SPRoute: A Scalable Parallel Negotiation-based Global Router

Jiayuan He[1], Martin Burtscher[2], Rajit Manohar[3], Keshav Pingali[4]

[1,4]*Department of Computer Science, University of Texas at Austin, Austin, TX, USA*
[2]*Department of Computer Science, Texas State University, San Marcos, TX, USA*
[3]*Computer Systems Lab, Yale University, New Haven, CT, USA*
[1]hejy@cs.utexas.edu, [2]burtscher@txstate.edu, [3]rajit.manohar@yale.edu, [4]pingali@cs.utexas.edu

***Abstract*—The complexity of global routing increases rapidly as chip designs grow larger. In many global routers, maze routing is the most time-consuming stage. One way to reduce its runtime is parallelization. Existing parallel maze routers work either by identifying and routing independent nets or by partitioning the chip area into non-overlapping regions. In this paper, we describe a scalable parallel global router called SPRoute that initially exploits net-level parallelism, automatically lowers the parallelism when livelock is identified, and finally switches to fine-grain parallelism to guarantee convergence. We evaluate SPRoute on a 28-core machine on the ISPD 2008 global routing contest benchmark suite. It achieves an average speedup of 11.5 with a wirelength penalty of 0.6% on overflow-free benchmarks, and an average speedup of 4.5 with a total overflow penalty of 7% on hard-to-route benchmarks over sequential SPRoute. Compared to FastRoute 4.1, SPRoute achieves an average speedup of 11.0 and 3.1 on overflow-free benchmarks and hard-to-route benchmarks, respectively.**

***Index Terms*—Global Routing, Parallel Computing**

I. Introduction

Global routing is a crucial stage in the VLSI physical design flow. It can be used either to quickly estimate routability and congestion in the early stages before routing, or to provide guidance for detailed routing to generate the wire layout. In the past few years, global routing has attracted considerable research effort and many robust global routers have been developed, including FGR [16], MaizeRouter [11], BoxRouter [3], NTHU-Route [2], NCTU-gr [9] and FastRoute [13]. Most of these global routers are based on the negotiation-based rip-up and reroute maze routing technique introduced in PathFinder [10]. Techniques such as monotonic and 3-bend routing, introduced in FastRoute 4.1 [13], can improve the routing quality and reduce the runtime. However, the focus of most global routers is on optimizing the routing quality, *i.e.*, minimizing overflow, wirelength and number of vias; reducing runtime is a secondary concern.

As chip designs grow larger, the complexity and runtime of global routing is increasing rapidly. Figure 1 shows the general design flow of many modern global routers. In this flow, maze routing is the most important step. It is not only crucial to routing quality but it is also the most time-consuming step. Therefore, we believe it is imperative to develop a fast and high-quality maze routing technique for global routing.

One way to reduce the high runtime is parallelization, but existing parallel global routers do not provide much speedup over sequential execution even when they exhibit good scalability. In this paper, we first study net-level parallelization. In net-level parallelization, each thread acquires a net and

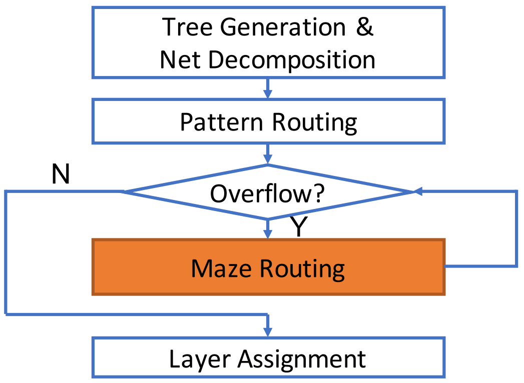


Fig. 1: Design flow of many modern global routers. First, each net is decomposed into 2-pin edges to generate a tree. Second, pattern routing is used to generate a routing solution for the decomposed 2-pin nets. If there is overflow on the grid after pattern routing, the global router enters an iterative rip-up and reroute maze routing stage until the total overflow decreases to zero or the maximum number of iterations is reached. Finally, a layer assignment stage is applied to generate a multi-layer solution.

applies negotiation-based rip-up and reroute maze routing on the thread's local grid. After local maze routing is finished, the thread updates the routing resource usage in the global grid. Since other threads may have routed nets through the same regions concurrently, resource usage may have exceeded resource availability. One way to avoid this is to enforce disjointness of the routing regions of different nets but this limits parallelism and results in longer running times. Another way to address this problem is to rollback some threads when resource conflicts are detected but this can result in livelock.

To address these problems, we propose a *two-phase* maze routing approach that initially exploits net-level parallelism, automatically lowers the parallelism when livelock is identified, and finally switches to fine-grain parallel processing of individual nets. This solution permits nets to be routed in parallel through the same region as long as enough routing resources are available, so performance will be good for "easy-to-route" designs. For more difficult designs in which there is congestion in some routing regions, this solution still permits some parallelism to be exploited.

