_3 in parallel, the runtime estimation is $t(S_1) + max\{t(S_2), t(S_3)\}$. The minimum unit of partitioning is K, such that any cutline will only go along the cell edges but will not go through any cell. The maximum level L limits the depth of partitioning, where the full set of nets is at level 0 and the subsets S_- and S_+ after bi-partitioning are at level 1.

The routing task partitioning can be *optimally* solved by dynamic programming in polynomial time. Due to the page limits, we only outline the idea of this algorithm. Since the number of rectangles $|\{R\}|$ and the number of bi-partitions $|\{\langle R, l\rangle\}|$ are polynomial, we will only show the time complexity related to a single rectangle or a single bi-partition below.

The algorithm consists of two stages: the precomputation stage and the dynamic programming stage. We can precompute the runtime estimation of all the $t(\sigma(R^{1,1}))$ in linear time, and then precompute the runtime estimation of every single $t(S_0\langle R^{w,h}, l\rangle)$ in $O(w^2h^2)$ time, by summing up $t(\sigma(r))$ for every r who is a subregion of R and whose lower-left and upper-right corners are separated by l. After that we can precompute the runtime estimation of every single $t(S(R))$ in $O(1)$ time, by a simple summation $t(S(R)) = t(S_0\langle R, l\rangle) + t(S(R^l_-)) + t(S(R^l_+))$ when the runtime estimations in smaller regions are available.

Based on all the precomputations in polynomial time, we can start the dynamic programming algorithm. The optimal time $T(S(R), 0) = t(S(R))$ since there are no partitions to parallelize. The optimal time

$$T(S(R^{w,h}), L+1) = \\ min \begin{cases} T(S(R^{w,h}), L) \\ t(S_0\langle R, l\rangle) + max\{T(S(R^l_-), L), T(S(R^l_+), L)\} \\ \quad \text{for all cutline } l \end{cases}$$



(a) Number of pins in each subset (b) Number of nets in each subset (c) Runtime of routing for each subset

Fig. 2: Statistics of the task size and the runtime of ParRoute+

TABLE I: Notations for the partitioning problem

Notation	Description
$n \in S$	Net n belong to the net subset S.
K	The minimum unit of partitioning: the routing region is partitioned to $K \times K$ cells, with the grid point $(0,0)$ and $(K+1,K+1)$ at the lower-left and upper-right corners respectively. For simplicity, we assume that a pin does not collide with any edge or any grid point.
L	The maximum level of partitioning.
$R_{x,y}^{w,h}$	The rectangle with its lower-left corner at (x,y), width w and heigh h. When there is no ambiguity, the subscript and the superscript may be omitted.
$(R_-^l, R_+^l) = \langle R, l\rangle$	A bi-partitioning of R by the segment l. l is a horizontal or vertical segment, which consists of the grid points inside R with two endpoints on the edges of R. R is partitioned by l into two subregions R_-^l and R_+^l.
$\sigma(R)$	The nets whose minimum bounding box is R.
$S(R)$	The nets whose minimum bounding box is inside R (inclusive).
$S_0\langle R, l\rangle$	The nets in $S(R)$ across the cutline l.
$t(n)$ or $t(S)$	The sequential runtime *estimation* to route the net(s).
$T(S,L)$	The parallel runtime *estimation* with L as the maximum level of partitioning.

where a single entry can be computed in $O(w+h)$ time and there are $(K+1)^2K^2/4$ entries in total. The solution of the routing task partitioning problem is $T(R_{0,0}^{K,K}, L)$.

It can be proved by induction that the dynamic programming algorithm generates the optimal task partitioning. Detailed analysis can show that the whole algorithms consumes polynomial time and space.

C. Effectiveness of Recursive Bisection

Table II illustrate the effectiveness of the recursive bisection for five representative benchmarks. The columns from left to right are respectively: the circuit name, the number of CLBs, the size of FPGA, the number of nets, the channel width that is 1.4× the minimum channel width [4], [7] and the speedup of optimal recursive partitioning (OPT) and recursive bisection (BIS) using various levels of partitioning.

To obtain the optimal partitioning in feasible time, we set the K in Table I to 20 for all benchmarks. The results are given from 1 to 5 levels of recursive partitioning. Average speedup is also included for comparison.

These data show that the recursive bisection results in a close-to-optimal estimated runtime. Thus, we use the recursive bisection for the parallel routing in the experiments in Section V. Even if the estimated runtime of the recursive bisection is far from optimal for some other circuits, we can include an extra step at the beginning of the routing flow and switch to the optimal recursive partitioning.

