Scalable and Deterministic Timing-Driven Parallel Placement for FPGAs

by

Chao Chris Wang

BASc in Engineering Science, University of Toronto, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2011

Abstract

This thesis describes a parallel implementation of the timing-driven VPR 5.0 simulated-annealing placement engine. By partitioning the grid into regions and allowing distant data to grow stale, it is possible to consider a large number of non-conflicting moves in parallel and achieve a deterministic result. The full timing-driven placement algorithm is parallelized, including swap evaluation, bounding-box calculation and the detailed timing-analysis updates. The partitioned region approach slightly degrades the placement quality, but this is necessary to expose greater parallelism. We also suggest a method to recover the lost quality.

In simulated annealing, runtime can be shortened at the expense of quality. Using this method, the serial placer can achieve a maximum speedup of 100X while quality metrics degrades as much as 100%. In contrast, the parallel placer can scale beyond 500X with all quality metrics degrading by less than 30%. Specifically, at the point where the parallel placer begins to dominate over the serial placer, the post-routing minimum channel width, wirelength and critical-path delay degrades 13%, 10% and 7% respectively on average compared to VPR's original algorithm, while achieving a 140X to 200X speedup 25 threads. Finally, it is shown that the amount of degradation in the parallel placer is independent of the number of threads used.

Preface

[1] C. C. Wang and G. G. F. Lemieux, "Scalable and deterministic timing-driven parallel placement for FPGAs," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 153 - 162.

[2] J. B. Goeders, G. G. F. Lemieux and S. J. E. Wilton, "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition," to appear in *2011 International Conference on ReConFigurable Computing and FPGAs*, 2011.

Portions of Chapter 3, 4, and 5 have been published at FPGA 2011 [1]. A version of Section 6.4 has been submitted for publication. In all cases, the algorithm design, implementation, experimentation and the analysis of results were solely conducted by the author, Chris Wang.

The half-box window decomposition scheme (method 2 of Section 3.4.1) and Equation 3.1 were proposed by Jeffrey Goeders et al. Their publication will appear at ReConFig 2011 [2].

Table of Contents

List of Tables

List of Figures

Glossary

API Application Programming Interface 23

ASIC Application Specific Integrated Circuit 6

BLE Basic Logic Element 7

CAD Computer-Aided Design 1

CLB Configurable Logic Block 3

DAG Directed Acyclic Graph 10

FPGA Field Programmable Gate Array 1

HBWD Half-box Window Decomposition 34

HDL Hardware Description Language 9

HPWL Half-perimeter Wirelength 12

IP Intellectual Property 7

LAB Logic Array Block 11

LAN Local Area Network 25

LUT Look-Up Table 5

MPPA Massively Parallel Processor Array 27

QoR Quality of Result 1

SMP Symmetric Multi-Processing 22

T-VPack Timing-driven Versatile Packing 12

VPR Versatile Place and Route 3

Acknowledgments

I would like to thank my supervisor Dr. Guy Lemieux for his encouragement and support over the past two years. Without his determination and grateful insights, this thesis would not had been possible.

Thanks to all the members of the SOC lab for the fruitful discussions, especially to Jeffrey Goeders for his contributions to this work. Special thanks to Dr. Mark Greenstreet for getting me started with parallel programming as well as allowing me to perform experiments on his Niagara machine.

Thanks to all of my friends that have entertained me throughout my degree. Specifically, Ruby for housing my computer while I was away on internship and Jack (who requested that I should thank him for something) for reading an early version of this thesis.

To my family and my fiancée Karen, thank you for being by my side and supporting me over the years.

Chapter 1

Introduction

1.1 Motivation

As Field Programmable Gate Array (FPGA) logic capacity steadily increases at the rate anticipated by Moore's Law, FPGA Computer-Aided Design (CAD) tools must synthesize, place and route more logic blocks and more nets every generation. To make things more interesting, exotic silicon interconnection technologies are making their debut and this has allowed FPGA logic capacity to achieve even 'more than Moore' under special circumstances [3]. While this benefits the logic-hungry FPGA users, it further complicates the work allocated to FPGA CAD tools. Keeping runtime and Quality of Result (QoR) constant while the number of objects continues to grow is a demanding task. This is mostly due to fact that processor performance improvements have not been tracking FPGA capacity growth since the mid-2000s as shown in Figure 1.1. Unfortunately, the current market trend does not suggest processor speeds will increase in the near future either.

