PARALLELIZING PATTERN ROUTING ALGORITHMS FOR CHIP DESIGN

by

Aarti Pravinkumar Kothari, M.S.

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
May 2020

Committee Members:

Martin Burtscher, Chair

Apan Qasem

Mina Guirguis

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

I dedicate this dissertation to my husband, Adesh Namekumar, for his extravagant support and constant encouragement throughout this entire journey. His confidence in my ability to get this done is what brought me to this stage. I will always be appreciative of everything that he has taught me.

I also dedicate this achievement to my parents who have supported me throughout this process. I will always be grateful for everything that they have done for me.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor, Professor Martin Burtscher, for giving me the opportunity to do this research while providing invaluable guidance and molding me thoroughly. His encouragement and sincerity have deeply inspired me, and I consider it my privilege to have worked under him.

I would also like to thank the distinguished members of the panel headed by Professor Martin Burtscher, as well as with Guirguis Mina S and Qasem Apan M for the taking the time to review and approve my work.

I would also like to thank other members of my research team for their valuable feedback on my work. A special mention to Michael He for all his pointers and answers during the evaluation phase of my work.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

In VLSI design, minimizing wirelength is critical to maximize chip performance and minimize power consumption. A router determines the paths taken by the wires in a chip while satisfying design manufacturing rules. For modern circuit designs, a chip may contain billions of devices connected by millions of such nets. Due to the complex constraints and dependencies that need to be considered during routing, serial routing algorithms are ruling the industry today. In this thesis, I developed parallel routing algorithms for execution on multicore CPUs and GPUs. I created two routing algorithms such that they can simultaneously route pin pairs and update edge capacities while keeping wirelengths short and congestion low. Compared to a preexisting serial router routing netlists with between 0.2 and 2.6 million nets, my serial implementation is over 6x faster and requires 0.38% less wirelength. My deterministic parallel implementation produces the same result and is 2.5x faster than my serial code.

1. OVERVIEW OF CHIP DESIGN

Chip design is the subset of semiconductor engineering that encompasses the logic and circuit design techniques required to design Integrated Circuits (ICs). Chip manufacturing is a very complex process involving a series of steps that take a long period of time since each of these steps is a process by itself.

Steps in Chip Design

Figure 1 shows the steps involved in chip manufacturing while also detailing the sub-steps of Physical Design, which includes Signal Routing - the focus of this study.

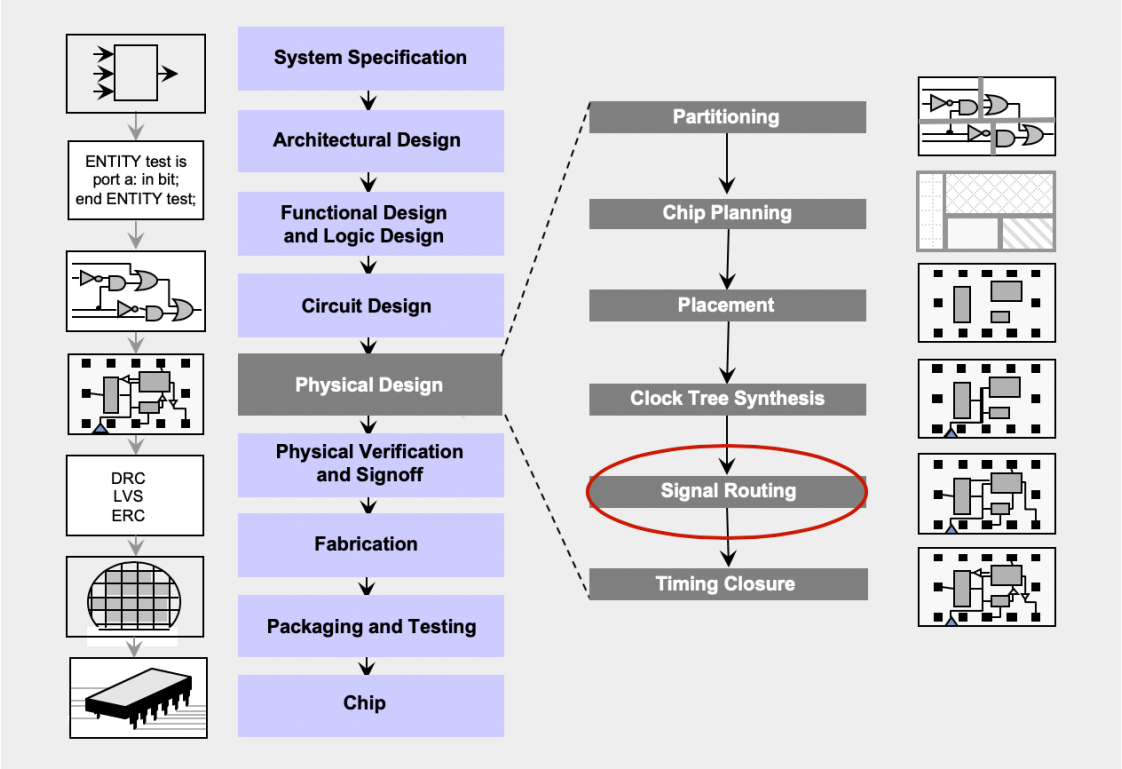

Fig. 1. Steps involved in chip manufacturing [1]

In physical design, the layout of the entire chip is divided in terms of routing regions. In Figure 2, the gate cells a, b, c, d, e, f, g and h are the logic gates that are placed inside their respective blocks during the placement step. Each gate cell has pins,

also called terminals, through which they connect logically. These terminals are connected by wires or nets that are basically connecting routes between two or more pins on a chip. This is where routing comes into the picture.

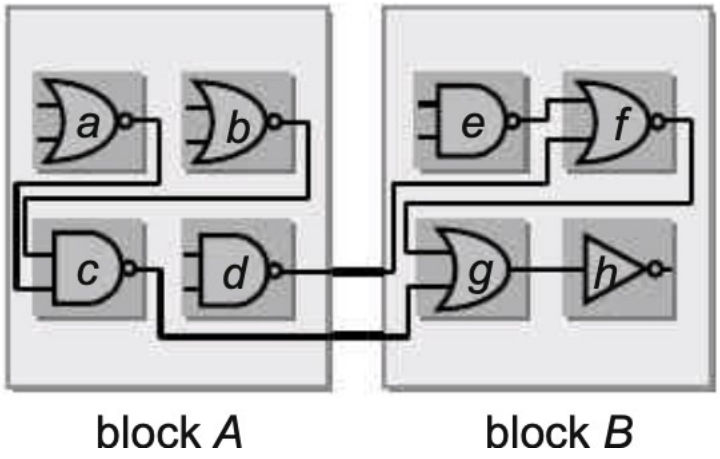

Fig. 2. Logic gates placed inside blocks having input and output terminals [1]

Challenges in Chip Design

For every tool that can perform routing, and in every industry where chips are used for the various electronic products manufactured today, the challenges faced today include:

- improving performance of the chip
- reducing power
- reducing area
- minimizing turnaround time to market

In all these challenges, routing plays an important role. Wirelength greatly affects the speed and area of a chip. A good router must produce routes having minimal wirelength and minimal congestion to improve overall chip performance and reduce power

consumption. Chips with a lower wirelength will also require less area. A fast router can affect the turnaround time for large modern-day circuit designs.

2. BASICS OF ROUTING

Routing is the process of connecting components of an IC constrained by prescribed manufacturing design rules. The primary task of the router is to create geometries for the nets such that there are no open or short circuits or any design rule violations. These geometries consist of routing tracks, which are basically metal layers that run as wires that can only be routed through tracks that are spaced out by a certain pitch that is specified by lithographic design rules.

Each net may connect several pins based on their coordinate positions. In modern circuit designs, there may be many devices that connect to the same net and there are millions of such nets that need to be routed.

In Figure 3, the output pin of Gate Cell1 is connected to the input pins of Gate Cell 2 and Gate Cell 3 by the wire “Net”. The pins are shown in red. The wires connecting these pins in the routing step are shown in green.