D. Partitioning Balance

To balance the task load in the two subsets at the same level, we must find an easy-to-compute metric that is highly correlated with the actual runtime. We considered two different metrics for partitioning:

1) Number of pins, which was shown to be highly correlated with the runtime in [4].
2) Number of nets, which is even easier to compute.

Figure 2(a) and 2(b) shows the statistics data of the pins and nets in each subset for the benchmark *mcml*. The y-axis shows the percentage of these metrics for different subsets at each level, where the first 3 bars represent the first level, the next 6 bars represent the second level, and the last 9 bars represent the third level. Runtime in Figure 2(c) is included for comparison.

IV. Routing Task Parallelization

We present our coarse-grained parallel PathFinder routing algorithm: ParRoute and ParRoute+. ParRoute partitions the routing tasks into three subsets, and two of these subsets are recursively partitioned and routed in parallel. ParRoute+ recursively partitions and routes all three subsets to get further speedup. The effects on the number of iterations and the convergence are also analyzed.

A. Recursively Parallel Routing

Our parallelization approach is to partition the nets into three subsets recursively, which are formulated as tasks and then map to separate processes. Each task routes its own set of nets and maintains its own data structures, including a routing resource graph and the associated congestion information. The owner of a tasks uses MPI

TABLE II: Comparison of estimated speedup between optimal and bisectional partitioning

Benchmark	CLBs	Array Size	Nets	Channel Width	1 level		2 level		3 level		4 level		5 level	
					OPT	BIS	OPT	BIS	OPT	BIS	OPT	BIS	OPT	BIS
diffeq1	1460	35x35	3953	48	1.36	1.36	2.43	2.23	3.29	3.08	4.29	4.01	5.25	4.93
blob_merge	2702	51x51	6606	68	1.38	1.34	2.47	2.27	3.36	3.12	4.19	3.87	5.48	4.84
mkPktMerge	3767	58x58	7474	52	1.47	1.47	3.12	2.97	4.13	3.62	5.21	4.97	6.03	5.48
bgm	4225	73x73	27853	116	1.42	1.42	2.54	2.28	3.48	3.17	4.50	4.41	5.12	4.93
mcml	7934	101x101	81282	196	1.47	1.47	3.16	3.03	4.23	3.99	5.26	5.04	6.08	5.57
Avg.					1.42	1.41	2.74	2.56	3.70	3.40	4.69	4.46	5.59	5.15

messages to assign sub-tasks to other processes and collect partial routing results from the owner of the sub-tasks.

In Listing 2 we implement the parallel PathFinder algorithm using MPI in a distributed-memory system, where the send and receive operations are the primitives for communications and synchronizations among processes. The master process in line 4-8 are responsible for the whole routing flow, which is similar to Listing 1 except for the parallel kernel in line 5. The other processes in line 10-14 are responsible for performing the task assignments. The tasks form a binary tree as illustrated in Figure 1(c), where the dashed lines in red represent the task assignments, and the dashed lines in blue represent the recursive function calls.

We statically assign tasks to processes, so that each process knows whom it should ask for tasks at line 12, as well as whom it should assign tasks to at line 25. The task labeled t in a perfect binary tree generated by the recursive task partitioning has the properties that parent(t) = $\lfloor (t-1)/2 \rfloor$, lchild(t) = $2\times t+1$ and rchild(t) = $2\times t+2$. The top-level task assignment assigns the root node with task id $t = 0$ in an L-level perfect binary tree to process with id $p = 0$. Process p first completes the routing task t; then it assigns the $(L-1)$-level subtree rooted at task rchild(t) to process $p + 2^{L-1}$ using MPI message, and assigns another $(L-1)$-level subtree rooted at task lchild(t) to itself by recursive function call (or replace this tail recursion by iteration). According to such static assignment, we can compute the task_id(p) at line 10, which is the first task assigned to process p by MPI message, and compute the owner_pid(t) at line 11 and 25, which is the id of the process responsible for performing the routing task t.