To keep runtime in check, the two main companies offering high-capacity

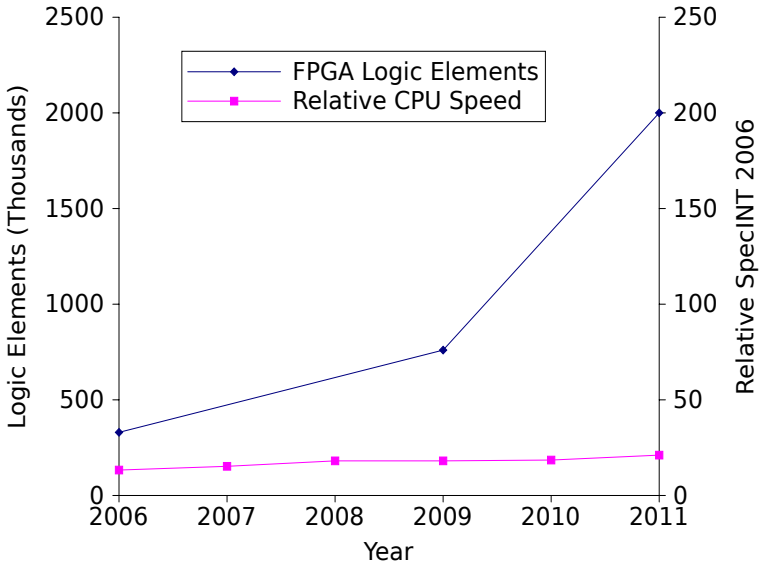


Figure 1.1: CPU speed versus FPGA logic capacity

FPGAs, Altera and Xilinx, have been continuously optimizing their tools. While this has helped, it is unlikely that such algorithm engineering efforts can be sustained at the rate required by several more generations of Moore's Law. As a result, continuous technology scaling without comparable scaling of FPGA synthesis runtime will lead to a runtime crisis. The runtime crisis manifests itself as a reduction in productivity and an increase in engineering costs.

With the current market trend of increasing the number of CPU cores rather than faster CPU cores, a promising solution to the runtime crisis is to employ parallel CAD algorithms. This would effectively allow the number of working processor cores and FPGA capacity to both scale at similar rates. Seeing the opportunity, Altera [4] and Xilinx [5] have both started to implement parallel algorithms that offer some runtime improvement.

1.2 Objective & Contribution

Mapping a larget circuit to an FPGA can take hours or even days. One of the most time-consuming steps in the FPGA CAD flow is placement. A good quality placement is essential to the overall quality; it can influence the interconnect delay, congestion, wirelength and power. Therefore, the parallel algorithms must demonstrate good scaling results not only in terms of runtime, but also a good QoR is needed. In addition, a deterministic (reproducible) result is needed to ease the debugging and verification efforts [6] as it is tremendously difficult to identify issues with an irreproducible result. It is not only useful during algorithm development where regression tests must compute the same answer, but also during customer support where error reproducibility becomes important to confirm bugs.

Simulated annealing, the placement engine used in Versatile Place and Route (VPR) and Altera's Quartus II tools, is widely regarded as producing very good QoR and being able to handle complex legalization constraints. A comparison between a simulated-annealing based algorithm, namely VPR, and a few best-in-class academic placers based on other techniques was recently presented in [7]. The conclusion was that "simulated annealing based placement would still be in dominant use for a few more device generations. [7]" In VPR, the number of moves needed to find an optimized solution grows as $O(N^{4/3})$, a result adopted from [8], where N is the number of Configurable Logic Blocks (CLBs) being placed. As more objects must be placed, the VPR placement algorithm slows down due to super-linear scaling.

Parallel placement research has been underway for more than two decades. To the best of our knowledge, all published works with the exception of [6] have tar-

geted wirelength only. Furthermore, determinism, an extremely important feature for bug reproducibility and testing [6], has not been considered except in [6, 9]. Our work addresses both of these issues, and is the first academic parallel placer that produces a timing-driven and deterministic result with significant speedup.

This thesis presents a deterministic and timing-driven simulated-annealing placement algorithm. Speedup is achieved by allowing moves and committing to occur in parallel. The key is to avoid generating moves that have *hard conflicts* by greatly restricting the range of motion for CLBs during each move. By using stale placement information for distant CLBs being moved by other threads, the overhead of fine-grained synchronization between threads can be avoided. Although threads lack fine-grained synchronization, they still achieve a deterministic (reproducible) result that depends only upon the number of threads and does not depend on processors nor race conditions. Although the final result does depend upon the number of threads, the QoR is relatively consistent as the number of threads varies. Also, any number of processors can be used to achieve the same result with a fixed number of threads.