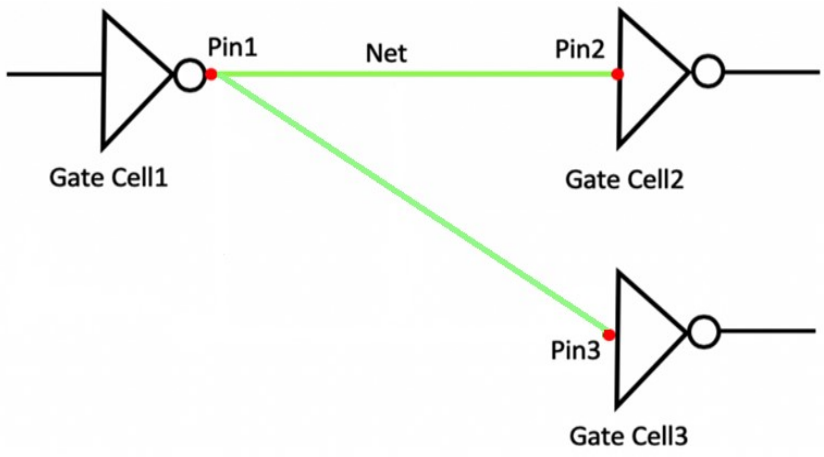

Fig. 3. Basic example of routes connecting pins of three gates

3. STEPS IN SIGNAL ROUTING

At the top-level, routing is a three-step approach. This is shown in Figure 4.

- Rectilinear Steiner Minimum Tree (RSMT) generation
    - Computation of Steiner points to minimize wire length
- Global routing
- Detailed routing

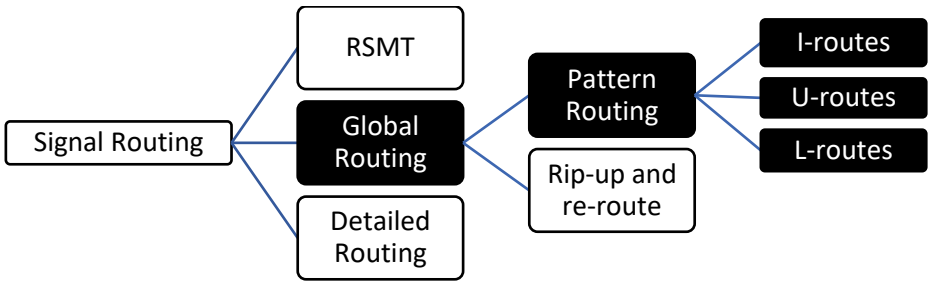

Fig. 4. Steps in Signal Routing

Each of these steps is explained in the following sections. The steps highlighted under Global Routing in Figure 4 are the focus of my thesis.

Rectilinear Routing

Each of the metal layers used in routing can have a different pitch and spacing but in general, these tracks can only be rectilinear in nature. This means that routing tracks can only run horizontally or vertically for each metal layer due to lithographic constraints. To interconnect *n* pins in a plane using only horizontal and vertical wires requires the use of rectilinear distances instead of Euclidean distance.

Suppose I need to connect two points as shown in Figure 5. The simplest way to connect them is diagonally, which is also the shortest distance. But using rectilinear distances, I would have to choose one of the red, blue or yellow routes.

Due to various paths available to connect pin pairs, there needs to be a mechanism

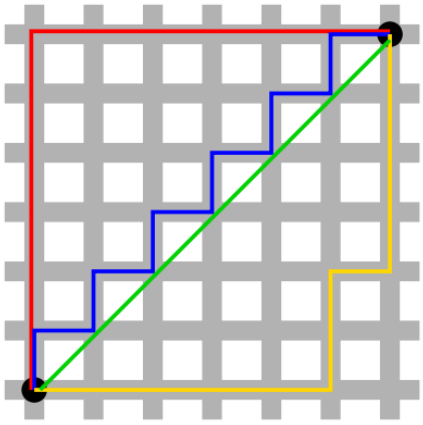

Fig. 5. Rectilinear distance versus Euclidian distance [2]

to make the router take the path that minimizes wire lengths to some extent and possibly the number of turns. This might require the insertion of Steiner points to connect two pin pairs.

Steiner Points

Suppose I need to connect the 3 points a, b and c with the coordinates shown in Figure 6. Since, only rectilinear distances are allowed, the obvious route would be the

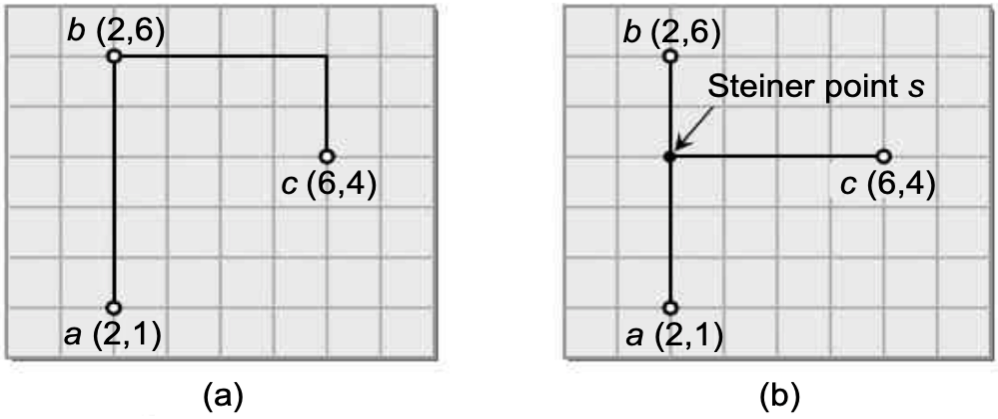

Fig. 6. Steiner point [1]

one shown in Panel (a) where the wirelength comes out to be 12 units.

An alternative route would be the one shown in Panel (b), which is shorter as it is only 10 units in length. This connection point, which gives an optimal wirelength, is called a Steiner point. This minimized wirelength is what ultimately makes a chip faster, have lower capacitance and make it much more energy efficient. In addition, it also uses lesser overall capacity, so there is more real estate for running other wires.

Applying this concept to the net connecting the three gate cells in Figure 3, the router can minimize the wire length by adding a Steiner point S1 assuming that the vertical wire in the path from S1 to Pin3 is a routing track.

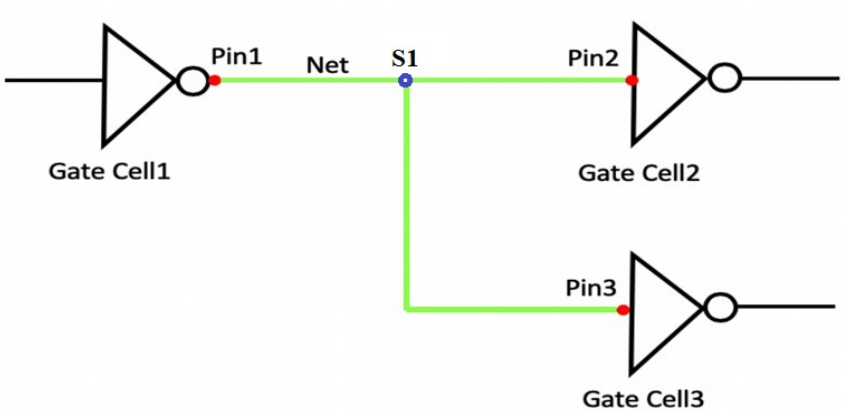

Fig. 7. Gate cells having input and output pins connected through nets

Routing Goals

This along with the theory behind routing clarifies the two important goals of any router:

- Correctness of circuit

The primary task of the router is to connect the pins of the nets such that there are no open or short circuits or any design rule violations.

- Performance of circuit

The secondary task is to interconnect the pins using only horizontal and vertical wires such that the total wirelength is minimized.

Global Routing

Global routing is where the chip is divided into tiles, earlier referred to as blocks, through which coarse grain routes are run. The configuration determined by global routing remains fixed during the next step, so any poor solution will affect the quality. My focus is on global routing, where the entire chip is divided into regions that consist of tentative routes. The actual layout of the wires is only decided in the detailed routing step.

Detailed Routing

In the detailed routing step, each net is assigned a specific routing track based on decisions such as net ordering and pin ordering, which can have a great impact on the final solution and its quality. This two-stage approach is required to handle the high complexity of routing.

4. TERMINOLOGIES USED

The area over which the components of a chip are laid out is referred to as the layout grid, which is divided into tiles. Each tile in the layout grid shares its boundary or edge with the neighboring blocks. These shared edges are referred to as borders. These are shown in Figure 8.

