ParaDiMe: A Distributed Memory FPGA Router Based on Speculative Parallelism and Path Encoding

Chin Hau Hoo
Department of Electrical & Computer Engineering
National University of Singapore
Singapore
chinhau.hoo@u.nus.edu

Akash Kumar
Center for Advancing Electronics
Technische Universität Dresden
Dresden, Germany
akash.kumar@tu-dresden.de

***Abstract*—The increase in speed and capacity of FPGAs is faster than the development of effective design tools to fully utilize it, and routing of nets remains as one of the most time-consuming stages of the FPGA design flow. While existing works have proposed methods of accelerating routing through parallelization, they are limited by the memory architecture of the system that they target. In this paper, we propose a distributed memory parallel FPGA router called ParaDiMe to address the limitations of existing works. ParaDiMe speculatively routes net in parallel and dynamically detects the need to reduce the number of active processes in order to achieve convergence. In addition, the synchronization overhead in ParaDiMe is significantly reduced through a careful design of the messaging protocol where paths to sinks are encoded in a space-efficient manner. Moreover, the frequency of synchronization is tuned to ensure convergence while minimizing the communication overhead. Compared to VTR, ParaDiMe achieves an average speedup of 19.8X with 32 processes while producing similar quality of results.**

I. Introduction

The increasing speed and capacity of FPGA have been made possible by Moore's law [1]. However, there has not been a matching increase in design productivity. As a result, there is a gap between the capacity of FPGAs and the ability to utilize it effectively. The gap increases the development cost and time-to-market of FPGA-based designs, and one of its main causes is inadequate FPGA design tools. While there are front-end design improvements such as IP-based design methodology and high-level synthesis to bridge the gap, back-end designs such as routing still remain as some of the most time-consuming stages of the FPGA design flow.

It is possible to speed up FPGA routing through parallelization but it is non-trivial due to the nature of the underlying algorithm [2]. Most FPGA routers are based on the classic Pathfinder algorithm [3], which resolves congestion by iteratively increasing the congestion cost of overused routing resources (RRs). In order to reach a congestion-free state when routing multiple nets in parallel, it is important for the congestion costs to remain synchronized among threads or processes to avoid misusing resources in congested regions. Existing shared and distributed memory parallel routers employ different methods of synchronizing the congestion costs depending on the memory architecture of the system that they run on. The synchronization overhead of shared memory routers is lower than that of distributed memory routers but the relatively low number of processors in shared memory systems limit the maximum achievable speedup of such routers. On the other hand, distributed memory parallel routers are not constrained in terms of computing resources but suffer from high synchronization overhead due to the high remote memory access latency.

In this paper, we propose a distributed memory FPGA router called ParaDiMe that speculatively routes net in parallel with various techniques to address the aforementioned problem of high synchronization overhead. Our contributions are summarized as follows.

- Speculative parallelism in a distributed memory architecture with active process reduction to achieve high speedup while ensuring convergence
- High-frequency synchronization to ensure that congestion costs remain updated among processes
- Path encoding and route tree memoization to significantly reduce the synchronization overhead

The rest of the paper is organized as follows. Section II and III give an overview about FPGA routing and its related works. Section IV, V, VI, and VII present the design of ParaDiMe. Section VIII and IX describe the experiment methodology and evaluation results. Section X concludes the paper.

II. Background

The routing resources in an FPGA are usually modeled using a graph G(V, E) where V is the set of vertices (nodes) that represent the routing resources (pins and prefabricated wires) and E is the set of edges that represent the programmable switches connecting the routing resources. Associated with each vertex is a capacity that indicates the number of nets that can use a routing resource simultaneously. The routing problem can then be seen as a problem to find for each net, a minimum delay tree, which is a subgraph of G, spanning the source and sinks of the net. In addition, the union of all the trees must not result in overuse of RR nodes.

Most FPGA routers such as VTR [4] solve the problem using a variant of the Pathfinder algorithm [3]. The Pathfinder algorithm works by gradually increasing the congestion cost of using an overused RR node until all overuses are eliminated. The congestion cost consists of two components – present (first order) and historical (second order). The first order cost is updated after routing each net while the second order cost is updated at the end of every routing iteration. The second order cost is crucial to ensure that the router converges to a congestion-free state. On the other hand, the first order cost poses a significant challenge to parallelizing FPGA routers. Since it is updated after routing every net, the cost that is seen when routing a net depends on the route taken by nets that have been routed previously. In other words, the first order cost introduces dependency among the nets.

III. Related Works

Despite the dependency imposed by the first order cost, there are two ways in which existing works extract parallelism from the Pathfinder algorithm. The first approach is by routing nets with non-overlapping bounding boxes in parallel [5], [6]. This method works because nets are only allowed to use routing resources that are within their bounding boxes. Therefore, nets with non-overlapping bounding boxes do not depend on one another and can be routed in parallel. The second approach is to relax the independent bounding box requirement and speculatively route the nets in parallel [7]. When a contention on a routing resource is detected, the route progress of the offending threads/processes is discarded, and routing is restarted.

An example of the independent bounding box approach is by Gort and Anderson [5] who proposed a distributed memory version of the Pathfinder algorithm. Congestion costs are synchronized among processors using the message passing interface (MPI), and they are sent in a non-blocking manner to other processors once a processor finishes routing a net. Gort and Anderson also tried to further reduce the communication overhead by spatially partitioning the FPGA. Nets whose bounding boxes are entirely within a partition (intra-partition nets) can be routed without the need to communicate with other processors while nets whose bounding boxes span multiple partitions (inter-partition nets) need to synchronize with other processors. Their router achieved a speedup of 2.85X with 8 processes.