This paper describes an implementation of this approach called SPRoute, a Scalable Parallel global Router. SPRoute

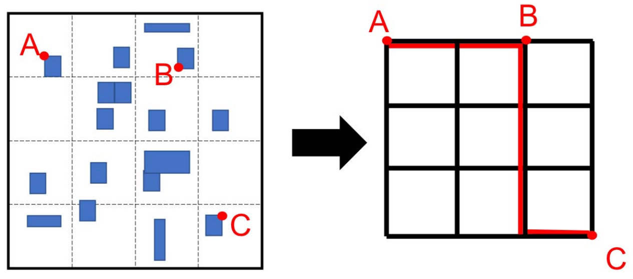


Fig. 2: Grid graph of global routing. The chip area is partitioned into 4×4 bins and forms a 4×4 grid graph. The red lines on the grid graph show a solution to route a net that connects pins A, B and C.

uses ideas from the sequential FastRoute 4.1 [13]. To make it easy to use, SPRoute has a single parameter set for all inputs. We evaluate SPRoute on the ISPD 2008 global routing contest benchmark suite [1]. It achieves an average speedup of 11.5 with a wirelength penalty of 0.6% on a 28-core machine on overflow-free benchmarks, and an average speedup of 4.5 with an overflow penalty of 7% on hard-to-route benchmarks over sequential SPRoute. Compared to FastRoute 4.1, SPRoute achieves an average speedup of 11.0 and 3.1 on overflow-free benchmarks and hard-to-route benchmarks, respectively.

In summary, the contributions of this work are as follows.

1) We propose a novel two-phase maze routing algorithm that combines a net-level parallel phase and a fine-grain parallel phase. The proposed approach resolves the livelock issue in net-level parallel routing and achieves good speedup.
2) We describe an implementation of this two-phase maze routing algorithm called SPRoute. We show that SPRoute achieves good speedup on multi-core machines with an acceptable wirelength and overflow penalty compared to state-of-the-art global routers.
3) We study the limited parallelism in the fine-grain stage and investigate factors that limit speedup.

SPRoute will be released as open-source code.

The rest of the paper is organized as follows. Section II provides background on global routing and discusses the livelock issue of net-level parallelization. Section III summarizes related work. Section IV describes the proposed two-phase maze routing approach in detail. Section V provides experimental results. Section VI concludes the paper.

II. Background

A. Global Routing

In global routing, a set of nets and a routing region are given as input. Each net consists of a set of pins. Each pin represents a pin of a cell in the circuit and has a fixed coordinate generated by the placement stage. The routing region is represented as a grid graph. The nodes of the grid graph represent the routing grid bins and the edges represent the connections between adjacent grid bins. Figure 2 shows an example of a 4×4 grid. The chip area is partitioned into 4×4 bins and forms a 4×4 grid graph. The blue rectangles represent the cells of the circuit. Vertices A, B and C are three pins on the same net and need to be connected. The red lines on the grid graph show a solution to route this 3-pin net.

The capacity of an edge is the maximum number of wires that are allowed to route through the edge and is given as an input. The usage of an edge is the number of wires going through the edge. The overflow of an edge is the number of wires that exceeds the capacity.

The global routing problem is the following: for every input net, find a route that connects the pins of the net on the grid graph. The first objective is to minimize the total overflow of all edges. The second objective is to minimize the wire length. The cost of vias is also considered in multi-layer designs.

B. Modern Global Routers

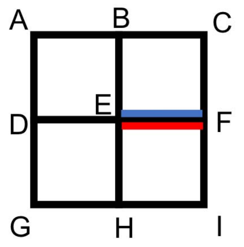
Most modern global routers are based on negotiation-based rip-up and reroute maze routing, which was introduced by PathFinder [10]. The general design flow is shown in Figure 1. First, each multi-pin net is decomposed into a set of 2-pin nets in the net decomposition stage. A widely used technique for decomposition is to generate a rectilinear Steiner minimum tree (RSMT) using FLUTE [4]. In the second stage, pattern routing is used to generate the routing solution for the decomposed 2-pin nets. Each global router applies its own pattern routing techniques, such as probabilistic L-shaped pattern routing in NTHU-Route [2] or L & Z and 3-bend routing in FastRoute [13]. If there is overflow on the grid after pattern routing, the global router enters an iterative rip-up and reroute maze routing stage until the total overflow decreases to zero or the maximum number of iterations is reached. Finally, a layer assignment stage is applied to generate a multi-layer solution.

Maze routing is the most time-consuming stage in the design flow of many modern global routers. Thus, it is necessary to develop fast and high-quality maze routing techniques for global routers.

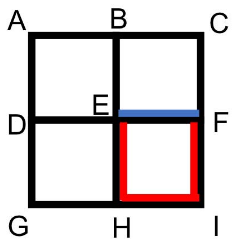
C. Livelock in Net-level Parallelism

One way to reduce the runtime of maze routing is parallelization. In net-level parallelism, each thread first acquires a net and rips up the net if there is overflow on the edges it uses. The net is ripped up by subtracting the usage from the global grid. Then the thread applies maze routing on the thread's local grid and finally adds the usage to the global grid.