The work in this thesis is based on the open-source VPR 5.0 [10]. With 25-threads operating in parallel, it achieves a speedup ranging from 140X to 200X compared to the original single-threaded VPR algorithm with post-placement minimum channel width, wirelength and critical-path delay degrading by 13%, 10%, and 7% on average, respectively.

A paper based on this work has been published at FPGA 2011 [1]. An enhancement to the block decomposition scheme, the half-box window decomposition scheme (method 2 of Section 3.4.1), and converting the number of iterations per temperature to a commonly known metric (Equation 3.1), both performed by

Jeffrey Goeders et al. will be published at ReConFig 2011 [2]. Finally, placement based on individual Look-Up Tables (LUTs) (Section 6.4) has been submitted for publication.

1.3 Thesis Organization

This thesis is organized as follows: Chapter 2 presents background information on FPGA CAD, parallel programming, parallel placement and previous works. Chapter 3 describes the basic parallel placement algorithm. Chapter 4 explains the optimizations chosen and parameter tuning process. Chapter 5 describes the benchmarks used, hardware environment and experimental methods and results. Chapter 6 presents future works and conclusions are given in Chapter 7.

Chapter 2

Background

This chapter first presents an overview of FPGAs, and then explains one of the commonly used architectures – the island-style architecture. Second, the FPGA CAD tool flow is introduced with an emphasis on placement. Finally, previous placement works are presented and two algorithms which form a basis for the work in this thesis are presented in detail.

2.1 FPGA Overview

An FPGA is a highly versatile integrated circuit that is capable of implementing *any* logical function. It offers a few notable advantages over other technologies, such as Application Specific Integrated Circuits (ASICs). First, FPGAs allow a fast time-to-market as the entire chip is pre-fabricated and the only time needed is to design the logic functions, compile the design using FPGA CAD tools, and load the resulting configuration bitstream into the device. This is much easier than custom design of a chip because silicon-related knowledge is not required and CAD tools are much simpler. Second, it requires a very low non-recurring expense, by

avoiding the need for custom masks, which may cost millions of dollars per set. Third, the re-programmability allows the designer to reconfigure the device in the event of a design change or an implementation bug. This eliminates the need to modify masks and re-fabricate the chip which is both time consuming and costly.

2.2 FPGA Architecture

This thesis targets FPGAs with island-style architectures as depicted in Figure 2.1. It is the most popular choice for FPGA research and commercial vendors alike. As seen, the chip is surrounded by inputs and outputs which connect to external signals. Signals travel from the IOs through the routing network and are processed in CLBs. CLBs may be replaced by heterogeneous blocks, such as configurable memory or multiplier blocks, or other Intellectual Property (IP) hard blocks. Processed signals will either be routed back to the IO as an output or routed to another destination for further processing.

FPGAs consist of numerous CLBs, also known as clusters, which contain a pre-configured number of Basic Logic Elements (BLEs), N. A typical BLE and a cluster are shown in Figure 2.2. The BLE on the left side consists of a K-input LUT with two possible outputs selectable by a multiplexer. The path that passes through a flip-flop is the clocked output and the direct wire indicates the non-clocked (combinational) output path. A LUT is the most basic element of an FPGA; it can be used to implement any combinational logic function utilizing up to K inputs. The cluster on the right side contains N BLEs, each with K inputs.

For this thesis, an FPGA architecture based on the following parameters is used:

- LUT Size (K): 6

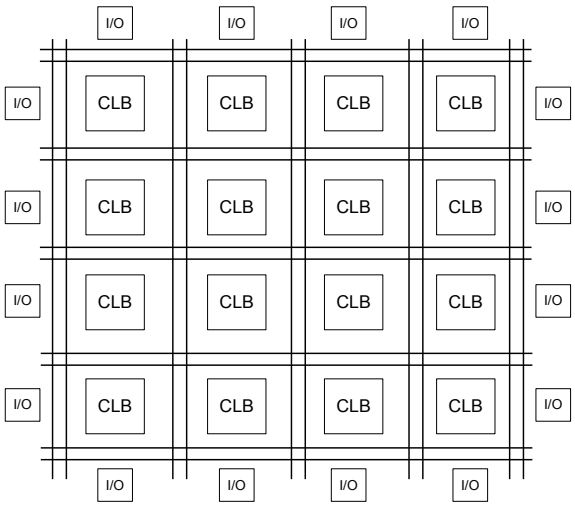


Figure 2.1: Island style FPGA, based on [11]