Shen and Luo [6] also proposed a distributed memory parallel router based on spatial partitioning that achieved a speedup of 7.06X with 32 processes. In their approach, nets spanning multiple partitions are routed sequentially before nets that are entirely within a partition are routed in parallel. MPI is used to distribute nets to the processors and transfer route trees.

TDR [8] is also a distributed memory parallel router where the routing resource graph is partitioned into disjoint sets. Each processor is allocated a partition of the RR graph and a set of nets to be routed. Since the RR partitions are independent due to the properties of the disjoint switch box topology that TDR specifically targets, there is no need to synchronize congestion costs among processors during routing, and MPI is used only to distribute nets to the processors.

Chan et al [9] analyzed how often the first and second order congestion costs should be synchronized among processors and built a distributed memory parallel router based on the analysis. In their approach, the first order congestion costs are synchronized as soon as a net is routed while the second order congestion costs are synchronized only at the end of a routing iteration.

The shared memory parallel router by Moctar and Brisk [7] is an example of the aforementioned speculative approach. Instead of being restricted to partitioning [5], [6], [8], multiple threads speculatively perform VTR's [4] maze expansion by using thread-safe priority queues. The approach has a good speedup of 5.46X with 8 threads.

Hoo et al [10] modeled the routing problem as a linear program and decomposed it into independent sub-problems through Lagrangian relaxation. The sub-problems were solved in parallel in a shared memory system, yielding a high speedup of 7.05X with 8 threads. However, the method was shown to be effective for the global routing problem instead of the detailed routing problem. Another shared memory parallel router by Hoo et al [11] uses a combination of fine-grained synchronization and partitioning to achieve a significant speedup of 26.2X with 8 threads relative to VTR. The super-linear speedup is achieved because only congested nets are rerouted when there is difficulty resolving congestion.

Algorithm 1 ParaDiMe

```
 1: function PARADIME($N$, $repartition$)
 2:     $i \leftarrow 0$
 3:     $routed \leftarrow$ false
 4:     $idle \leftarrow$ false
 5:     $prev\_n\_overused \leftarrow \infty$
 6:     $n\_ranks \leftarrow$ MPI_Comm_size()
 7:     $rank \leftarrow$ MPI_Comm_rank()
 8:     $route\_time \leftarrow \{\}$
 9:     $P \leftarrow$ partition($N$, $n\_ranks$, $route\_time$)
10:     while $i < max\_iterations$ and $!routed$ and $!idle$ do
11:         if $i == 1$ and $repartition$ then
12:             $P \leftarrow$ partition($N$, $n\_ranks$, $route\_time$)
13:         end if
14:         for $net \in P[rank]$ do
15:             Rip up $net$
16:             Broadcast rip up message of $net$      ▷ Section VII-A2
17:             Non-blocking check for messages        ▷ Section VI
18:             route_net($net$)                       ▷ Algorithm 2
19:             $route\_time[net] \leftarrow$ get_route_time($net$)
20:         end for
21:         Broadcast trailer message                  ▷ Section VII-A3
22:         while not received trailer message from other ranks do
23:             Blocking check for messages            ▷ Section VI
24:         end while
25:         Update second order congestion costs
26:         $n\_overused \leftarrow$ get_n_overused_nodes()
27:         if $n\_overused == 0$ then
28:             $routed \leftarrow$ true
29:         else if $n\_overused > prev\_n\_overused$ and $n\_ranks > 1$ then
30:             $cnets \leftarrow$ get_congested_nets($P$)
31:             $n\_ranks \leftarrow n\_ranks/2$
32:             $P \leftarrow$ partition($cnets$, $n\_ranks$, $route\_time$)
33:             if $rank \geq n\_ranks$ then
34:                 $idle \leftarrow$ true
35:             end if
36:         end if
37:         $prev\_n\_overused \leftarrow n\_overused$
38:         $i \leftarrow i + 1$
39:     end while
40: end function
```

IV. ParaDiMe

The distributed memory parallel routers mentioned in the previous section are based on either fine-grained or coarse-grained partitioning of routing resources. However, there are drawbacks to using partitioning for parallel routing. Routers based on fine-grained partitioning [8] work only on a very specific FPGA architecture because independent track domains are formed (trivially) only when the switch box topology is disjoint. On the other hand, routers based on coarse-grained partitioning [5], [6] are not scalable because the number of

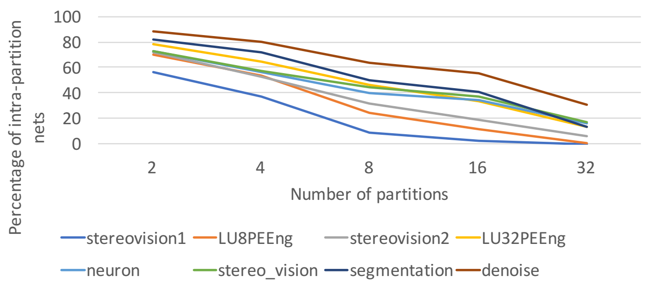

Fig. 1: Percentage of intra-partition nets versus number of partitions

intra-partition nets decreases as the number of spatial partitions increases as shown in Figure 1. In other words, the amount of parallelism decreases as the number of partitions increases. While the figure assumes that the FPGA is partitioned into equally sized segments, the same trend of decreasing number of intra-partition nets is observed for the partitioning described in [5] and [6].