Unless care is taken, exploiting net-level parallelism can result in race conditions or even livelock when different nets are routed on the same region. The probability of livelock depends on the number of concurrent nets, *i.e.*, the number of threads in net-level parallelism. Figure 3 shows a simple example of livelock on a 3×3 grid in which the capacity of each edge is 1. Assume nodes E and F are connected by two nets and that both initial routing solutions use edge (E, F). Edge (E, F) is an overflow edge with a capacity of 1 and a usage of 2. Suppose two threads acquire one net each and start rip-up and reroute at the same time. Both threads detect the

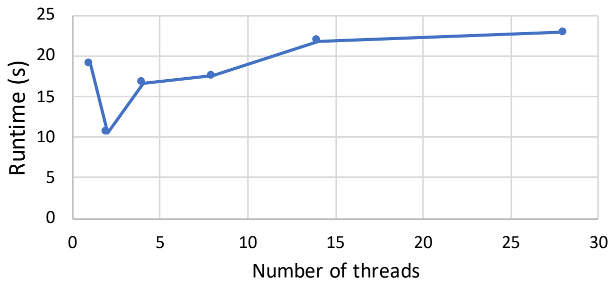


(a) Parallel

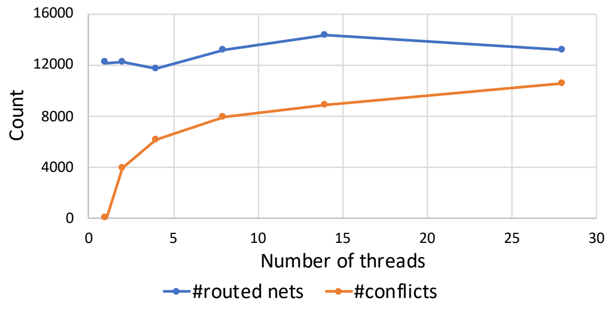


(b) Sequential

Fig. 3: An example of livelock. (a) In parallel execution, two threads identify two nets have overflow on edge (E, F), rip them up, and then reroute on edge (E, F) again. (b) In sequential execution, one net routes after the other, so the later net is not ripped up.



(a) Runtime Scalability



(b) Number of Routed Nets vs. Conflicts

Fig. 4: Scalability and conflicts when locks are applied to rip up edges in the first iteration of the $newblue3$ input.

overflow edge (E, F), and after rip-up the usage of edge (E, F) becomes 0. During maze routing, both threads will read the usage of edge (E, F) as 0 and route on it again. This results in livelock.

One way to resolve the livelock is to lock the routing region when one net is working on it, thus sequentializing the conflicting nets. However, this reduces the amount of parallelism. Figure 4 shows the runtime and conflicts when locks are applied to rip up edges. The program only scales to 2 threads because more threads result in high conflict rates and thus sequentialize conflicted nets. The runtime of 14 and 28 threads is higher than the single-thread runtime. Figure 4 shows the performance of locking only the rip-up edges rather than the entire routing region, which leads to more contention.

The livelock issue in net-level parallelism has been studied in NCTU-gr 2.0 [9] and Han *et al.* [6]. NCTU-gr 2.0 uses a collision-aware technique that adjusts the cost of the grid when identifying collisions. The cost of edges in the conflicting region is increased to reduce the probability of using edges in the region. Han *et al.* implemented a scheduler to sequentialize the conflicting nets to guarantee mutual exclusion. However, the resulting amount of parallelism is limited and depends highly on the input circuit.

To address this problem, we propose a net-level-sequential fine-grain-parallel technique to resolve the livelock issue and achieve competitive speedups. This technique is implemented in the Galois system, described in Section II-D.

D. *Galois System*

Parallelism in graph algorithms such as maze routing can be described abstractly using a data-centric algorithm abstraction called the *operator formulation* [14].

The operator formulation has a local view and a global view of algorithms. The local view is called the *operator* and it specifies a state update rule that is applied to an *active node* in the graph. In maze routing, the operator updates the four neighboring nodes of the active node and expands the boundary. The global view of an algorithm is the *schedule*, which specifies the order in which active nodes must be processed. In the case of maze routing, grid nodes that are closer to the source nodes have higher priority and should be expanded first.

The Galois system is an open-source C++ library designed to ease the implementation of parallel graph algorithms described using the operator formulation. The programmer replaces serial loop constructs (e.g., *for* and *while* loops) and serial data structures with parallel loop constructs and concurrent data structures provided by the Galois system. Galois is designed so that the programmer does not have to deal with low-level parallel programming constructs such as threads, locks, barriers, condition variables, etc.

In this work, the Galois *do_all* parallel loop and $LargeArray$ are used to exploit net-level parallelism. The for_each parallel loop and $OrderedByIntegerMetric$ ($OBIM$) scheduler are used to exploit fine-grain parallelism.

III. Related Work

Several parallel global routers have been developed to reduce the running time of routing. PGRIP [17] is a parallel global router based on integer programming. It partitions the chip area into subregions to form subproblems and solves each subproblem in parallel. Integer programming is a heavyweight tool, which increases the runtime of PGRIP though the code is parallel. NCTU-GR 2.0 [9] is a net-level parallel method. The paper studies race conditions on routing resources and introduces a collision-aware rip-up and reroute solution by adjusting the cost of conflicting routing resources. Han *et al.* [6] implement net-level parallelism on GPUs by identifying and scheduling independent nets to avoid race conditions on routing resources. The level of parallelism and solution quality highly depend on the input circuit. VFGR [15] utilizes both net-level and region-level parallelism and switches between them according to the bounding box of the nets.

IV. Parallel Maze Routing Implementation

Our global router SPRoute is based on FastRoute 4.1 [13]. We parallelized the maze routing stage, which is the most time-consuming stage in the design flow. The parallelized maze routing has two phases: net-level parallelism and fine-grained parallelism as shown in Algorithm 1.