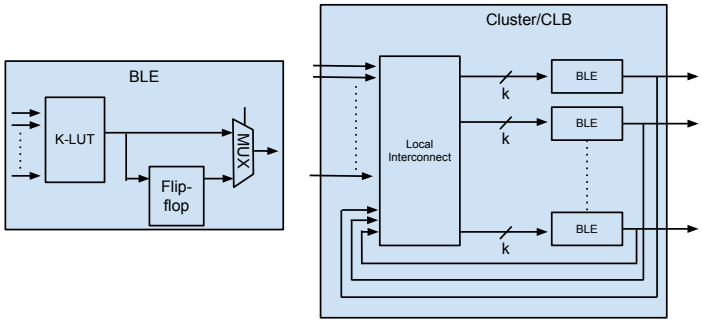


Figure 2.2: BLE and cluster, based on [11]

- Cluster Size (N): 10
- Inputs per cluster: 35
- Length of wires: 4
- Wire type: Unidirectional

- Switch Block Type: Wilton
- C-Block Input Connectivity: 0.15
- C-Block Output Connectivity: 1/*N*
- Switch type: buffered
- Transistor process technology: 65nm

The architecture files used in this thesis are obtained from the iFAR repository [12], which model realistic FPGA architectures. [1]

2.3 FPGA CAD Tool Flow

In order to implement logical functions on FPGAs, the circuits must first be described using Hardware Description Language (HDL), such as Verilog. The description is then passed through a series of conversions and finally resulting in a bitstream which can be used to program the FPGA. The conversion process is known as the FPGA CAD tool flow and is outlined in Figure 2.3.

The flow is traditionally divided into five distinct stages: synthesis, technology mapping, clustering, placement and routing. The following sections explain each of the five stages.

2.3.1 Synthesis

Synthesis is the first step in the CAD tool flow. It converts the circuit described in HDL into gate-level logic functions and flip-flops. Technology-independent optimizations are also performed here, and the output is a netlist of connected cells representing the input circuit.

[1] n10k06l04.fc15.area1delay1.cmos65nm.bptm taken from iFAR version 0.3-296.

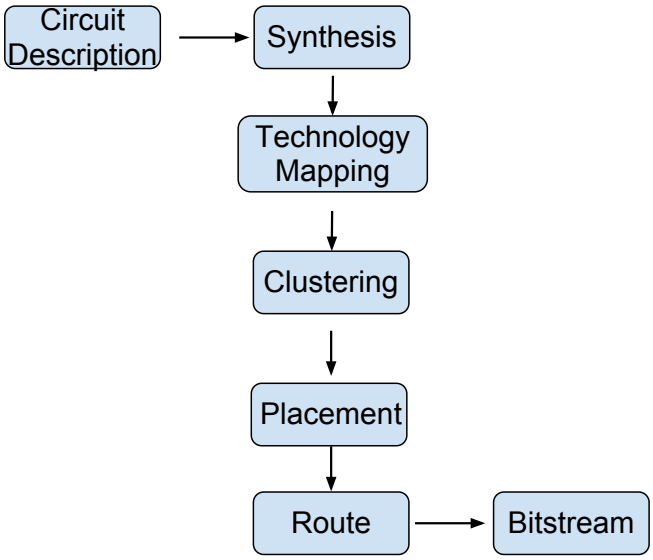


Figure 2.3: FPGA CAD tool flow

2.3.2 Technology Mapping

Technology mapping is responsible for translating the entire input circuit obtained from the synthesis stage output into a netlist consisting of K-input LUTs and flip-flops. An example of replacing some logic with a 4-input LUT is shown in Figure 2.4. The input circuit is first converted to an equivalent Directed Acyclic Graph (DAG), which is then mapped to LUTs with constraints on the maximum number of inputs (K) and outputs (1). Graphically, this can be seen as finding a group of nodes that have at most K inputs and one output to form each LUT.

Technology mapping algorithms strive to minimize specific objectives, such as area, delay, power or any combination thereof. The output is a netlist of pre-configured LUTs and flip-flops.