In order to overcome the limitations of partitioning, we propose a new distributed memory parallel router called ParaDiMe where each process speculatively routes a disjoint subset of all the nets without partitioning. In ParaDiMe, each process has a local copy of the first order congestion costs, and it is synchronized using MPI's non-blocking point-to-point communication (MPI_Isend and MPI_Irecv) at different stages of the routing process. Non-blocking communication is used to allow overlap between communication and computation, which improves speedup.

Algorithm 1 shows the pseudocode of ParaDiMe, and every process runs the same copy of it. Therefore, the algorithm can be understood by viewing it from the perspective of a single process. It is important to note that the point of Algorithm 1 is to highlight the features of ParaDiMe. Therefore, unnecessary information is omitted if it does not affect the description accuracy of the actual algorithm.

ParaDiMe starts by partitioning the set of nets to be routed, N into n_ranks number of partitions, which equals to the number of processes running the router. During partitioning, the netlist is sorted in decreasing number of sinks before being distributed in a round-robin manner to n_ranks processes. Sorting the nets in this order achieves a relatively good load balancing as shown in Section IX-D. If $repartition$ is set to true, the netlist is also repartitioned in the second iteration (Line 12). The repartitioning aims to reduce the load imbalance among the processes in the case where the number of sinks, which was used in the initial partitioning, is not a good measure for the actual route time. During repartitioning, the nets are sorted in decreasing order of their route time, which was obtained in the first iteration.

After partitioning the input netlist, each process starts routing the nets (Line 11 - 38) that are assigned to it until one of these three conditions is satisfied – there are no overused RR nodes, the maximum number of iterations is reached or the current process is no longer active.

For each net, the existing route tree of the net is ripped up, and the event is broadcasted to other processes based on the protocol described in Section VII-A2. Then, the process checks for incoming messages from other processes, and the local first order congestion costs are updated based on the content of the messages. Then, the route_net function is called to route the net, and its route time is stored to be used for repartitioning in the second iteration.

After routing every net, a trailer message is broadcasted to indicate that the current process has finished routing its assigned nets. The importance of the trailer message is discussed in Section VII-A3. After broadcasting the trailer message, the process check for messages from other processes so that by Line 25, the first order congestion costs of all processes are consistent. The check is blocking because the process has finished routing its nets. After ensuring that the first order congestion costs are consistent, the second order congestion costs are updated as well.

Finally, ParaDiMe checks whether there are overused RR nodes (Line 27). In the case where there is none, the routing process is completed. If there are overused nodes, the current number of such nodes is compared with that of the previous iteration to determine if reducing the number of active processes is necessary. If the reduction is not required, ParaDiMe continues routing nets in the next iteration with the same number of processes. The number of processes is reduced by half when the number of overused nodes stops decreasing monotonically, which is a sign that the router is having difficulty resolving congestions. The difficulty is caused by outdated first order congestion costs during routing, and it is more pronounced when the router is in an almost congestion-free state. To mitigate the outdated costs, the number of active processes is cut by half. In addition, only congested nets from the current iteration are rerouted after the reduction. The congested nets are partitioned in the same way as the repartitioning in Line 12 in that the congested nets are sorted in decreasing order of their route time. The active process reduction and congested nets distribution is illustrated in Figure 2.

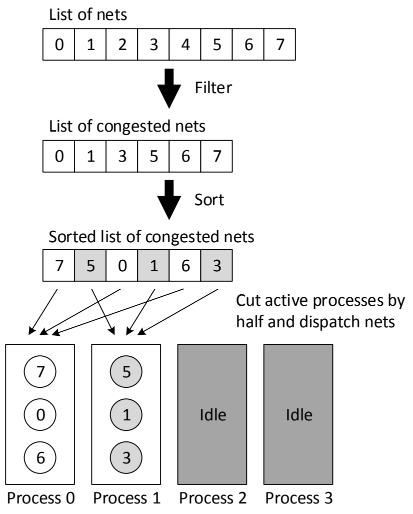

Fig. 2: Active process reduction and congested nets distribution

V. Net Router

When multiple nets are routed in parallel speculatively as described in the previous section, it is crucial for a processor to

Algorithm 2 Net Router

```
 1: procedure ROUTE_NET($net$)
 2:     $min\_heap \leftarrow \{\}$
 3:     $route\_tree \leftarrow \{$source of $net\}$
 4:     for $sink \in net.sinks$ do
 5:         Add $route\_tree$ to $min\_heap$
 6:         $count \leftarrow 0$
 7:         while $!min\_heap$.empty() and $!found\_sink$ do
 8:             $current \leftarrow min\_heap$.pop()
 9:             if $current == sink$ then
10:                 $found\_sink \leftarrow$ true
11:             else if $current$.cost $< state[current]$.cost then
12:                 $state[current]$.cost $\leftarrow current$.cost
13:                 for $n \in current$.neighbors do
14:                     $c\_cost \leftarrow$ get_congestion_cost($n$)
15:                     $t\_cost \leftarrow$ get_timing_cost($current$, $n$)
16:                     $n$.cost $\leftarrow$ get_total_cost($c\_cost$, $t\_cost$)
17:                     $min\_heap$.push($n$)
18:                 end for
19:                 if $count > 0$ and $count \% FREQ == 0$ then
20:                     Non-blocking check for messages    ▷ Section VI
21:                 end if
22:                 $count \leftarrow count + 1$
23:             end if
24:         end while
25:         Add path to $sink$ to $route\_tree$
26:         for $node \in$ path to $sink$ do
27:             Update first order congestion cost of $node$
28:         end for
29:         Broadcast route message                    ▷ Section VII-A1
30:         Non-blocking check for messages            ▷ Section VI
31:         $min\_heap \leftarrow \{\}$
32:         for $node \in$ modified RR nodes do
33:             $state[node]$.cost $\leftarrow \infty$
34:         end for
35:     end for
36: end procedure
```