Algorithm 1 Maze routing

```
Require: A set of nets N and a grid graph $G(V, E)$
Ensure: Routing solution of each net n in N
for each net n in N do                    # net-level parallelism
  for each two-pin net e of n do
    if $ripup\_check(G, e)$ then
      $ripup(G, e)$
      $WL.push(e.source\_nodes)$
      while $!WL.empty()$ do   # fine-grain parallelism
        $node = WL.pop()$
        $neighbors = maze\_expand(G, node)$
        $WL.push(neighbors)$
      end while
      $trace\_back(G, e)$
      $adjust\_tree(n, e)$
    end if
  end for
end for
```

During maze routing, a 2-pin net is checked to see if there is overflow on the edges it uses (line 3). If so, it is ripped up by subtracting the usage on the edges and its source nodes are pushed into the worklist WL (lines 4 and 5). The worklist WL stores the boundary of the maze expansion. The boundary is iteratively expanded until no node is left in the worklist (lines 6 to 9). Finally, the net traces back on the grid graph to generate a routing solution and updates the usage in the grid graph (line 11) before adjusting the tree structure if necessary (line 12).

In Algorithm 1, lines 1, 2 and 6 comprise three nested loops that are possible candidates for parallelization. Section IV-A describes net-level parallelism to parallelize line 1. Section IV-B describes fine-grain parallelism to parallelize line 6. Line 2 iterates on 2-pin nets of a net. The source and destination nodes depend on the remaining 2-pin nets of the same net, thus there are dependencies at the 2-pin nets level, making this loop difficult to parallelize.

A. Net-level Parallelism

Figure 5 shows the implementation of net-level parallelism. All threads share a global grid that contains the usage information on the grid edges. Threads atomically subtract the usage during rip-up or add to the usage when establishing a route. Each thread has a local grid that stores all the information on the grid node, including its distance from the source node, the parent node from which the boundary is expanded, etc.

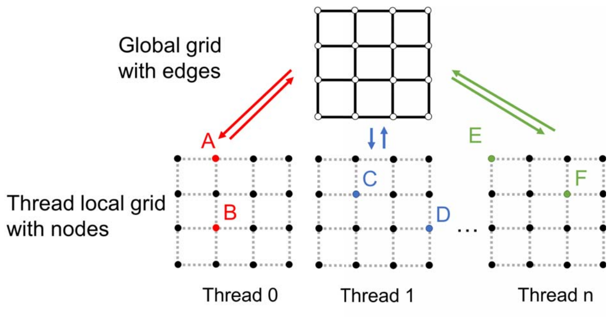


Fig. 5: Net-level Parallelism

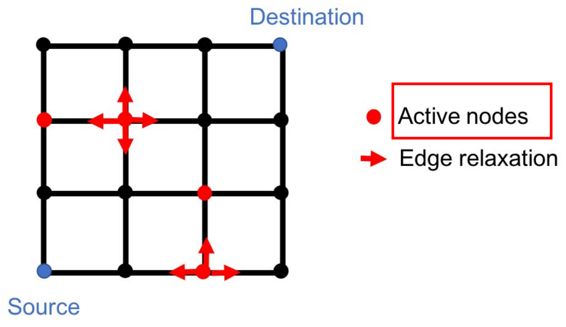


Fig. 6: Fine-grain Parallelism

During maze routing, each thread executes lines 3 to 12 in Algorithm 1. Lines 3 and 8 both read the usage of the edges from the global grid and make decisions based on the usage. Line 3 determines if the net should be ripped up and line 8 determines if the neighbors should be pushed into the worklist. Since we do not apply locks to the routing regions, the usage read by a thread may be stale, causing livelock, as explained before. Thus, the solution of net-level parallelism is nondeterministic.

After maze routing is done, we compare the total overflow of this iteration with the previous iteration. Once the total overflow stops decreasing, we interpret it as evidence of livelock and reduce the number of threads in net-level parallelism. If livelock still exists, we finally switch to the fine-grain parallelism phase.

B. Fine-grain Parallelism

The fine-grain phase parallelizes line 6 in Algorithm 1, as illustrated in Figure 6. In fine-grain parallelism, a node on the grid graph is called active when it is in the worklist. The parallel loop starts from a set of source nodes in the worklist. During the expansion, each thread first pops a node from the worklist and acquires the lock for itself and its four neighbors. Then it relaxes the edges of the node to compute the new distance of the neighbors. The distance values are updated if the new distance is less than the neighbor's current distance.

As discussed in Section II-D, an algorithm has a local view and a global view in the operator formulation. In maze routing, the local view (operator) is the edge relaxation and boundary expansion. The global view is the schedule of active nodes, i.e.,

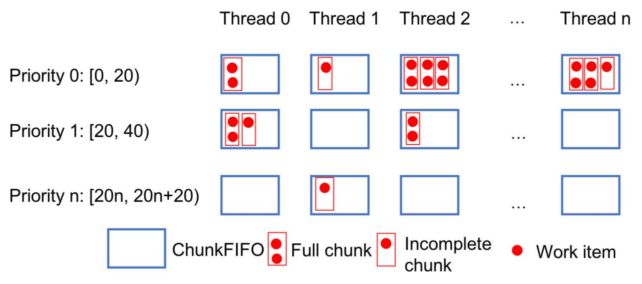


Fig. 7: OrderedByIntegerMetric (OBIM) scheduler with per thread chunked FIFO

which node has the highest priority to be popped out from the worklist and executed first. In sequential maze routing, nodes with a shorter distance have higher priority and thus a priority queue is generally used. However, priority queues are not good concurrent priority schedulers [8]. In Galois, a scalable priority scheduler called OrderByIntegerMetric (OBIM) is provided to support *soft* priorities [12].