Listing 2 Parallel PathFinder

```
 1: **ParRoute**(nets $\{N_i\}$, arch $G = \langle V, E \rangle$)
 2: $pid \leftarrow$ process_id()
 3: **if** $pid == 0$ **then**
 4:     **while** routing incomplete or congestion exists **do**
 5:         $\{RT_i\} \leftarrow$ **ParMaze**(0, $\{N_i\}$)
 6:         update history costs of the nodes in $V$
 7:     **end while**
 8:     **return** $\{RT_i\}$
 9: **else**
10:     $tid \leftarrow$ task_id($pid$)
11:     $manger \leftarrow$ owner_pid(parent($tid$))
12:     **receive** nets $\{N_j\}$ from $manager$
13:     $\{RT_j\} \leftarrow$ **ParMaze**($tid$, $\{N_j\}$)
14:     **send** $\{RT_j\}$ to $manager$
15: **end if**
16: **end ParRoute**
17:
18: **ParMaze**(task id $tid$, nets $\{N_j\}$)
19: **if** $tid$ is at the last level **then**
20:     $\{RT_j\} \leftarrow$ **SeqMaze**($tid$, $\{N_j\}$)
21:     **return** $\{RT_j\}$
22: **else**
23:     $(S_0, S_-, S_+) \leftarrow$ partition($\{N_j\}$)
24:     $\boxed{\{RT_j : N_j \in S_0\} \leftarrow \textbf{SeqMaze}(S_0)}$
25:     $worker$ = owner_pid(rchild($tid$))
26:     **send** $S_+$ to $worker$
27:     $\{RT_j : N_j \in S_-\} \leftarrow$ **ParMaze**(lchild($tid$), $S_-$)
28:     **receive** $\{RT_j : N_j \in S_+\}$ from $worker$
29:     **return** $\{RT_j\}$
30: **end if**
31: **end ParMaze**
```

B. Enhanced Recursive Parallel Routing

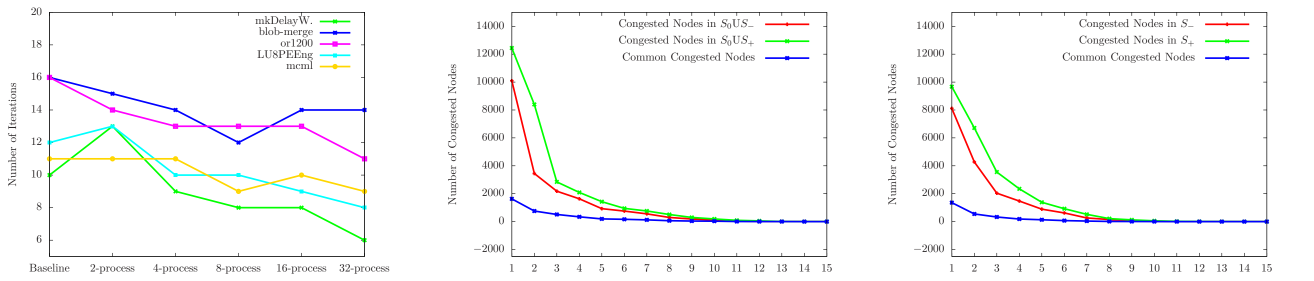
The two processes on the left in Figure 3 show the synchronization scheme in ParRoute. Using synchronization we impose artificial orders that the routing of S_- and S_+ do not start until S_0 finishes, and the routing of S_0 in the next iteration does not start until S_- and S_+ finish. The nets in each subset are either routed sequentially or partitioned in the same way and routed in parallel. It is clear that our parallel routing algorithm is deterministic.

The idle time affects the efficiency of the parallelization. In order to improve efficiency and reduce runtime, the nets in S_0 is also parallelized in the same way. An example is shown on the right in Figure 3, where it leads to an effective runtime reduction.

Such runtime reduction can be implemented by modifying line 24 in Listing 2, in the same way as the procedure of ParMaze to explore further parallelism without adversely affecting the routing quality. The enhanced parallel implementation is call ParRoute+.

C. Empirical Results on Convergence

We observe an interesting fact in Figure 4(a) that the number of iterations decreases as the number of processes increases. This is because that our ParRoute/ParRoute+ imposes different net ordering. We performed experiments to change the net orders in the sequential PathFinder as in ParRoute/ParRoute+, and also observed the reduction in the number of iterations. Unfortunately such reduction



(a) Number of iterations (b) Congested nodes in S_0 (c) Common congested nodes in S_- and S_+

Fig. 4: Convergence Data from ParRoute+.