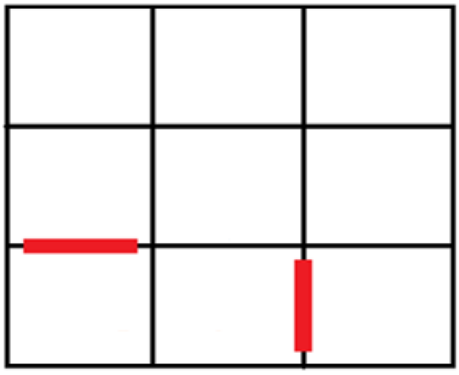

Fig. 8. Layout grid divided into tiles with blockages shown in red

Border capacities, determined by the number of routing tracks, can be used to determine the available wire capacity and the extent of congestion at an edge. These capacities are referred to as the overall horizontal and vertical routing capacities in the context of this thesis. Each of these borders might have blockages, which decrease the routing capacity from one tile to another.

5. INPUTS TO PATTERN ROUTING

Now that we have a background on what routing is and the terminologies involved, it is important to understand the main parts of the input to the pattern routing step. Each input netlist consists of information about the capacity of the different metal layers, number of signal nets, and number of pins, their positions within each net and additional congestion information for each of the metal layers.

Each net connects several pins whose coordinate positions help determine the kind of routes a wire can take. Each connection thus boils down to a pair of pin coordinates whose positions dictate the different path a wire can take to connect them. The capacity and congestion information help determine the overall horizontal and vertical capacity available during this routing process.

6. PROBLEMS DURING PARALLELIZATION

It is important in industries that use routing tools for chip design that the routing tool is fast while producing routes of optimal length, low congestion and high performance. This list of difficult objectives is what makes routing so hard and the reason why it is serialized in the modern chip design today.

In serial routing, each of the pin pairs are routed one after the other. Based on the available capacity at the borders of each tile, the router decides an optimal route from the different possibilities that are checked. Each of these routes can be optimized either for wirelength or for congestion or both. Since this is done serially, there is no contention when checking the capacities at the border that two tiles share. This interdependency is what makes parallelization difficult.

To understand the main issue during parallelization, let's assume an edge capacity of 1 in a 3x3 layout grid as shown in Figure 9. Let's also assume there are two pins that need to be routed to connect to two other pins in the neighboring tiles.

In the serial approach (left panel), the first pin pair that is routed will see an edge capacity of 1 taking the straight path to the neighboring tile. Since this edge no longer has any capacity left, the second pin pair can no longer route through the same edge. Instead, it will be routed through other adjacent tiles in a roundabout manner. Both these routes are shown in the figure by the solid line paths.

In the parallel approach, both pin pairs check for available edge capacity at the same time. This will make the router incorrectly take the (dotted) paths through the same edge because the two threads will simultaneously check, and both find that the edge has capacity left. These are shown in the right panel in Figure 9.

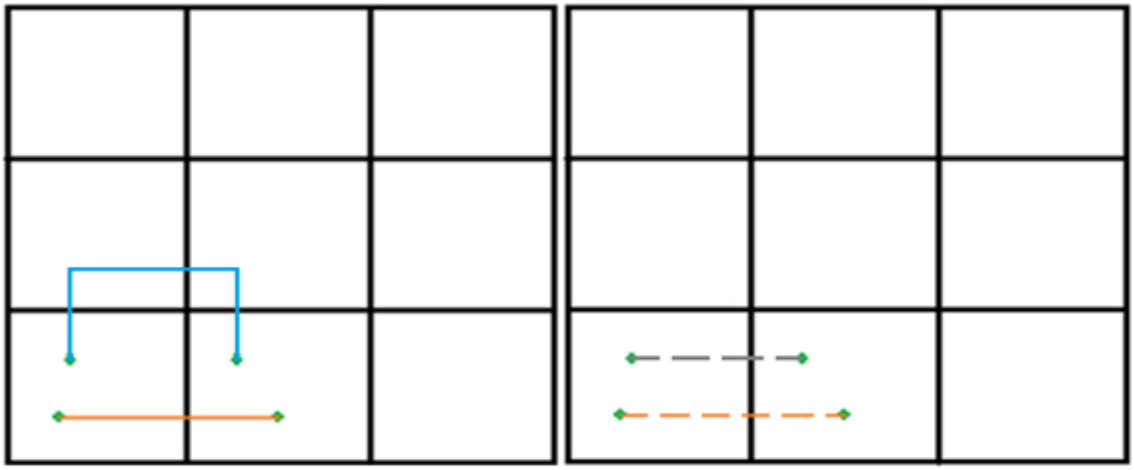

Fig. 9. Serial approach (left) and parallel approach (right) assuming an edge capacity = 1

7. ALGORITHM USED IN PATTERN ROUTING

In my thesis, I assume pin pairs have already been separated into two categories depending on the kind of route they can take based on their coordinate positions. When pin coordinates have a common axis, they are routed using I-routes (or U-routes when I-routes are not possible). If there are no common axis points, pin pairs are routed using L-routes. These are shown in Figure 10 and are named after the shape of the route connecting two pin pairs.

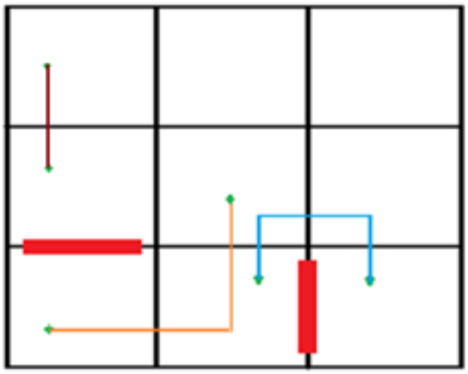

Fig. 10. Different types of routes in pattern routing

I-Routes

If a pair of pins have the same y-coordinates as shown in Figure 11, the optimal solution would be to connect them with a horizontal wire. Similarly, if a pair of pins have the same x-coordinates, then they can be connected using a vertical wire.

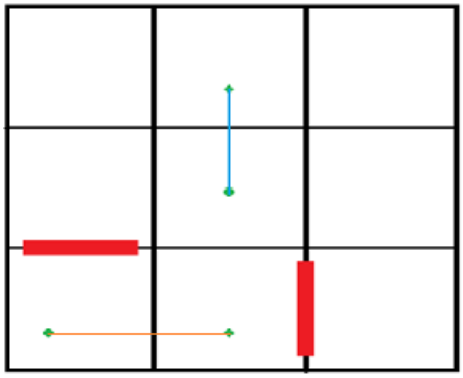

Fig. 11. Vertical and Horizontal I-routes

U-Routes

When there is not enough capacity to connect pins with a straight horizontal or vertical I-route, a different routing path called a U-route, as shown in Figure 12, is considered.

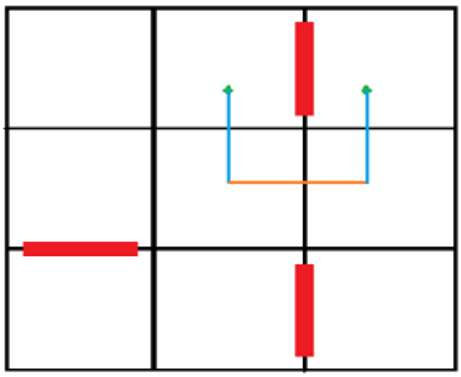

Fig. 12. Pin pair connected using U-route

These routes can differ in wirelengths depending on the length of the vertical routes in the “U” shape. Obviously, these are sub-optimal and hence are the second choice for these kind of pin pairs. If the I-route or the U-route path cannot be taken by a wire, then the router reports this pin pair as unrouted at the end of run.

L-Routes

When the co-ordinates of the pin pairs do not have any common point, they can be connected by a number of routing patterns depending on their position or depending on the available capacity. For the purpose of this research, all such pin pairs are connected by L-shaped wires that are routed through tiles that have enough capacity on the borders that it goes through. Each L-shaped route between a pin pair can have two possible paths that it can take. They are

- Horizontal-L, shown by dotted lines in Figure 13
- Vertical-L, shown using solid lines in Figure 13

If the first segment of the L-route is a vertical route and is followed by a horizontal route, it is called a vertical-L route. If the first segment is a horizontal route and is followed by a vertical route, then it is called the horizontal L-route.