1) Scheduler Design: The scheduler of the fine-grain parallel phase uses OBIM with PerThreadChunkFIFO, which provides thread locality to OBIM.

OBIM partitions the priority range into subranges and assigns each subrange the same priority level by a user specified indexer. The order within each priority level and its detailed implementation are specified by another secondary worklist. In our fine-grain parallel design, we use the $distance/OBIM\_delta$ rounded to an integer as the indexer, and PerThreadChunkFIFO as the secondary worklist. $OBIM\_delta$ is a function that shares the same parameters as the cost function on edges and changes across iterations as the cost function changes.

As an example, Figure 7 shows the structure of OBIM with an $OBIM\_delta$ of 20 and PerThreadChunkFIFO with a chunk size of 2. The highest priority is priority 0. In OBIM with PerThreadChunkFIFO, there is a ChunkFIFO for each thread in each priority level. A ChunkFIFO maintains a queue for chunks of work items. A full chunk can be stolen by other threads for load balancing while an incomplete chunk cannot.

In maze routing, when a thread is pushing a node into the worklist, it calculates the integer index of the node and pushes the node into a chunk in the corresponding ChunkFIFO. When a thread is popping a node from the worklist, it reads the highest priority level that has work items among all threads and then checks if there are work items in its local priority level. If so, it pops a chunk from the local ChunkFIFO and executes it. Otherwise, it steals a full chunk from another thread. Since OBIM maintains soft priorities in the worklist, the order in which nodes are popped from the worklist is nondeterministic. Therefore, the exploitation of fine-grain parallelism is nondeterministic and may produce different solutions in different runs with the same inputs.

2) Work Reduction: In maze routing, a grid node can be pushed into the worklist multiple times and thus introduce extra work. To reduce the extra work, we push the node as well as its distance to the worklist. While the node is in the worklist, a shorter path may be found and a duplicate of the node with a smaller distance is pushed into the worklist. In this case, the distance on the grid is updated to the smaller one. When a node is popped out, it aborts if the push-time distance is not equal to the distance on the grid. This optimization eliminates more than half of the work in some cases.

Line 6 in Algorithm 1 shows that the fine-grain parallel loop terminates when there are no items left in the worklist. However, the maze expansion can terminate when the shortest path is found even though the worklist is not empty. We implement the early termination by recording the minimum length of the paths that reaches the destination. If the worklist pops a node with distance above the recorded distance plus $OBIM\_delta$, which indicates that no active node in the worklist has the same priority as the recorded distance, then it is safe to exit routing of the current net.

V. Experimental Results

SPRoute is implemented in C++ on Galois 4.0 [5]. We evaluate SPRoute on the ISPD 2008 global routing contest [1] benchmark suite on a 28-core 2.2 GHz Intel Xeon Gold machine with 196 GB of memory. The machine has two sockets, i.e., 14 cores per socket. We adopt the same criterion as NCTU-gr 2.0 [9] and classify the inputs into two categories: overflow-free cases and hard-to-route cases.

In the experimental results, we first show convergence behavior and scalability of SPRoute to explain performance. Then we compare the total overflow, wirelength and runtime of SPRoute with these statistics for two state-of-the-art academic global routers, FastRoute 4.1 [13] and NCTU-gr 2.0 [9], both of which are sequential. The total wirelength and the number of vias are given in units of 10^5.

A. Convergence Behavior

In SPRoute, maze routing is called iteratively until the overflow converges to 0. One iteration of maze routing denotes one pass to rip up and reroute all the nets. Figure 8 shows the convergence behavior of different execution strategies for the *bigblue1* input from the ISPD 2008 routing contest. These execution strategies are (i) sequential execution, (ii) fine-grain parallel execution, (iii) net-level parallel execution with 8 threads, (iv) net-level parallel execution with 28 threads, and (v) SPRoute.

The figure shows that for $bigblue1$, fine-grain parallel execution and sequential execution behave similarly, and their overflow decreases to zero in iteration 15. The fine-grain parallel execution strategy is net-level sequential so it does not encounter problems with livelock. Net-level parallel execution does not converge in 28 iterations for both 8 threads and 28 threads. The overflow of net-level parallel execution with 28 threads stops decreasing at iteration 9 and remains high. The overflow of net-level parallel execution with 8 threads is close to zero after iteration 9. However it does not decrease to zero even after 28 iterations because of livelock. This illustrates the trade-off between parallelism and convergence in net-level parallel execution.