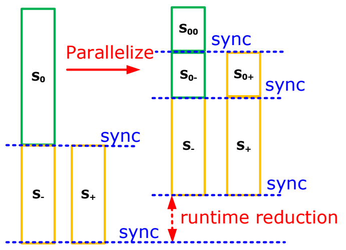


Fig. 3: Runtime reduction in ParRoute+

TABLE III: Benchmark summary

Benchmark	Architecture File	Nets
mkDelayW.	k6_frac_N10_mem32K_40nm	5224
blob_merge	k4_N4_90nm	6606
mkSMAdap.	k4_N4_90nm	7154
mkPKtMerge	k4_N4_90nm	7474
or1200	k4_N4_90nm	8078
stereovision0	k6_frac_N10_mem32K_40nm	9312
stereovision1	k6_frac_N10_mem32K_40nm	13523
LU8PEEng	k6_frac_N10_mem32K_40nm	16278
bgm	k6_frac_N10_mem32K_40nm	27853
stereovision2	k6_frac_N10_mem32K_40nm	36479
mcml	k6_frac_N10_mem32K_40nm	81282

is not free, in Section V we will show the impact on routed wirelength. Regardless, net ordering enables the trade-off between quality and runtime.

The PathFinder algorithm converges when the usage of every routing resource node is below its capacity. In order to empirically investigate in the convergence property, we count the number of congested nodes in Figure 4(b) and 4(c), where the x-axis is the iteration number, and the y-axis is the number of congested nodes. Figure 4(b) shows the congested nodes in $S_0 \cup S_-$, $S_0 \cup S_+$ and their intersection, and Figure 4(c) shows the congested nodes in S_-, S_+ and their intersection. The data tells us that there is only a small percentage of common congested nodes, and the number of common congested nodes reduces to zero much more quickly than the other congested nodes. Therefore, the separated routing of nets in S_- and S_+ has small impact on the convergence.

V. Experimental Study

In this section, we present and analyze the results obtained using the parallelized routing algorithm ParRoute and ParRoute+ described in the previous section. We evaluate the speedup and quality of both methods, and also compare them with the original VPR 7.0 router and the state-of-the-art parallel FPGA routers.

A. Experimental Setup

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 original single-process single-thread VPR, which was implemented in C without any parallelization overhead.

As it is more important to reduce runtime of large circuits, only those circuits with more than 5000 nets are included in our experiments; smaller circuits were excluded. All the experiments were run with the 11 largest circuits from VTR benchmarks commonly used in FPGA CAD research [8]. Table III summarizes the benchmark circuits, the architecture file, and the number of nets. The larger circuits are synthesized and placed on the architecture *k6_frac_N10_mem32K_40nm* that is similar to modern FPGAs, and the smaller circuits are synthesized and placed on a simple architecture *k4_N4_90nm* to increase the problem size. The FPGA routing architecture contains unidirectional wire segments that span two logic block tiles.

We used ABC [13] for logic synthesis and technology mapping, and use T-VPack [5] and VPR placer [8] for packing and placement respectively. Across all runs, every benchmark was routed using a channel width of 1.4× the minimum channel width [4], [7] needed by VPR.

B. Experimental Results

Figure 5 shows the speedups of ParRoute versus the number of processes. The speedups of each circuit are shown in a cluster of bars, and the average speedups are shown in the last cluster. On average, speedups of 1.49×, 2.94×, 3.95×, 4.43× and 5.88× are achieved with 2, 4, 8, 16 and 32 processes, respectively. These results