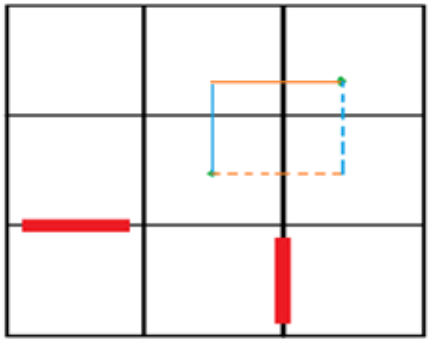

Fig. 13. Pin pairs showing horizontal and vertical L-route paths

The router considers both these paths before rendering a pin pair unrouteable due to lack of capacity.

8. PATTERN ROUTING APPROACH

Based on coordinate positions of pins after the RSMT step, I observed that a high percentage of wires required to be connected by I-routes. Since these have the least leeway, all I-routes are first routed based on the capacity at the borders of each tile on the route. The routes that do not have enough capacity for I-routing are then passed to the U-routing function with the updated available capacity. From the different possibilities, the router will choose the route that is minimal in wirelength. Finally, the other pin pairs are passed through the L-routing function since these are the most flexible way of connecting the remaining pin pairs. If a pin pair cannot take any of these routes, it is returned as unrouteable.

Among these three routing algorithms, I parallelized the I-routing function since a large percentage of the pin pairs require I-routes. The U-routing function, on the other hand, is not used as extensively and only handles the pin pairs left after I-routing. Since the number of pin pairs is low, there is not much need for parallelizing this function. The L-routing function handles a different category of pin pairs which are also a good percentage of the total pin pairs to be routed. Hence, I have attempted to parallelize this function as well.

Parallelizing I-Routing

Using CUDA, the main idea behind parallelizing I-routing on GPUs is to read available capacity in parallel while making sure there is no contention between the threads reading them, and then figure out pin pairs that can be routed in a mutually exclusive manner. The capacities are decremented in a serial manner per route by respective threads operating in a grid space.

There are three levels of parallelization in I-routing

- In the first level, pin pairs are bucketized depending on whether they are connected by horizontal or vertical I-routes
    - Horizonal and vertical I-routes are treated separately in parallel since their available capacities are independent
    - Border capacities are updated per route
- At the next level, each bucket is sub-divided into the grid tile row or column to which the I-routes belong
    - Each row/column can be treated in parallel since they are independent in terms of available capacities and pairs of pins to be routed
    - Border capacities need to be updated per route for every row/column
- In the final level, all pin pairs in the same row (or column) are routed in parallel by a warp of threads

Unrouted pin pairs are saved by threads in parallel using atomics to avoid race conditions.

Parallelizing L-Routing

This approach is referred to as the safe routing algorithm and consists of the following two phases:

- In the first phase, the router increments each border on both possible routes for each pair of pins to get a sense of the overall worst-case congestion on these edges.

- In the second phase, the pin pairs are actually routed based on the condition that a route exists between a pin pair such that the capacity on each edge of the route is at least as high as the incremented counts for that edge from the first phase.

Since this approach is based on the congestion information at each edge due to all the pins of all the nets, it ensures that none of the edges are ever routed over their capacity. Therefore, it is called “safe”.

After every route in the second phase is completed, the overall capacities of each horizontal and/or vertical edge is updated. When all the pins have gone through this process, the final capacity is now used by the next routing function.

For parallelization of this routing scheme, I used OpenMP to parallelize the code on the CPU with default block scheduling. To maximize the speedup achieved during parallelization as well as the number of routes successfully routed, the total number of pin pairs are divided into varying chunk sizes that are routed in successive iterations over multiple rounds. This allows having updated capacity information for the remaining pin pairs to be routed over every iteration and round.

To illustrate the benefit of this divide and conquer approach, let us assume an

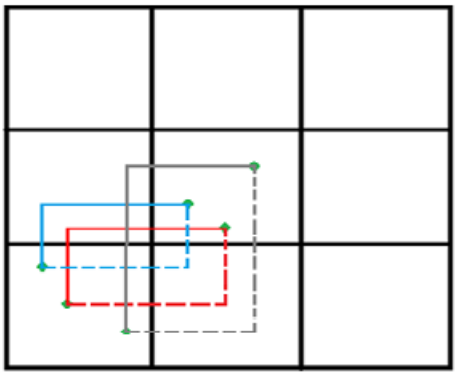

Fig. 14. Pin pairs showing possible L-route paths for three pin pairs

edge capacity of two for the routing grid shown in Figure 14. If the three pin pairs need to be routed using safe L-routes, each of them has two possibilities as shown in the figure.

Based on the above algorithm, the total capacity calculated in the first phase of safe routing is higher than the available edge capacities and, hence, none of these pin pairs can be successfully routed. However, if we route the pin pairs in reduced chunk sizes, we might be able to route a few or all of these pin pairs.

Let us assume a chunk size of 2 to see how the above case changes. The capacity of the first 2 routes in the first phase is now equal to the available edge capacity. This means that the first 2 routes can route any way they like using horizontal L-routes or vertical L-routes or both. One such possibility, where the first two pin pairs connect using vertical L-routes, is shown by the solid lines in Figure 15. This leaves the last pin pair with at least one open path that it can take through one of the other edges that still have some or all their capacity intact. This last route is shown by the path taken by the dotted lines in Figure 15.

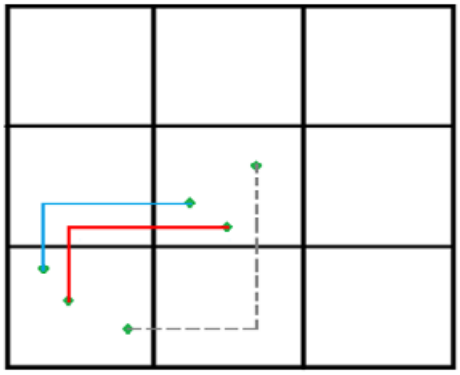

Fig. 15. With edge capacity = 2, pin pairs can be connected by L-routes when chunk size = 2

9. RELATED WORK

In the past years, the research on global routers has mainly been aimed towards improving routing quality by minimizing overflow and reducing run times. Running routing algorithms on multicore architectures and GPUs can help boost performance and perhaps even the quality in solving such problems. However, in global routing, the regions through which nets need to be routed overlap often, making it hard to break the layout grid into independent regions that can be routed in parallel. Another problem in parallelization is the differences in the load balance between the threads.

One parallel global routing algorithm, PGRIP [6], concurrently processes rectangular sub-regions of the chip area using an integer programming (IP) formulation both for routing each sub-problem and for connecting them. To achieve a no (or low) overflow global routing solution, a patching phase that uses IP to connect partial routing solutions is required, which provides feedback to each routing sub-problem when attempting to connect them. PGRIP then applies an IP-based procedure to solve the sub-problems in a systematic order.

A fundamental obstacle to this decomposition approach is the routing of nets that have terminals in multiple sub-problems. This was solved by GRIP [5] using a modified grid wherein these “inter-region” nets can be routed anywhere through a specified boundary of a sub-problem. After solving a sub-problem, fixed “pseudo-terminal” locations will be known on the sub-problem boundaries and must be honored by the neighboring subproblems that have not been solved yet. To obtain a complete solution, the subproblems are solved in a sequential order, which limits parallelism, while allowing independent sub-problems to be concurrently processed. PGRIP works similar to GRIP

but reduces the long execution times by ensuring that the resulting routings of the sub-problems can be effectively patched together (through one-time synchronization). The patching itself is also accomplished by IP. The result is an extended IP procedure of GRIP, with significant parallelism and a higher solution quality compared to other methods available at that time.

Load balancing is one of the issues seen in GRIP. NCTU-GR [7] is a net-level parallel multi-threaded global router that employs a task-based concurrency strategy (TCS) based on a proposed sequential global routing algorithm to achieve better load balancing, using a collision-aware rip-up and reroute solution by adjusting the cost of conflicting routing resources. This involves the decomposition of nets connecting pin pairs, building of skeleton edge sets for each net, generation of an initial congestion graph after touring all the nets that connect these pin pairs, and iteratively rerouting the overflowed nets until an overflow-free routing result is obtained.

In TCS, each pin pair to be routed is defined as a task. A task queue is maintained to dynamically allocate tasks to threads and minimize load imbalance. Threads are synchronized only at the end of each iteration. Compared to the partitioning-based strategy applied in PGRIP, this algorithm achieves a higher degree of concurrency by keeping all threads busy with almost perfect load balance.