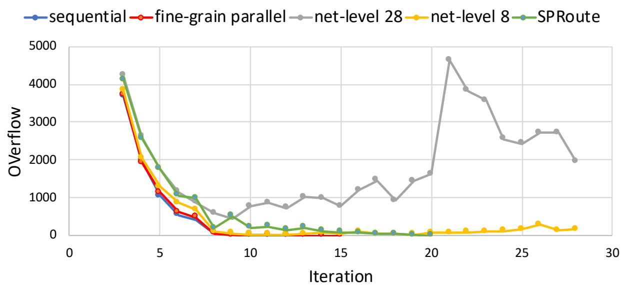


Fig. 8: Convergence relationship between total overflow and iteration number on the $bigblue1$ input

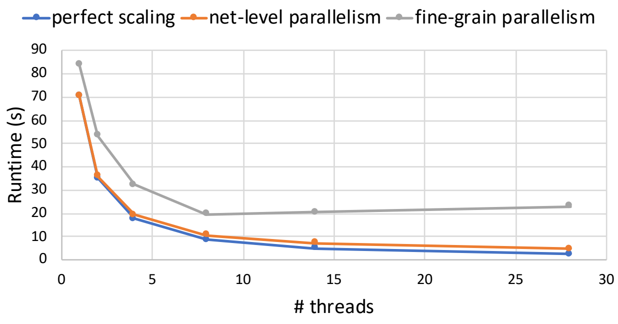


Fig. 9: Runtime of the first maze routing iteration on the $newblue3$ input

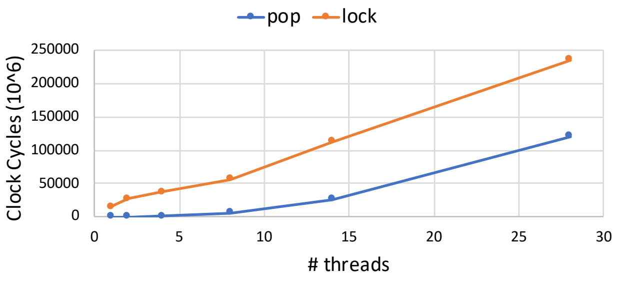


Fig. 10: Clock Cycles of Pop and Lock

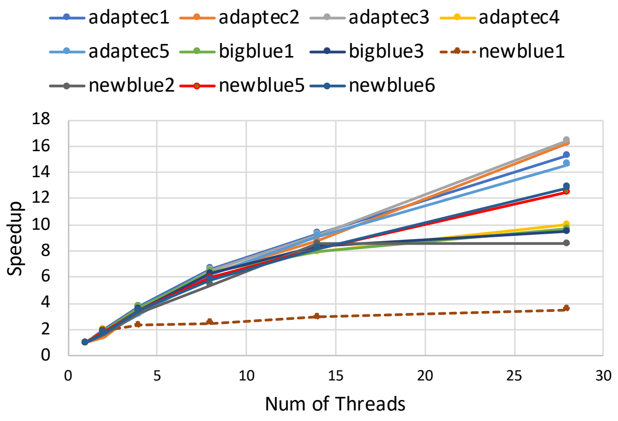


Fig. 11: Scalability on Overflow-free Cases

SPRoute implements a hybrid scheme that achieves a good speedup with net-level parallel execution and guarantees convergence by switching to fine-grain parallelism when livelock is detected. In Figure 8, the overflow of SPRoute until iteration 7 is similar to that of net-level parallel execution with 28 threads. When livelock is identified, SPRoute lowers the level of parallelism in net-level parallel execution and finally switches to fine-grain parallelism. SPRoute converges in iteration 20.

B. Scalability

1) Scalability of Fine-grain Parallel Execution: Figure 9 shows the runtime of the first maze routing iteration on $newblue3$. Net-level parallel execution provides close to perfect scaling but fine-grain parallel execution flat-lines after 8 threads. Since 8 threads provide the best speedup ($4.3\times$), we use 8 threads in SPRoute when switching from net-level parallelism to fine-grain parallelism.

We use Intel's Vtune Amplifier [7] to profile the first maze routing iteration on $newblue3$ and investigate the reasons of the limited speedup in fine-grain parallelism. Figure 10 shows the clock cycles of two major time-consuming functions. Pop denotes the cost of popping nodes from the worklist plus the cost of waiting if there are not enough nodes in the worklist. $Lock$ denotes the cost of locking neighbors plus the cost of waiting if any of them are already locked. The clock cycles shown in the figure are the aggregated clock cycles of all threads. As the number of threads increases, the clock cycles of a scalable function should remain the same whereas the clock cycles increase if the function is not scalable. The figure shows that both pop and $lock$ do not scale. This explains the reason for the limited parallelism in the fine-grain stage.

Since fine-grain parallelism is widely used in graph applications, we compare the number of nodes, number of edges, average parallelism and average worklist size between maze routing and the Single Source Shortest Path (SSSP) problem in Table I. The USA road network is a standard graph for graph analytics studies. We estimate parallelism by using the size of the maximal independent set in the worklist, i.e., the maximal number of work items that can run concurrently. Table I shows that the average parallelism and worklist size for maze routing on $newblue3$ are both several orders of magnitude smaller than for SSSP. The worklist size depends on the size of the routing region. Note that, in this example, the average number of routing edges of a 2-pin net is 105, which indicates that the maze expansion only explores a small routing region in the entire 973×1256 grid and thus the worklist size is small.