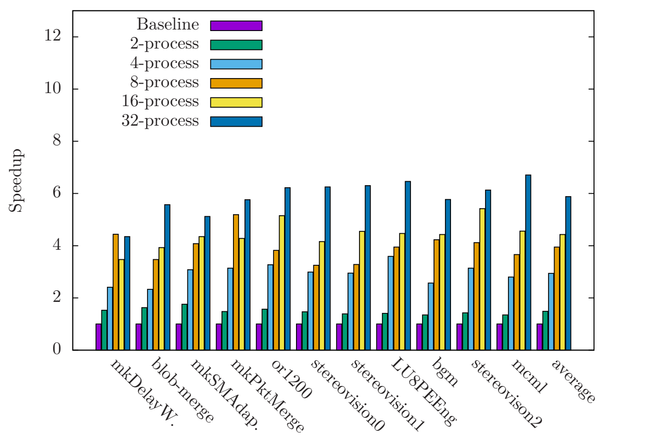


Fig. 5: Speedup of ParRoute with 2, 4, 8, 16 and 32 processes

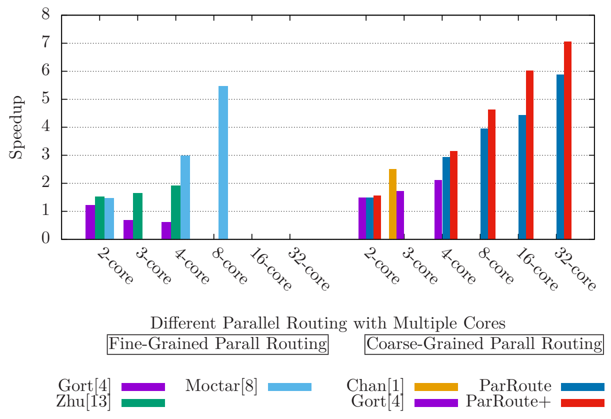


Fig. 7: Comparisons between ParRoute/ParRoute+ and state of the arts

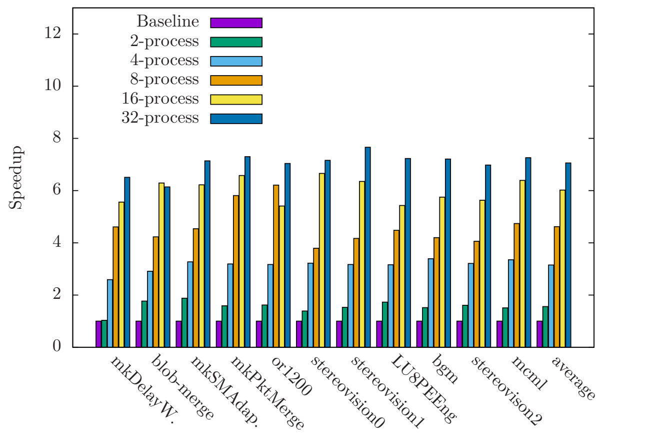


Fig. 6: Speedup of ParRoute+ with 2, 4, 8, 16 and 32 processes

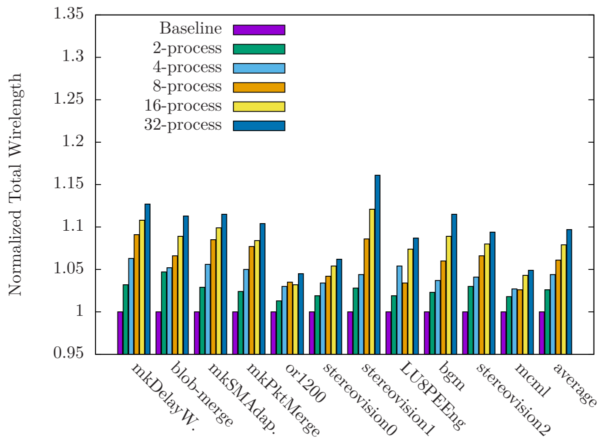


Fig. 8: Impacts on the routed wirelength

demonstrate adequate accelerations of ParRoute with up to 32 processes, and it is expected that we can use more processes to accelerate the routing of even larger circuits.

Moreover, ParRoute can be enhanced by parallelizing the routing tasks of the nets across subregions, the overall runtime can be further reduced. Figure 6 reports the speedups of ParRoute+, which achieves average speedups of 1.56×, 3.15×, 4.62×, 6.02× and 7.06× with 2, 4, 8, 16 and 32 processes, respectively. Please note that the computing resources that required by ParRoute+ are not greater than ParRoute.

For comparison, we list the results of previous coarse-grained and fine-grained parallelization methods in Figure 7. An early coarse-grained parallel router by Chan and Schlag[1] reported a speedup of 2.5× using three processors, and Gort and Anderson [4] reported a speedup of 2.1× using four cores with deterministic parallelization. The latest work by Moctar and Brisk [7] based on fine-grained speculative parallelism shows a speedup of 5.46× using 8 threads. Given adequate amount of computing resources, our ParRoute+ achieves the maximum speedup of 7.06× among existing results. Please note that ParRoute+ is compatible with the fine-grained parallelization, and can get much further speedup. The expected speedup mixing ParRoute+ and fine-grained speculative parallelism can be as great as $7.06 \times 5.46 \approx 38$ using $32 \times 8 = 256$ processing elements.

Figure 8 reports the normalized routed wirelength of ParRoute+ using various processes. Though on average the routed wirelength is increased by 10% using ParRoute+ using 32 processes, it is still meaningful for non-timing-critical applications and fast design iterations.