read the most updated first order costs to achieve convergence. Existing works [5], [9] synchronize the first order costs only after routing every net instead of every sink. Therefore, when routing large nets, a processor might see outdated congestion costs while other processors are routing smaller nets. This could lead to longer time for convergence as illustrated in Section IX-F. In our approach, the first order costs are synchronized while finding a path to the sink and after routing every sink as shown in Algorithm 2.

Algorithm 2 is basically VTR's net router [4] that is modified to synchronize first order congestion costs among processes. The synchronization while finding a path to the sink happens in Line 19-21 where the net router starts a non-blocking check for messages from other processes after expanding the neighbors of every $FREQ$ number of nodes. The local first order congestion costs are then updated based on the received messages. This is to ensure that the local first order congestion costs stay updated even when it takes a long time to find a path to a sink. The synchronization after routing each sink happens in Line 29-30 where the path to the sink is encoded in the format described in Section VII-A1 and broadcasted using MPI_Isend before checking for messages from other processes.

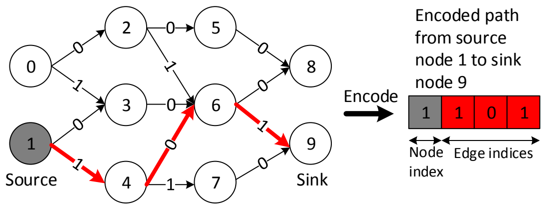

Fig. 3: Path encoding

VI. Receiving Messages

In Algorithm 1 and 2, messages from other processes are received by calling MPI_Irecv. Since the function is non-blocking, its completion is determined by calling MPI_Testsome or MPI_Waitsome. MPI_Testsome returns a flag immediately indicating whether the MPI_Irecv has completed while MPI_Waitsome waits until the MPI_Irecv is completed before returning. When the MPI_Irecv is completed, the message received is handled based on the protocols described in Section VII-A, and another MPI_Irecv is started to continue receiving messages from the same process if it has not sent a trailer message.

VII. Broadcasting Messages

An important factor that affects the speedup of distributed memory parallel router is the amount of data that is transferred among the processors. The more the data transferred, the higher the communication overhead and the lower the speedup. Existing works [5], [6], [9] do not attempt to minimize the message size when synchronizing congestion costs. In ParaDiMe, the message is encoded to significantly reduce its size. Furthermore, we keep track of the route tree of nets routed by other processors so that ripping up those nets can be done without the need for other processors to send the entire route trees again.

In this section, we explain the design of ParaDiMe's messaging protocol and the choice of MPI functions to reduce the communication overhead.

A. Protocol

Each message consists of a header and an optional payload. The header carries a 4-bit Type field and a 28-bit NetIndex field. The Type field indicates the message type while the NetIndex field indicates the net that the message refers to. The three types of messages (RipUp, Route, and Trailer) and the need for the NetIndex field are explained in the following subsections.

1) Route: The Route message is the only message type with a payload. The payload is an encoded version of the path to a sink. The motivation behind encoding the path to a sink is to reduce the size of the Route message and minimize the communication overhead. A path to a sink can be represented as a list of RR nodes. However, sending the list directly as a message is space-inefficient. It is possible to significantly reduce the message size without losing the ability to recover the path information. Since each process has access to the full RR graph, only the first RR node in the path needs to be sent as it is. Each subsequent RR node in the path can be sent as an index of the edge that connects the RR node and its predecessor. An example of how a path can be encoded in shown in Figure 3. The reason this encoding reduces the message size is that the maximum outdegree of a RR graph is significantly smaller than the number of nodes. For example, the RR graph of the FPGA used to implement one of the largest Titan benchmarks (denoise) has around 3.8 million nodes but the maximum outdegree is only 154. Therefore, the number of bits required to store each of the second RR nodes onwards in the path is reduced by a factor of 3 ($log_2(3.8 \times 10^6)/log_2 154$). In general, the compression ratio is $\frac{N log_2(order)}{log_2(order)+(N-1)log_2(max\_outdegree)}$ where $order$ is the number of nodes in the RR graph and N is the number of RR nodes in the path. During initialization, ParaDiMe dynamically calculates the minimum number of bits required to represent the largest edge index in the input FPGA architecture.

When a process receives a Route message, the payload is decoded, and the local first order congestion costs of the RR nodes associated with the decoded path are updated. In addition, the RR nodes are also added to the local route tree of the net indicated by the NetIndex field so that they can be used while processing the RipUp message as described in the next subsection.

2) RipUp: The RipUp message informs a process that a net has been ripped up. It does not have a payload because ParaDiMe keeps track of the route trees of nets that are routed by other processes as described in the previous subsection. Therefore, the NetIndex field from the header is sufficient to tell the receiving process which net to rip up, and the process can easily refer to the corresponding route tree that was stored earlier to update the associated local first order congestion costs.