2) Scalability of SPRoute: Figure 11 shows the scalability of SPRoute on overflow-free benchmarks. The average scaling on 28 threads is 11.5. Ten of the eleven inputs achieve more than $8\times$ speedup. On $newblue1$, SPRoute does not provide good scalability because it requires a large number of maze routing iterations to converge. In our two-phase implementation, most of the time is spent in the fine-grain phase, which provides limited scalability.

	Grid	#Nodes	#Edges	Avg. Parallelism	Avg. Worklist Size
Maze routing on $newblue3$	973×1256	1222088	2441947	575	12456
SSSP on $USA\ road\ graph$		23947347	57708624	743172	2415366

TABLE I: Average Parallelism and Worklist Size

C. Comparison with Academic Global Routers

Table II shows the wirelength, number of vias, and overall maze routing time of five global routers on the overflow-free cases. The standard deviation is provided to account for the non-determinism in parallel SPRoute. The wirelength and the number of vias are consistent with 0.06% and 0.05% standard deviations respectively. The runtime of parallel SPRoute has an average standard deviation of 8.7% because it depends on when the switch is made between the two phases.

To build a deterministic global router, we can either execute independent nets in rounds or assign priorities to nets to remove the randomness in resolving livelocks. However, as discussed in Section II, most nets overlap with each other and these two methods will reduce speedup. In our ongoing work, we are studying how to build a deterministic and fast maze router.

The row denoted $Avg.\ Ratio$ shows the overall wirelength, vias and runtime normalized to those of parallel SPRoute. Smaller ratios are better. Parallel SPRoute achieves an $11.5\times$ speedup with a 0.6% wirelength penalty compared to sequential SPRoute and a $11\times$ speedup with a 0.6% wirelength increase against FastRoute 4.1. NCTU-gr 2.0 in fast mode focuses on improving the runtime with acceptable wirelength increase, and is outperformed by SPRoute by a wirelength of 2% and a speedup of $8.4\times$. NCTU-gr 2.0 in regular mode is outperformed by a number of vias of 2% and a speedup of $10.0\times$.

Table III shows the result for the hard-to-route cases. TOF and MOF denote the total overflow and the maximum overflow of all edges. WL and VIA denote the wirelength and the number of vias. Parallel SPRoute has a $4.5\times$ speedup with a total overflow penalty of 0.7% and a wirelength penalty of 0.5% over sequential execution, and it has a $3.1\times$ speedup with a total overflow penalty of 7% and a wirelength penalty of 0.6% relative to FastRoute 4.1. On the hard-to-route cases, the fine-grain parallel phase accounts for most of the runtime and thus the speedup is limited by the available parallelism in the fine-grain phase.

Compared to NCTU-gr 2.0, SPRoute uses 0.3% more wirelength and achieves $2.48\times$ less total overflow than regular mode, and achieves 3% less wirelength and $5.17\times$ less total overflow than fast mode. The runtime of SPRoute and NCTU-gr 2.0 are not comparable on the hard-to-route cases because the termination condition of maze routing is different.

VI. Conclusion

This paper describes how parallelism can be exploited to reduce the high runtime of the maze routing stage in global routing. We propose a novel two-phase parallel technique that initially uses net-level parallelism and then steps down the parallelism when livelock is identified. If livelock still exists, SPRoute ultimately switches to fine-grain parallelism to guarantee convergence. SPRoute, our scalable parallel global router, is implemented in the Galois system and evaluated on the ISPD 2008 global routing benchmark suite. On overflow-free benchmarks, SPRoute achieves an average speedup of $11.5\times$ and $11\times$ with only 0.6% wirelength penalty over sequential SPRoute and FastRoute 4.1, respectively. On hard-to-route benchmarks, SPRoute achieves an average speedup of $4.5\times$ and $3.1\times$ over sequential SPRoute and FastRoute 4.1, respectively.

References

[1] ISPD 2008. Ispd 2008 global routing contest, http://www.ispd.cc/contests/08/ispd08rc.html.

[2] Yen-Jung Chang, Yu-Ting Lee, and Ting-Chi Wang. Nthu-route 2.0: a fast and stable global router. In *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 338–343. IEEE Press, 2008.

[3] Minsik Cho, Katrina Lu, Kun Yuan, and David Z Pan. Boxrouter 2.0: Architecture and implementation of a hybrid and robust global router. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 503–508. IEEE Press, 2007.

[4] Chris Chu. Flute: fast lookup table based wirelength estimation technique. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 696–701. IEEE Computer Society, 2004.

[5] Galois. Galois 4.0, https://github.com/intelligentsoftwaresystems/galois.

[6] Yiding Han, Dean Michael Ancajas, Koushik Chakraborty, and Sanghamitra Roy. Exploring high-throughput computing paradigm for global routing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(1):155–167, 2014.

[7] Intel. Vtune, https://software.intel.com/en-us/vtune.

[8] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Priority queues are not good concurrent priority schedulers. In *European Conference on Parallel Processing*, pages 209–221. Springer, 2015.

[9] Wen-Hao Liu, Wei-Chun Kao, Yih-Lang Li, and Kai-Yuan Chao. Nctu-gr 2.0: Multithreaded collision-aware global routing with bounded-length maze routing. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 32(5):709–722, 2013.

[10] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Field-Programmable Gate Arrays, 1995. FPGA'95. Proceedings of the Third International ACM Symposium on*, pages 111–117. IEEE, 1995.