NCTU-GR [7] also proposes a rectilinear Steiner minimum tree (RSMT) aware routing scheme to guide heuristic-BLMR and monotonic routing to build a routing tree with shorter wirelength. In addition, this paper presents two bounded-length maze routing (BLMR) algorithms (optimal-BLMR and heuristic-BLMR) that run much faster than traditional maze routing algorithms by limiting the search region and improving resource

utilization by reducing redundant wires. However, this step is related to the rip-up and reroute stage of global routing, which is not the focus of my thesis.

While the partitioning-based strategy in PGRIP prevents the simultaneous usage of the same routing resource by more than one thread in a region due to sub-region restriction, the TCS-based approach could possibly lead to a situation in which one thread may simultaneously compete with other threads for the same routing resource causing a race condition between routing resources, which may produce overflows. Note that my proposed approach is free from data races and guarantees not to produce any overflow.

Although PGRIP is parallel, the usage of integer programming makes the run time relatively long. As a remedy, VFGR [8] explores region-level and net-level parallelism to achieve much shorter run times using a proposed congestion model to handle multi-threaded global routing that can eventually also lead to a significant run time reduction during detailed routing.

Based on the observation that the resources consumed by a net are highly correlated with the size of the net, smaller nets were found to consume resources in a smaller region on lower routing layers while larger nets take resources in a larger scope on higher routing layer. Based on this approach, multiple levels of hierarchy are constructed on the routing region, in which each hierarchy level has different sizes of global routing cells into which nets are fitted. Global routing is carried out bottom-up and parallelism is used at a region-level or net-level. Because all regions are independent, region-level parallelism is performed to speed up the entire global routing step. After routing in each region, congested nets and detoured nets are deferred to higher routing levels. At higher levels that contain relatively small amount of global routing regions as

well as when the number of nets to be routed varies greatly over regions, load balancing becomes difficult. To handle this issue, they utilize net-level parallelization for higher level routing.

Though runtimes are significantly reduced, VFGR results in many overflows at edges indicating congestions after the detailed routing step, which would mean design-rule violations.

Unlike the parallel routers that work by partitioning the chip area into non-overlapping regions, SPRoute [10] proposes to also exploit net-level parallelism by using a two-phase maze routing approach when there is overflow after the pattern routing step.

SPRoute decomposes each multi-pin net into individual pin pairs to generate a rectilinear Steiner minimum tree (RSMT) using FLUTE [3]. In the next stage, pattern routing is used to generate the routing solution for the decomposed pin pairs. If there is overflow 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.

The two-phased maze routing approach initially exploits net-level parallelism, automatically lowers the parallelism when livelock is identified, and finally switches to fine-grain parallel processing of individual nets to guarantee convergence. For the net-level parallelization, each thread acquires a net and 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. To address this problem, threads are rolled back when resource

conflicts are detected but this can result in livelock. To address livelocks, a net-level-sequential fine-grain-parallel technique is used to achieve competitive speedups compared to methods that limit parallelism by enforcing disjointness of routing regions. This solution permits nets to be routed in parallel through the same region as long as enough routing resources are available. Hence, performance will be good for designs that are easier to route, whereas for more congested designs, this solution will only permit some level of parallelism. It should also be noted that SPRoute does not parallelize the pattern routing stage but only the following rip-up and reroute stage.

Finding enough independent parallelism to efficiently perform routing on GPUs is difficult. Existing task-based parallel routing algorithms must be completely revamped to make use of the GPU. Han et al. [9] implemented net-level parallelism in a hybrid GPU-CPU environment by identifying and scheduling independent nets to avoid race conditions on routing resources. However, the level of parallelism and solution quality highly depend on the input circuit.

Higher throughput is achieved using net-level concurrency (NLC) in a scheduler to create groups of nets that can be routed in parallel while dynamically analyzing data dependencies between multiple nets. Han's GPU-based global router uses a breadth first search (BFS) heuristic on the low-latency problems by finding the shortest weighted paths and routing multiple nets simultaneously while the CPU router concurrently uses A* maze routing on the high-latency problems. Unlike partitioning, NLC allows sharing of routing resources between threads, which means that threads must have current usage information of these resources. Also, to achieve high throughput from NLC, identifying congested regions and minimal resource collisions are important.

In my thesis, I implement both net-level and region-level parallelism to parallelize the pattern routing step in global routing and achieve shorter run times. My parallelization approach is new, guarantees that there will be no overflow, and is designed to exhibit large amounts of parallelism as needed for GPU execution. The various routing steps are coded such that there are no overlapping regions or resource conflicts that can limit parallelism. My serial algorithm is coded such that there are no dependencies that can stop us from parallelizing the code to run on GPUs. While the I-routing algorithm can never overflow by itself, the L-routing approach is safe enough to be able to guarantee that there are no overflows. In addition, I apply a RSMT based algorithm to achieve shorter wire lengths.

At the top level, the pin pairs to be routed are distributed among different algorithms so they can operate independently in their own space. During I-routing, the chip area is sub-divided into independent regions that are routed in parallel. Any resource utilization within a region is updated atomically by threads operating in that region. There is no collision within the data structure handling these resources since there are no overlapping regions. Further, routing at the net-level is performed by warps of threads to achieve higher speedup. This approach is scalable in the sense that the number of threads used in the GPU is automatically increased with the size of the layout grid.

In L-routing, pin pairs are broken down into chunks to improve the quality of the solution. Each of these chunks currently run on a multi-core CPU and are coded such that they can be parallelized to run on the GPU as well. Any pin pairs that are left unrouted can be handled separately and may require the rip-up and reroute of some pin pairs that have been successfully routed, as is the case with other pattern routers.

10. EVALUATION METHODOLOGY

I compared the results of the parallelized code with my serial routing algorithm and with a preexisting serial code, both in terms of quality of routing (wire length and number of pin pairs routed) and in terms of performance (run time). After determining all possible routes, my evaluation process involves comparison and correlation of trends in terms of maximum congestion observed, over-the-limit capacities at each of the edges, number of pin pairs routed in different rounds and iterations by each of the routing functions, total wire lengths for the successfully routed pin pairs, and the number of pin pairs left unrouted.

I compiled the OpenMP code with g++ version 7.3.1 using the flags "-O3 -march=native -fopenmp" and ran it on an AMD Ryzen Threadripper 2950X with 16 cores. From my experiments on the safe L-routing function using different number of threads and various scheduling mechanisms, I obtained optimal performance and load balance using 16 threads in parallel while running with the default block scheduling. Besides these, I also experimented running pin pairs in varying chunk sizes which are tuned in every successive iteration to increase the number of successful routes from the two-phased approach. The aim here was to try and find a trade-off between the total run time obtained over several iterations and the number of successful routes.

I compiled the CUDA code for the I-routing function with nvcc version 9.2.88 using the flags "-O3 -arch=sm_70" and ran it on a Nvidia TITAN V GPU with 5120 cores. The total number of threads used varied depending on the size of the layout grid for each of the inputs. Also, threads were always used in warps to process a group of nets divided based on the I-routing methodology.

Input Statistics

I used the netlists from the ISPD 2008 Global Routing Contest [4] as inputs where the number of nets range from 0.2 to 2.6 million. Pertinent information about these netlists is provided in Table 1.

Table 1. Input Netlist Information

Input file	Layout grid size	Number of pin pairs to be routed	Number of nets	Avg. pins per net	Maximum num of pins per net
adaptec1	324 x 324	622,155	219,794	3.29	2,271
adaptec2	424 x 424	657,130	260,159	3.09	1,935
adaptec3	774 x 779	1,206,084	466,295	3.02	3,713
adaptec4	774 x 779	1,180,460	515,304	2.71	3,974
adaptec5	465 x 468	1,821,027	867,441	3.03	9,863
bigblue1	227 x 227	687,362	282,974	3.04	2,621
bigblue2	468 x 471	1,160,243	576,816	2.68	11,869
bigblue3	555 x 557	1,666,377	1,122,340	2.41	7,623
bigblue4	403 x 405	3,250,738	2,228,903	2.99	20,766
newblue1	399 x 399	725,177	331,663	2.73	12,335
newblue2	557 x 463	1,054,500	463,213	2.83	8,089
newblue3	973 x 1256	1,025,685	551,667	2.50	16,008
newblue4	455 x 458	1,511,693	636,195	2.93	13,518
newblue5	637 x 640	3,009,997	1,257,555	2.92	17,913
newblue6	463 x 464	2,489,066	1,286,452	3.12	19,862
newblue7	488 x 490	4,539,308	2,635,625	2.83	17,324

Performance Evaluation

In the graphs below, the 16 different chips are listed along the X-axis. In Figure 16, the performance of my serial code is compared to the preexisting SPRoute [11] code in terms of run time in seconds along the Y-axis. The final set of bars along the X-axis is the geometric mean of all of them. The bars in solid black denote my serial code, referred to as ECLRoute moving forward. The patterned bars denote the serial code from SPRoute.