3) Trailer: Once a process has finished routing its nets, it does not need to receive messages from other processes immediately as long as the congestion costs are consistent at the end of a routing iteration. Therefore, the Trailer message tells other processes that multiple messages can be grouped into one large message before sending. This has the advantage of reducing the number of MPI_Isend calls in other processes while they are routing their remaining nets.

B. MPI function

There are three types of communication in MPI – point-to-point, collective and one-sided. Currently, we have explored both point-to-point and collective communication, and we plan to experiment with one-sided communication, which enables remote memory access, in the future.

There are two MPI collective functions that closely match the requirements of ParaDiMe for broadcasting data – MPI_Ibcast and MPI_Iallgatherv. Theoretically, these functions are optimized for broadcasting by either offloading the work to hardware or falling back to an optimal broadcast tree (eg. binomial tree) when hardware acceleration is unavailable. However, they are restrictive in that one of the input parameters require the knowledge of how much data other processes are going to send. There are two workarounds to this problem. The first workaround is to assume that all processes are sending a constant amount of data. However, it is very difficult to determine a good constant value because it has to be large enough to fit all possible messages but small enough to minimize sending of unnecessary data. The second workaround is a two stage approach where each process broadcast in stage one, the amount of data that it is going to send in stage two. This essentially doubles the amount of required collective function calls. Unfortunately, none of these workarounds yields a good parallel performance.

TABLE I: Summary of benchmarks used in the experiments

Benchmark	Total nets	Total blocks	Minimum channel width
stereovision1	10,797	1,217	104
LU8PEEng	15,990	2,373	114
stereovision2	34,501	2,926	154
LU32PEEng	53,199	7,536	174
neuron	54,056	3,512	206
stereo_vision	61,883	3,434	228
segmentation	125,592	9,047	292
denoise	257,425	18,600	310

Due to the ineffectiveness of collective communication, point-to-point communication is used for ParaDiMe, and the functions involved are MPI_Isend and MPI_Irecv. Unlike MPI_Ibcast and MPI_Iallgatherv, it is easy to send a variable amount of data using MPI_Isend and get the amount of data received using MPI_Get_count.

VIII. Experimental Setup

ParaDiMe was evaluated using a high performance computing (HPC) cluster with 16 nodes. Each node is equipped with two Intel Xeon E5-2680 V3 and 128 GB of RAM. Hyper-threading is disabled by the cluster administrator. The nodes are connected together via Mellanox Technologies MT27600 Infiniband adapters. The operating system is Red Hat Enterprise 6.8 with Linux kernel version 2.6.32. Intel ICC compiler version 16.0.2 with optimization flag O3 was used to compile both ParaDiMe and VTR [4]. The MPI library used for interprocess communication is MVAPICH2 2.2.

The benchmarks used for evaluation were obtained from the VTR [4] and Titan [12] packages, and they are summarized in Table I. Due to the limited amount of time that we have on the HPC cluster, we chose only some of the largest benchmarks from VTR and a subset of the benchmarks from Titan. The Titan benchmarks were chosen to cover a wide range of circuit sizes with $neuron$ being the smallest and $denoise$ being the largest out of the 23 Titan benchmarks. The benchmarks were packed and placed with default parameters using VTR of the same version as the Titan paper [12] (7.0 r4292) because the release version of VTR (7.0) crashes when loading Titan benchmarks due to a netlist loading bug. The architecture files used are k6_frac_N10_mem32K_40nm.xml and stratixiv_arch.timing.xml for VTR and Titan benchmarks respectively.

IX. Experimental Results

A. Overhead of intra/inter-node communication

ParaDiMe can be executed with different node configurations in the HPC. For example, ParaDiMe using two processes can be run with two processes on one node or two nodes with one process each. However, different node configurations lead to different communication overhead, which affects the speedup.

Figure 4 shows the average speedup of ParaDiMe across the benchmarks in Table I versus different node configurations.

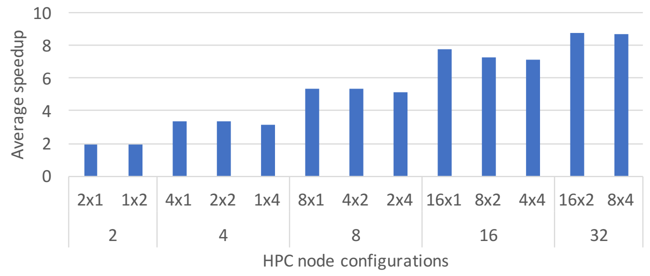

Fig. 4: Average speedup of ParaDiMe versus different node configurations

TABLE II: Execution time (in seconds) of VTR and single process ParaDiMe using different channel widths

	VTR			ParaDiMe		
Benchmarks	20%	30%	40%	20%	30%	40%
stereovision1	12	11	11	10	10	10
LU8PEEng	54	51	58	48	45	52
stereovision2	94	99	97	80	77	66
LU32PEEng	392	466	462	376	416	471
neuron	718	686	853	306	369	337
stereo_vision	414	476	462	199	198	201
segmentation	2130	3685	1928	1055	968	898
denoise	5361	3874	3728	2200	1785	1621

The speedup is measured by running ParaDiMe with 20% higher than the minimum channel width listed in Table I and without repartitioning in the second iteration. The first number in each node configuration is the number of nodes while the second number is the number of processes per node. For example, 2x1 means two nodes with one process in each node. Due to the memory requirements of larger benchmarks, the maximum number of processes per node is limited to four. Intuitively, it is expected that increasing the number of nodes decreases the speedup because inter-node communication has a higher overhead than intra-node communication. However, the results in Figure 4 shows the opposite. This is because inter-node communication in MPI utilizes the remote direct memory access (RDMA) capability of Infiniband adapters, which allows communication to be done in parallel with routing. On the other hand, intra-node communication in MPI requires processors to be actively involved in the transfer of messages via a shared memory kernel module. Therefore, the processors are not free to route nets while intra-node communication is happening. As a result, increasing the number of nodes actually allows more MPI communications to be offloaded to the Infiniband adapter, and thus increases the speedup.