VI. Conclusions and Future Work

Parallelization is a promising direction to reduce the runtime of FPGA CAD tools. In this paper, we present a recursive partitioning approach for deterministic parallel FPGA routing. The effectiveness of the recursive bisection approach is demonstrated by comparing with the optimal recursive partitioning. In the recursive partitioning strategy, we partition the nets into three subsets by imposing a pseudo-cutline in the routing resource graph, where the subset of nets across the cutline will be routed first, and the two remaining subsets of nets separated by the cutline will be recursively partitioned. We invoke multiple processes to complete the partitioned routing tasks, and use message passing interface (MPI) for synchronization and data exchange. We further show that the nets across the cutline can be further partitioned for more parallelism. Results show that our parallel approach provides 6.02× speedup using 16 processes and 7.06× speedup using 32 processes. Although the proposed approach increases the routed wirelength by 10% on average using 32 processes, it is still meaningful for non-timing-critical applications and fast design iterations.

The future work include the following aspects. 1) The quality degradation of routed wirelength could be reduced by more data synchronizations between the tasks in S_- and S_+, while currently there are no data synchronizations within one routing iterations. 2) The current median-cutline partitioning approach can be significantly improved by solving the routing task partitioning problem. 3) The impact on the critical path delay in the timing-driven mode will be studied. 4) More in-depth analysis about the net ratio in S_0, and the chances of net conflicts among S_- and S_+.

Acknowledgment

This work is partly supported by National Natural Science Foundation of China (NSFC) Grant 61202073, Research Fund for the Doctoral Program of Higher Education of China (MoE/RFDP) Grant 20120001120124, and Beijing Natural Science Foundation (BJNSF) Grant 4142022.

References

[1] P. Chan and M. Schlag, *"Acceleration of an FPGA router,"* IEEE Symp. Field Programmable Custom Computing Machines,

[2] A. DeHon, R. Huang, and J. Wawrzynek, *"Hardware-assisted fast routing,"* Proc. 10th Annu. IEEE Symp. Field-Programmable Cust. Comput. Mach., pp. 205-215, 2002.

[3] M. Gort and J. Anderson, *"Design Re-Use for Compile Time Reduction in FPGA High-Level Synthesis Flows,"* In ACM Int'l Symp. on FPT, pages 4-11, 2014.

[4] M. Gort and J. Anderson, *"Deterministic multi-core parallel routing for FPGAs,"* Int. Conf. Field Programmable Technology, pp. 61-69, Dec. 8-10, 2010.

[5] A. Marquardt, V. Betz, and J. Rose. *"Using cluster based logic blocks and timing-driven packing to improve FPGA speed and density,"* In Int'l Sym. on FPGAs, pages 37-46, Monterey, CA, 1999.

[6] L. McMurchie and C. Ebeling, *"Pathfinder: A negotiation-based performance-driven router for FPGAs,"* in Proc, 3rd Int, ACM/SIGDA Symp. Field-Programmable Gate Arrays, Monterey, CA, Feb. 1995, pp.111-117.

[7] Y. Moctar and P. Brisk, *"Parallel FPGA Routing based on the Operator Formulation,"* ACM/IEEE Design Automation Conference, San Francisco, CA, USA, June 01-15, 2014.

[8] J. Rose, J. Luu, C. Yu, O. Densmore, J. Goeders, A. Somerville, K. Kent, P. Jamieson and J. Anderson, *"The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing,"* ACM/SIGDA Int. Symp. FPGAs, pp. 77-86, 2012.

[9] R. Rubin and A. Dehon, *"Timing-driven pathfinder pathology and remediation: quantifying and reducing delays noise in VPR-pathfinder,"* ACM/SIGDA Int. Symp. FPGAs, pp. 173-176, Feb. 27- Mar. 1st, 2011.

[10] J. Swartz, V. Betz and J. Rose, *"A fast routability-driven router for FPGAs,"* ACM/SIGDA Int. Symp. FPGAs, pp. 140-149, Feb. 22-24, 1998.

[11] I. Watson, C. Kirkham and M. Lujan, *"A Study of a Transactional Parallel Routing Algorithm,"* in International Conference on Parallel Architecture and Compilation Techniques, 2007.

[12] C. Zhu, J. Wang, and J. Lai. *"A novel net-partition-based multithreaded FPGA routing method,"* In Int'l Sym. on FPL, Sept. 2-4, 2013.

[13] Berkeley Logic Synthesis and Verification Group. *"ABC: A system for sequential synthesis and verification, Release 70930."* http://www.eecs.berkeley.edu/ alanmi/abc/.

[14] *"Open MPI: Open source high performance computing,"* http://www.open-mpi.org/, 2010.