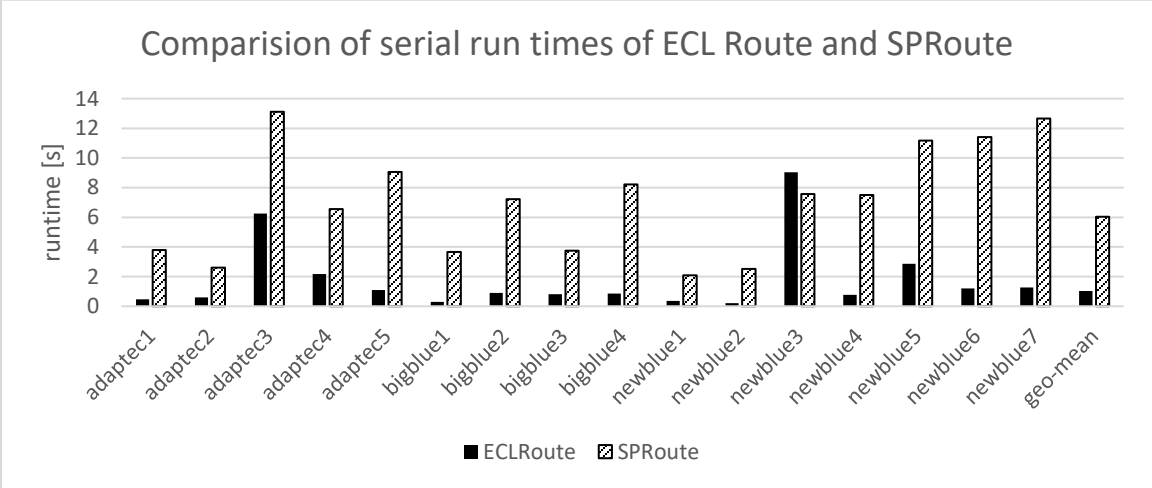

Fig. 16. Comparison of serial run times of ECLRoute and SPRoute

Clearly, my code is faster even while running serially except on one input. The speedup achieved by my serial code relative to the pattern routing step in SPRoute can be observed from the differences in the run times shown in the left half of Table 2 for each of the inputs. The row corresponding to the geo-mean calculated for all these run times shows that my serial code is over 6x faster with all these netlists considered together. The anomaly on newblue3, where the serial ECLRoute code is slower than SPRoute, may be

attributed to the relatively large layout grid combined with a lower than average total number of pin pairs, i.e., this input may be particularly easy to route.

Table 2. Runtime Comparison of ECLRoute with pattern routing step in SPRoute

	Serial Run times (s)		**I-Route Run times (s)**	
Input file	**ECLRoute**	**SPRoute**	**Serial ECLRoute**	**Parallel ECLRoute**
adaptec1	0.475	3.783	0.003	0.001
adaptec2	0.605	2.595	0.004	0.002
adaptec3	6.243	13.124	0.009	0.002
adaptec4	2.161	6.558	0.01	0.003
adaptec5	1.103	9.060	0.011	0.004
bigblue1	0.302	3.665	0.003	0.002
bigblue2	0.889	7.229	0.006	0.002
bigblue3	0.814	3.747	0.008	0.006
bigblue4	0.847	8.218	0.015	0.006
newblue1	0.359	2.095	0.003	0.002
newblue2	0.207	2.517	0.006	0.003
newblue3	9.039	7.566	0.006	0.003
newblue4	0.761	7.505	0.008	0.002
newblue5	2.859	11.178	0.018	0.005
newblue6	1.205	11.398	0.012	0.005
newblue7	1.271	12.657	0.02	0.011
Geo-mean	1.036	6.025	0.007	0.003

Moreover, my approach is parallelizable and runs even faster on a GPU. I parallelized the I-routing function to run on the GPU. The performance gain I obtained is evident from the right half of Table 2 where I compare the run times of the serial I-route function with the parallel version. The geo-mean in this case shows that the parallel code is about 2.4x faster on the GPU. The trends in the differences in run times are shown by the solid black bars in Figure 17.

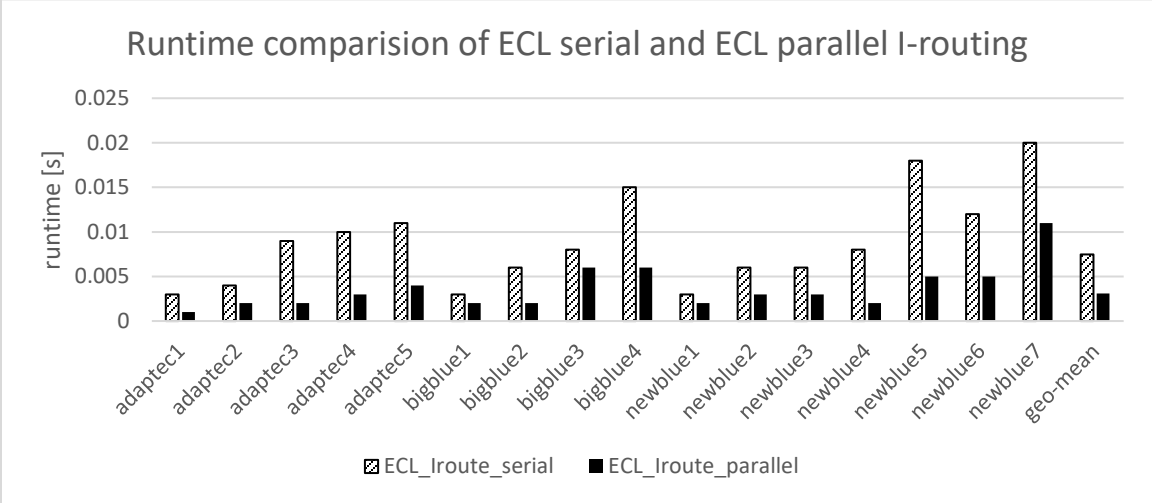

Fig. 17. Runtime comparison of ECL serial and ECL parallel I-routing

In the L-routing function, the total number of pin pairs to be routed are divided into chunks that vary in sizes depending on the number of pin pairs in every successive iteration. As the number of pin pairs to be routed decreases per iteration, the corresponding factor by which the chunk size needs to be reduced becomes a function of the total number of pin pairs that need to be considered. Looking at the trend in the run times in Figure 18, we can make two observations. Using a factor of 8 or 4 seems to make the serial run times worse when compared to using a factor of 2. This could be

because the higher reduction factors make the value of the chunk size reach its minimum faster. It is not obvious why a factor of 4 is worse than 2 and 8.

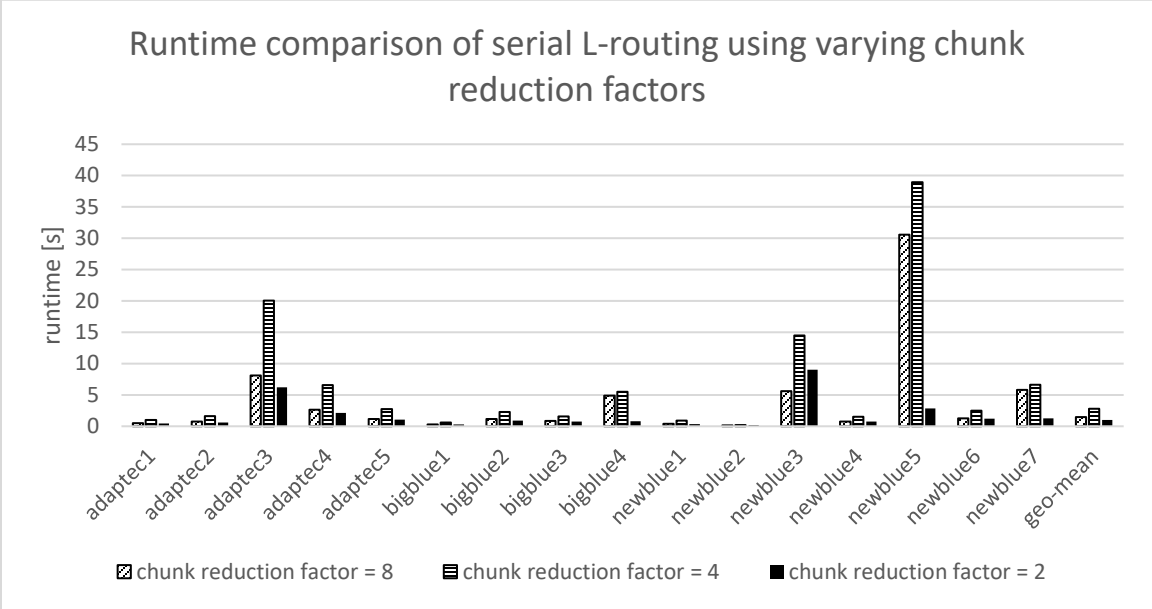

Fig. 18. Runtime comparison of serial L-routing function using varying chunk size reduction factors