In order to fully utilize the RDMA capabilities of the Infiniband adapter, the results in subsequent sections are obtained by using one process per node for all number of processes (1, 2, 4, 8, 16) except for 32 processes where two processes per node are used due to the limited number of nodes (16) in the HPC cluster.

B. Speedup across different benchmarks

Table II shows the execution time of VTR and single process ParaDiMe while Figure 5 shows the speedup of VTR and ParaDiMe relative to single process ParaDiMe across different benchmarks. In order to evaluate the performance of ParaDiMe under a high-stress condition where routing resources are limited, the values in the figure are obtained by running ParaDiMe with only 20% higher than the minimum channel width listed in Table I instead of 30% and 40% in existing works. Moreover, repartitioning in the second iteration (Section IV) is disabled because ParaDiMe actually performs better without repartitioning as explained in Section IX-E.

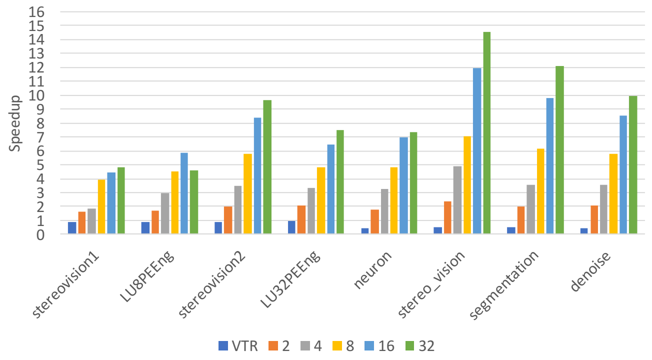

Fig. 5: Speedup of ParaDiMe across different benchmarks

Since benchmarks in Figure 5 are sorted in increasing order of the number of nets, it can be seen that smaller benchmarks such as stereovision1 and LU8PEEng have the lowest speedup among all benchmarks. In addition, their speedups do not scale well with the number of processes. In fact, the speedup of LU8PEEng with 32 processes is lower than that with 16 processes. Larger benchmarks do not suffer from this problem because the overhead of MPI communication is amortized over the larger amount of time spent routing.

For Titan benchmarks, ParaDiMe is around 2X faster than VTR. We observed that VTR spends a significant amount of time resolving congestion at the output pin when routing Titan benchmarks. This is because the architecture used to map Titan benchmarks allow multi-fanout nets to use multiple output pins to reach their sinks. This flexibility increases the congestion at the output pins, which are limited in numbers. In order to solve the problem, ParaDiMe has an enhancement that is similar to [5] where multi-fanout nets are restricted to using only one output pin to reach their sinks. Therefore, ParaDiMe is faster than VTR in routing Titan benchmarks. Another reason for ParaDiMe's significant speedup over VTR is that Titan benchmarks are harder to route and ParaDiMe transitions to rerouting only congested nets earlier while VTR continues to reroute every net. Although the number of processes is recursively reduced by half when rerouting only congested nets, the reduction in workload outweighs the decrease in processing power.

C. Critical path delay

Since ParaDiMe routes multiple nets in parallel, the nets might contend for the same routing resources and affect the critical path delay if they are on the critical path. Therefore, using VTR as a baseline for comparison and the same ParaDiMe configuration as the previous section (20% and no repartitioning), we evaluate the effect of the number of processes on the critical path delay.

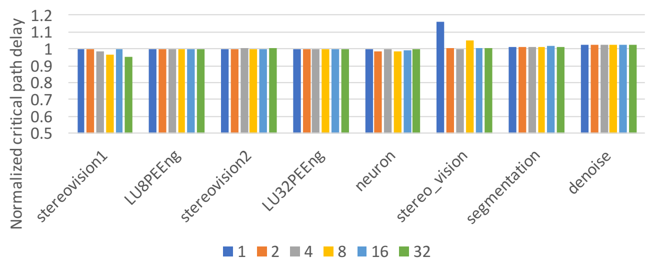

Fig. 6: Normalized critical path delay of ParaDiMe for different benchmarks

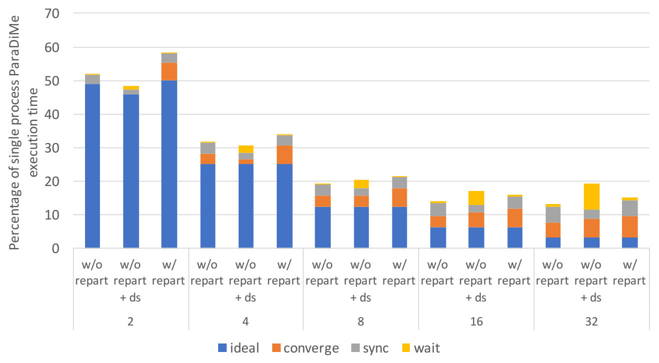

Fig. 7: Execution time profile of ParaDiMe

As shown in Figure 6, the quality of result of ParaDiMe is relatively independent of the number of processes. In addition, ParaDiMe produces critical path delays that closely matches that of VTR's except for stereo_vision with one process.