[11] Michael D Moffitt. Maizerouter: Engineering an effective global router. *IEEE Transactions on Computer-Aided*

TABLE II: Performance on Overflow-free Cases

Benchmark	Sequential SPRoute			Parallel SPRoute			FastRoute 4.1			NCTU-gr 2.0 Fast			NCTU-gr 2.0 Regular		
	WL	VIA	Time(s)	WL	VIA	Time(s)	WL	VIA	Time(s)	WL	VIA	Time(s)	WL	VIA	Time(s)
adaptec1	36.3	17.3	113.4	36.9	17.9	7.4	36.4	17.3	112.5	37.8	20.4	18.2	36.2	18.2	70.8
adaptec2	33.3	18.9	20.2	33.4	18.9	1.3	33.3	18.9	18.9	34.0	20.0	11.1	33.3	19.4	13.3
adaptec3	96.6	34.4	83.0	96.8	34.3	5.1	96.7	34.4	77.0	97.8	38.5	21.1	96.0	35.1	79.0
adaptec4	89.9	31.3	4.5	89.9	31.3	0.5	89.9	31.3	4.4	90.3	32.8	11.7	89.9	32.1	5.2
adaptec5	104.0	51.7	200.1	104.5	52.4	13.7	104.1	51.7	193.9	107.9	58.3	61.9	105.6	53.9	269.0
bigblue1	37.9	18.7	148.6	39.1	19.7	15.3	37.9	18.7	146.9	41.6	23.4	40.3	39.2	20.5	160.9
bigblue3	79.0	51.1	35.8	79.2	51.4	3.7	79.0	51.1	31.4	80.4	53.3	21.8	79.4	52.3	41.3
newblue1	24.5	21.9	554.9	24.6	22.0	158.0	24.5	21.8	371.0	25.0	23.1	46.0	24.6	22.4	97.3
newblue2	46.7	28.4	2.1	46.7	28.4	0.3	46.7	28.4	2.3	46.8	29.8	8.5	46.6	29.3	2.1
newblue5	148.7	82.2	233.6	149.2	82.6	18.7	148.8	82.1	223.0	152.8	89.6	72.4	149.3	85.3	147.4
newblue6	103.7	73.8	302.2	104.5	74.9	23.5	104.3	73.8	266.7	110.5	82.8	53.5	105.3	77.5	144.4
Avg. Ratio	0.994	0.988	**11.5**	1.0	1.0	**1.0**	0.994	0.988	**11.0**	1.02	1.08	8.4	1.00	1.02	10.0
stdev				0.06%	0.05%	8.7%									

TABLE III: Performance on Hard-to-route Cases

Benchmark	Sequential SPRoute					Parallel SPRoute				
	TOF	MOF	WL	VIA	Time(s)	TOF	MOF	WL	VIA	Time(s)
bigblue4	138	2	121.4	108.9	5231	140	2	121.4	109.0	1532
newblue3	31270	359	77.4	31.5	1001	31242	363	78.8	32.5	117
newblue4	138	1	82.9	47.7	4411	140	1	82.8	47.9	1142
newblue7	76	2	185.5	167.9	34625	76	2	186.5	167.9	16030
Avg. Ratio	0.993	0.997	0.995	0.991	**4.5**	1	1	1	1	**1**
stdev						1.6%	0.49%	0.06%	0.04%	13.3%

	FastRoute 4.1				
	TOF	MOF	WL	VIA	Time(s)
bigblue4	138	1	121.4	108.9	4108
newblue3	31284	374	77.0	31.4	587
newblue4	136	1	82.9	147.6	3269
newblue7	58	2	186.4	166.9	28044
Avg. Ratio	0.93	0.882	0.994	0.988	**3.1**

	NCTU-gr 2.0 Fast					NCTU-gr 2.0 Regular				
	TOF	MOF	WL	VIA	Time(s)	TOF	MOF	WL	VIA	Time(s)
bigblue4	1056	8	127.1	113.4	84.7	512	2	122.0	109.7	173.3
newblue3	36850	788	80.0	32.8	160.1	37182	788	76.9	21.1	37467.1
newblue4	216	2	84.9	49.5	53.4	172	2	83.0	46.91	112.8
newblue7	792	2	191.3	170.8	313.5	286	2	187.2	165.7	1478
Avg. Ratio	5.17	2.29	1.03	1.03	0.37	2.48	1.54	0.997	0.998	80.1

Design of Integrated Circuits and Systems, 27(11):2017–2026, 2008.

[12] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[13] Min Pan, Yue Xu, Yanheng Zhang, and Chris Chu. Fastroute: An efficient and high-quality global router. *VLSI Design*, 2012:14, 2012.

[14] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. The tao of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46, pages 12–25. ACM, 2011.

[15] Zhongdong Qi, Yici Cai, Qiang Zhou, Zhuoyuan Li, and Mike Chen. Vfgr: A very fast parallel global router with accurate congestion modeling. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 525–530. IEEE, 2014.

[16] Jarrod A Roy and Igor L Markov. High-performance routing at the nanometer scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1066–1077, 2008.

[17] Tai-Hsuan Wu, Azadeh Davoodi, and Jeffrey T Linderoth. A parallel integer programming approach to global routing. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 194–199. IEEE, 2010.