Another variable that affected the number of pin pairs successfully routed during each of these iterations and the overall performance of L-routing was the initial value of the chunk size. The chart in Figure 19 shows the trend in serial run times while using differing initial values which are in turn, set as a percentage of the total number of pin pairs to be routed in an iteration. From this experiment, it was evident that setting the initial value of the chunk size equal to the total number of pin pairs helps the code perform best.

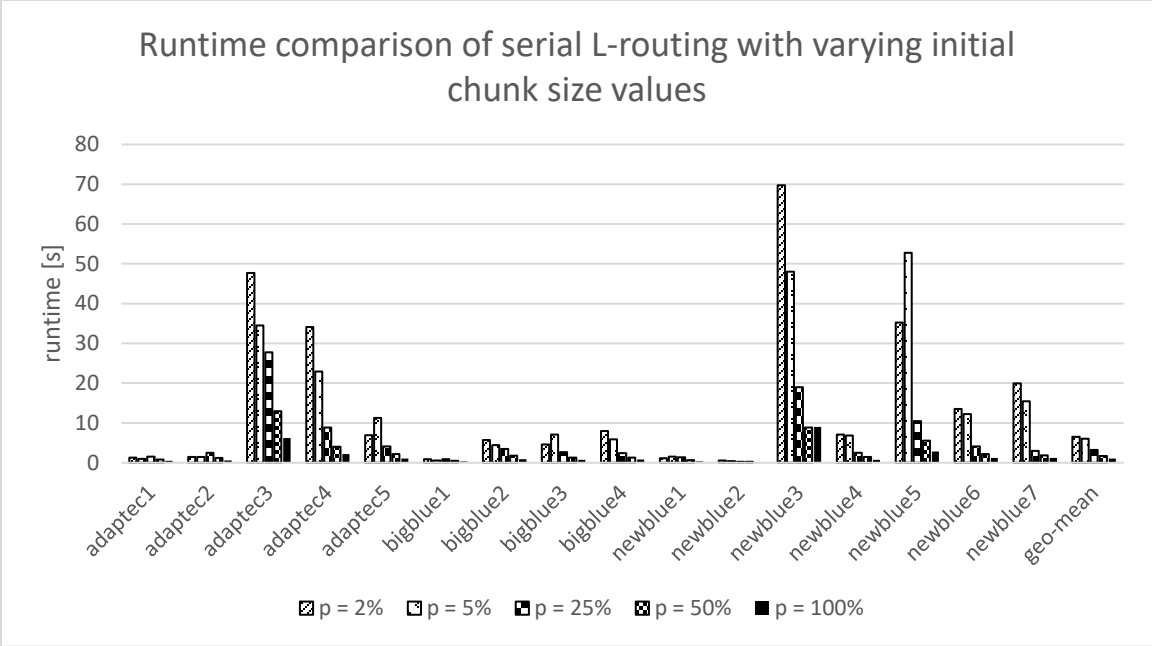

Fig. 19. Runtime comparison of serial L-routing function using initial chunk size values that vary as a percentage of total pin pairs

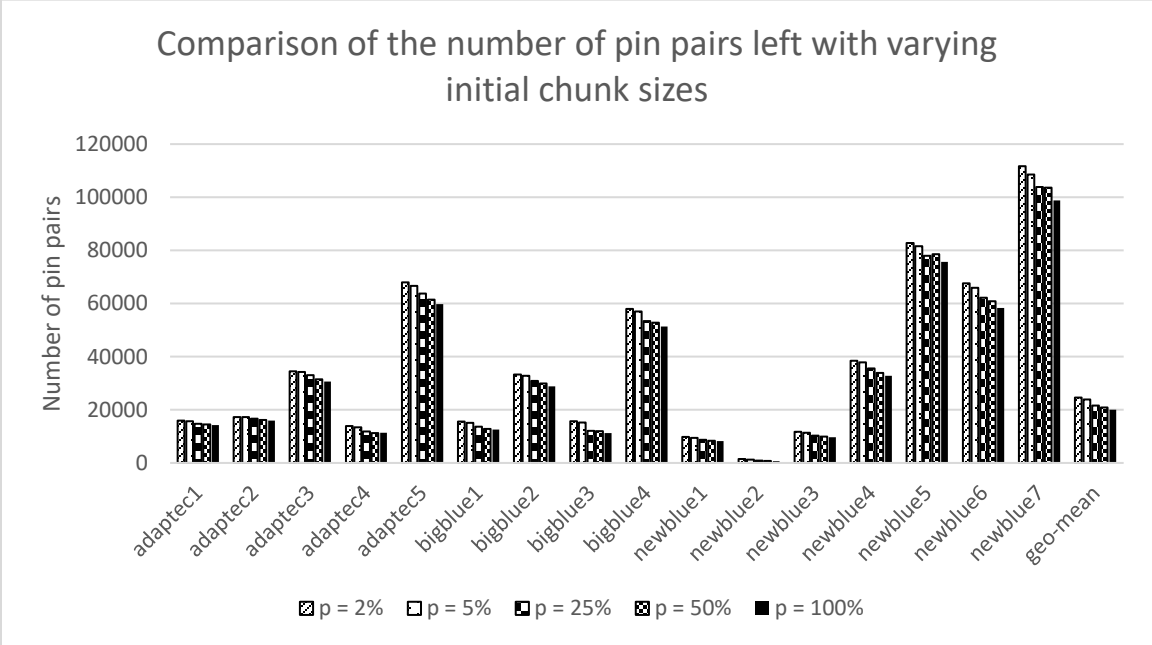

Fig. 20. Comparison of the number of pin pairs remaining after serial L-routing using varying initial chunk size values that vary as a percentage of total pin pairs

The chart in Figure 20 shows the trends in the number of pin pairs that are left

unrouted when the initial value of the chunk sizes was varied. Interestingly, this is lowest when the initial value is set to 100% of the total number of pin pairs being routed. This means that starting out with a lower initial value for the chunk sizes does not turn out to be the most optimal.

From the above experiments, it is seen that the L-routing function ran best with an initial chunk size equal to 100% of the pin pairs to be routed and a reduction factor of 2. Using this configuration, I parallelized the L-routing function just using OpenMP.

Figure 21 shows a plot comparing the run times of the serial code with the parallel code run using 16 threads. From these, we can see that all inputs have a higher run time using the parallel version. This can be attributed to the size of the workload for the L-routing function. Hence, the geo-mean in this case shows that the parallel version is much slower overall while running on a multi-core CPU. I believe this is due to slow atomic operations on the CPU and would be much faster on a GPU. Due to lack of time, I did not port this code to run on the GPU.