D. Bottlenecks

In order to determine the bottleneck that limits the speedup of ParaDiMe, different aspects of the execution time are measured and shown in Figure 7. $ideal$ is the percentage of single process time that ParaDiMe spends routing nets with N number of processes assuming perfect parallelization is possible, and it is calculated by $100\%/N$. For example, the $ideal$ percentage for two-process ParaDime is 50%. In practice, it is difficult to achieve the $ideal$ percentage because of the overheads incurred by parallelization. These overheads are represented by the $converge$, $sync$ and $wait$ percentages.

Despite the measures taken in ParaDiMe to ensure that the congestion costs remain as updated as possible, there is a possibility for a process to read outdated congestion costs when routing multiple nets in parallel due to the delay in receiving messages from other processes. This causes ParaDiMe to spend extra time in converging to a congestion-free state. The extra time is represented as the $converge$ percentage in Figure 7.

$sync$ is the percentage of single process time that ParaDiMe spends synchronizing congestion costs. In other words, $sync$ measures the communication overhead in ParaDiMe.

Lastly, $wait$ is the percentage of single process time that ParaDiMe spends idling due to the load imbalance caused by sub-optimal partitioning.

In this section, we explain only the bottlenecks for ParaDiMe without repartitioning. The observations for the

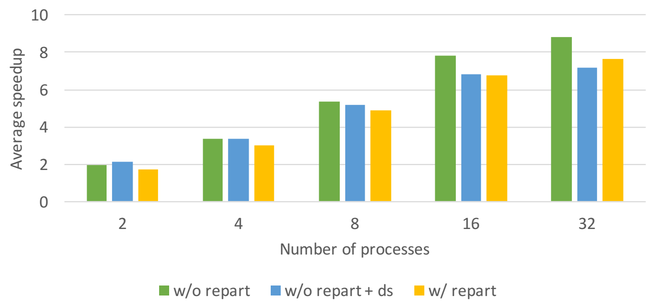

Fig. 8: Average speedup comparison of different versions of ParaDiMe

other two version of ParaDiMe are described in subsequent sub-sections. The advantage of encoding the path to sink in ParaDiMe is obvious from Figure 7 where the $sync$ percentage does not increase significantly even when the number of processes is increased to 32. In addition, the simple approach of load balancing by partitioning the nets in decreasing order of the number of sinks works relatively well as shown by the low $wait$ percentages. Unfortunately, the $converge$ percentage increases as the number of processes increases.

An interesting observation is that the $ideal$ percentage is less than the theoretical minimum (50%) when ParaDiMe is run with two processes. This is because of the reduction in workload when ParaDiMe only reroutes congested nets after detecting that the amount of congestion stop reducing monotonically.

E. Repartitioning

Intuitively, repartitioning at the second iteration should yield better speedup because it is based on the actual route time of nets from the first iteration instead of an approximation with the number of sinks. However, the results in Figure 8 show otherwise where ParaDiMe with repartitioning consistently performs worse than ParaDiMe without repartitioning. This is because sorting the nets in decreasing order of route time during repartitioning causes some high fanout nets with low route time to be routed later in the iteration. Routing high fanout nets later generally increases the effort required to resolve congestion, which is why VTR routes high fanout nets first. As a result, ParaDiMe with repartitioning spends more time achieving convergence than ParaDiMe without repartitioning. This is in line with the results in Figure 7 where the $converge$ percentage for ParaDiMe with repartitioning is higher than without. Moreover, the load balancing of ParaDiMe is already relatively good even without repartitioning as shown by the very low $wait$ percentage in Figure 7.

In addition, we observed that in certain benchmarks such as LU32PEEng, the route time of some nets changes significantly after the first iteration. Therefore, repartitioning based on the route time from the first iteration is ineffective in some cases. It is possible to repartition more than once but the overhead of doing so needs to be taken into account. We will explore the tradeoffs between better load balancing and higher repartitioning overhead in future works.

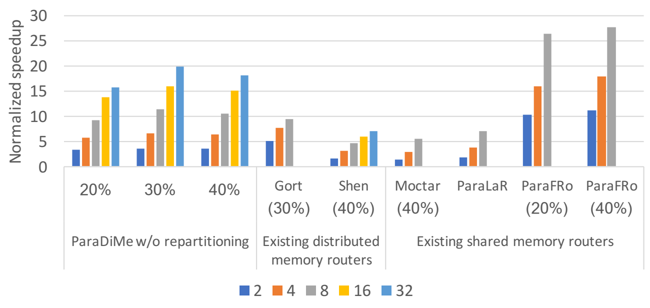

Fig. 9: Speedup comparison of ParaDiMe with existing works

F. Delayed synchronization

ParaDiMe synchronizes congestion costs more frequently than existing works [5], [6], [9] as explained in Section V. In this section, we show the advantage of doing so by comparing with a delayed synchronization (DS) version of ParaDiMe that only synchronizes congestion costs after routing every net.

From Figure 8, ParaDiMe DS only performs better than ParaDiMe when run with two processes. The first reason is the reduction in workload due to the earlier switch to rerouting only congested nets as described previously. The reduction in $route$ time shown in Figure 7 for ParaDiMe DS versus ParaDiMe confirms this reasoning. The second reason is that the delayed synchronization reduces the number of MPI calls during routing, which reduces the amount of communication overhead as shown in Figure 7. Moreover, the delayed synchronization does not significantly increase the staleness of the first order congestion costs, which is known to increase the $converge$ percentage, because there are only two processes.

On the other hand, ParaDiMe performs better than ParaDiME DS when run with eight or more processes. The MVAPICH implementation of MPI has different protocols for transferring messages of different sizes. The first protocol is called eager, and it is used to send small messages below a predefined threshold. The eager protocol is a one-way protocol where the sender simply assumes that the receiver has sufficient buffer space to store the incoming message. The second protocol is a two-way protocol called rendezvous that is used for messages that are larger than the threshold. In rendezvous, the large message is broken down into smaller chunks and the receiver has to acknowledge the receipt of a chunk before the sender continues sending the remaining chunks. It is obvious that the eager protocol has a lower overhead than the rendezvous protocol. Since ParaDiMe DS only synchronizes costs after routing a net, the message size generally exceeds the eager threshold. Therefore, the overhead of sending such messages is higher than ParaDiMe, which sends frequent but smaller messages that are below the eager threshold. The higher overhead of ParaDiMe is reflected in the $wait$ percentage instead of the $sync$ percentage in Figure 7 because the send requests are queued during routing but completed only after all the nets are routed.

G. Comparison with existing works

Figure 9 shows the speedup of ParaDiMe compared to existing works. The values are normalized to baseline VTR so that a fair comparison can be made (Speedup values in previous sections are self-relative). In addition, ParaDiMe was also executed with 30% and 40% higher than minimum channel width to allow for easier comparison with existing works. ParaDiMe achieves an average speedup of 3.62X, 6.68X, 11.4X, 16X, 19.8X with 2, 4, 8, 16, 32 processes respectively and 30% higher than the minimum channel width. From Figure 9, ParaDiMe significantly outperforms the distributed memory router by Shen and Luo [6]. As compared to the distributed memory router by Gort and Anderson [5], ParaDiMe demonstrates better scalability from 8 processes onwards. The reason is because the amount of parallelism in [6] and [5] is limited by the availability of intra-partition nets.

Compared to existing shared memory routers, ParaDiMe is faster than the router by Moctar and Brisk [7] and ParaLaR [10] even though ParaDiMe has higher synchronization overhead due to the use of distributed memory systems. However, ParaDiMe is slower than ParaFRo [11], which uses fine-grained synchronization with very low overhead.

X. Conclusion

In conclusion, we have proposed a distributed memory FPGA router based on speculative parallelism and path encoding. ParaDiMe scales well with the number of processes and achieves a speedup of 19.8X with 32 processes. In addition, the high speedup is achieved without sacrificing quality of result where ParaDiMe produces similar critical path delays as VTR.

Acknowledgment

This work is supported in part by the German Research Foundation (DFG) within the Cluster of Excellence “Center for Advancing Electronics Dresden” (cfaed) at the Technische Universität Dresden.

References

[1] “Introducing innovations at 28 nm to move beyond moores laws,” Altera, Tech. Rep. WP-01125-1.2, 2012.
[2] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, “The tao of parallelism in algorithms,” *ACM Sigplan Notices*, vol. 46, no. 6, pp. 12–25, 2011.
[3] L. McMurchie and C. Ebeling, “PathFinder: a negotiation-based performance-driven router for FPGAs,” in *ACM/SIGDA FPGA*, 1995.
[4] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, N. Ahmed *et al.*, “VTR 7.0: Next generation architecture and CAD system for FPGAs,” *TRETS*, vol. 7, no. 2, p. 6, 2014.
[5] M. Gort and J. H. Anderson, “Accelerating FPGA Routing Through Parallelization and Engineering Enhancements, Special Section on PAR-CAD 2010,” *IEEE TCAD*, vol. 31, no. 1, pp. 61–74, 2012.
[6] M. Shen and G. Luo, “Accelerate fpga routing with parallel recursive partitioning,” in *ICCAD*. IEEE, 2015, pp. 118–125.
[7] Y. O. M. Moctar and P. Brisk, “Parallel FPGA Routing based on the Operator Formulation,” in *DAC*. ACM, 2014, pp. 1–6.
[8] L. A. Cabral, J. S. Aude, and N. Maculan, “TDR: A distributed-memory parallel routing algorithm for FPGAs,” in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 2002, pp. 263–270.
[9] P. K. Chan, M. D. Schlag, C. Ebeling, and L. McMurchie, “Distributed-memory parallel routing for field-programmable gate arrays,” *IEEE TCAD*, vol. 19, no. 8, pp. 850–862, 2000.
[10] C. H. Hoo, A. Kumar, and Y. Ha, “Paralar: A parallel fpga router based on lagrangian relaxation,” in *FPL*. IEEE, 2015, pp. 1–6.
[11] C. H. Hoo, Y. Ha, and A. Kumar, “Parafro: A hybrid parallel fpga router using fine grained synchronization and partitioning,” in *FPL*. IEEE, 2016, pp. 1–11.
[12] K. E. 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,” *TRETS*, vol. 8, no. 2, p. 10, 2015.