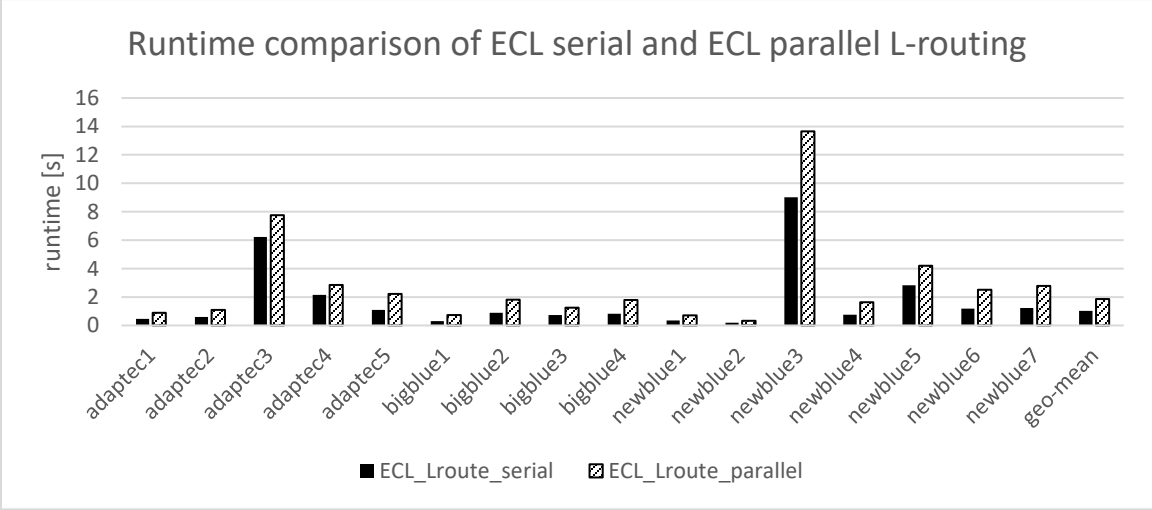

Fig. 21. Runtime comparison of ECL serial and ECL parallel L-routing

These are further detailed in Table 3 where we can see that the parallelized version takes almost twice the serial time considering the geo-mean. On the GPU, this

Table 3. Runtime comparison of serial and parallel L-routing function using 16 threads

	L-Route Run times (s)	
Input file	**Serial ECLRoute**	**Parallel ECLRoute**
adaptec1	0.469	0.88
adaptec2	0.598	1.102
adaptec3	6.221	7.753
adaptec4	2.141	2.842
adaptec5	1.083	2.215
bigblue1	0.298	0.736
bigblue2	0.88	1.806
bigblue3	0.742	1.24
bigblue4	0.828	1.801
newblue1	0.355	0.722
newblue2	0.2	0.324
newblue3	9.009	13.663
newblue4	0.751	1.632
newblue5	2.827	4.199
newblue6	1.189	2.521
newblue7	1.238	2.771
Geo-mean	1.017	1.865

could get way better.

Now that we have quantified the performance metrics for different sections of the code, let us also analyze a couple of quality metrics - number of pin pairs left after pattern routing (shows extent of routability) and total wirelength of all pin pairs to be routed.

In Figure 22, I compare the number of pins pairs that are left unrouted due to lack of capacity in ECL Route with the number of pin pairs that are ripped up and rerouted after the pattern routing step in SPRoute [11], in terms of percentage of the total pin pairs. Since my serial and parallel code is deterministic, this holds true for the parallel code as well. Again, the final set of bars along the X-axis is the geometric mean.

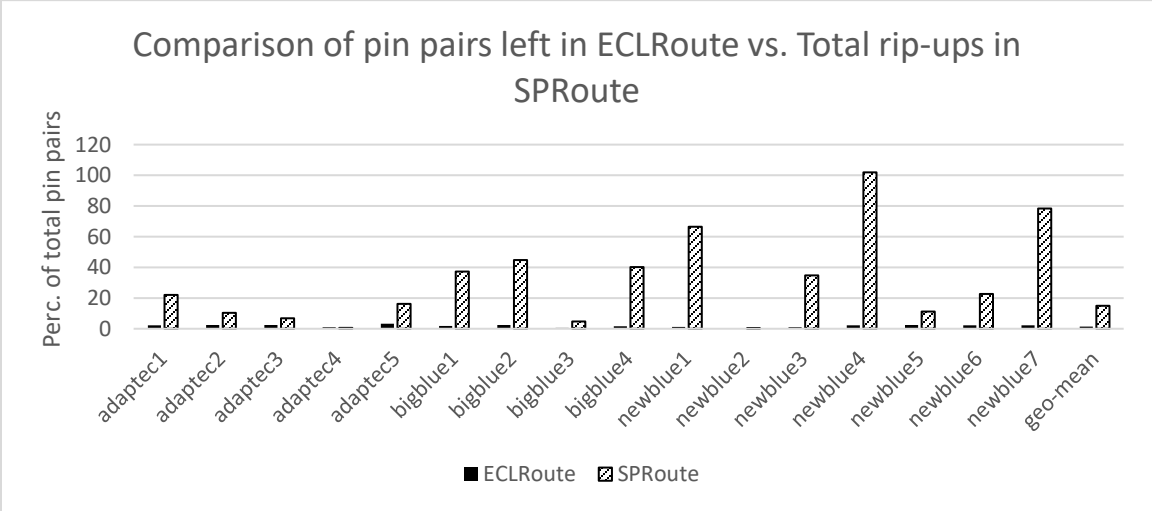

Fig. 22. Comparison of pin pairs left unrouted from ECLRoute with the total rip-ups in SPRoute in terms of percentage of the total number of pin pairs

The small solid black bars mean that the ECLRoute leaves very few unrouted pin pairs. In the comparison between ECL Route and SPRoute in Table 4, the left half of the table shows that my code routes a greater number of pin pairs using the same amount of capacities leaving behind over 10x fewer pin pairs unrouted. However, some of the

routed pin pairs may have to be unrouted to make it possible to route the remaining pin pairs, so these results are not directly comparable.

Table 4. Comparison of number of remaining pin pairs and total wire lengths from ECLRoute & SPRoute

	Number of pin pairs left		Total wire length	
Input file	**ECLRoute**	**SPRoute**	**ECLRoute**	**SPRoute**
adaptec1	14224	137009	3420664	3433014
adaptec2	15956	68335	3235094	3254417
adaptec3	30614	82457	9408487	9476061
adaptec4	11388	5610	8934023	8941814
adaptec5	59790	295610	9882678	9987092
bigblue1	12622	255878	3469144	3493099
bigblue2	28789	519234	4655280	4667581
bigblue3	11251	79609	7766981	7747762
bigblue4	51388	1305161	11833361	11706977
newblue1	8181	482740	2349607	2336317
newblue2	619	3767	4645512	4622653
newblue3	9678	355913	7419601	7647518
newblue4	32767	1541907	8017785	8045681
newblue5	75662	339399	14346024	14434224
newblue6	58367	566334	9842371	9910800
newblue7	98846	3558271	17981804	17985483
Geo-mean	20,076.69	207,773.94	6831341.19	6857344.744

The wire length comparison in the right half of Table 4 shows that ECL Route requires 0.38% shorter wires compared to SPRoute considering the geo-mean. However, these results do not account for the complex paths that the unrouted pin pairs might need, so they merely represent a lower bound.

11. CONCLUSION

For netlists having about 0.2 to 2.6 million nets, the serial version of ECL Route is over 6x faster with a wirelength that is 0.38% lower when compared to the SPRoute solution. When ported to the GPU, the I-routing function is 2.4x times faster than the serial ECL Route code. Overall, the ECL Route code can connect more pin pairs compared to the SPRoute solution, leaving behind 10x fewer unrouted pin pairs at the end of pattern routing.

In the chip industry, the speedup obtained during the routing phase can affect the time to market of a product. Parallelizing routing algorithms can decrease this greatly. Besides these, reduced wire lengths help in reducing power and layout area.

The aim of this thesis is to encourage porting over routing algorithms such as the L-routes to run on the GPU. Other future work can also include an extension to the code to enable routing of the remaining pin pairs to study their effect on the overall run time and performance.

REFERENCES

[1] VLSI Physical Design: From Graph Partitioning to Timing Closure by Andrew B. Kahng, Jens Lienig, Igor L. Markov & Jin Hu

[2] Wikipidea, https://en.wikipedia.org/wiki/Taxicab_geometry

[3] 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.

[4] ISPD 2008 Global Routing Contest, http://www.ispd.cc/contests/08/ispd08rc.html

[5] T.-H. Wu, et al. GRIP: Scalable 3D global routing using integer programming. DAC, 2009.

[6] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "A parallel integer programming approach to global routing," in *Proc. DAC*, 2010, pp. 194–199.

[7] 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.

[8] 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.

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

[10] He, J., Burtscher, M., Manohar, R., & Pingali, K. SPRoute: A Scalable Parallel Negotiation-based Global Router. IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019.

[11] Galois, https://github.com/IntelligentSoftwareSystems/Galois