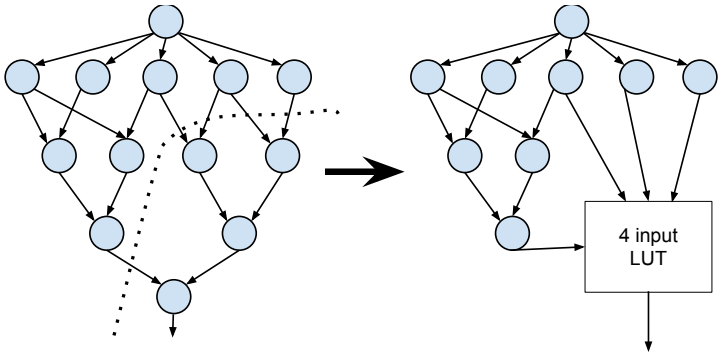


Figure 2.4: Technology mapping, based on [13]

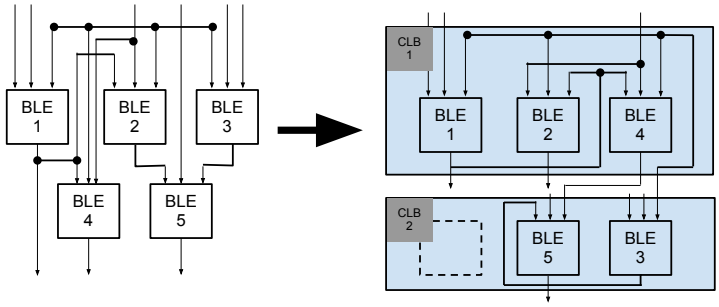


Figure 2.5: Clustering, based on [13]

2.3.3 Clustering

The main goal of clustering is to pack the LUTs and flip-flops created in technology mapping into clusters, abiding to the constraint that no more than *N* LUTs or *N* flip-flops can be included to form each cluster as shown in Figure 2.5. The other goal of clustering is to pack LUTs and flip-flops into BLEs. Although structurally similar, different vendors have adopted their own terminology to describe a cluster. Xilinx uses the term CLB and Altera uses the term Logic Array Block (LAB) to describe a cluster. Both terms have identical meaning within the scope of this thesis.

Minimizing delay is one of the many goals a clustering algorithm may choose

to optimize. Delay optimization requires the understanding that BLEs contained within the same cluster communicate through intra-cluster connections, which are faster than inter-cluster connections, also known as the routing network. A larger N value allows more BLEs to share intra-cluster connections. However, careful trade-off analysis must be considered as the intra-cluster connection latency degrades with increased cluster size. Other goals that clustering algorithms may also target are area, power, routability, congestion, etc. The clustering algorithm outputs an optimized network of CLBs, each containing up to N BLEs.

For this thesis, Timing-driven Versatile Packing (T-VPACK) [14] is used to perform the clustering of circuits.

2.3.4 Placement

The placement step involves assigning each CLB in the circuit to a unique physical location on the FPGA chip, as shown in Figure 2.6. Typical placement algorithms aim to minimize Half-perimeter Wirelength (HPWL), also known as the bounding-box cost, and delay. As well, some advanced placers aim at reducing power, congestion or other criteria, as well combined objectives. Decisions made by the placer are highly influential to the overall solution quality, since CLBs are locked in the placed locations after this stage. Hence it is imperative that trade-offs are carefully evaluated in this stage to ensure high-quality results.

Placement is a NP-complete problem; therefore, heuristic algorithms are used to ensure solutions can be found within a reasonable amount of time. Traditional serial placement algorithms can be grouped into four different categories: a) simulated-annealing, b) partition-based, c) analytical and d) others such as genetic algorithms. While each category poses different trade-offs, simulated-annealing

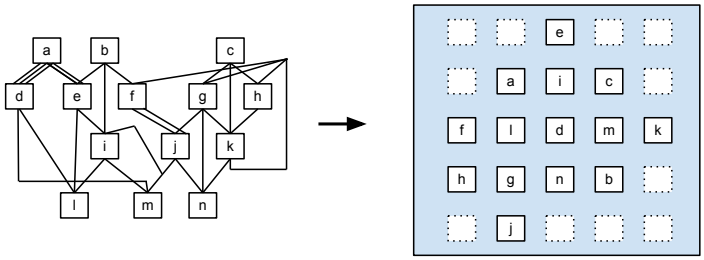


Figure 2.6: Placement, based on [13]

stands out as the dominating technique and is widely regarded as producing very good QoR and being able to handle complex legalization constraints. VPR [15], being a simulated-annealing based placer, has become the *de facto* standard for uniprocessor FPGA placement in academic research. The following section will describe the serial simulated-annealing algorithm implemented in VPR [15], as it forms the basis the algorithm being parallelized.

Simulated Annealing

As the name suggests, simulated annealing was created to imitate annealing, a process commonly used in metallurgy and material science. Annealing is accomplished by heating the material to a high temperature then cooling it slowly to achieve the desired structural properties. Cooling the material rapidly is another technique used, and this process is known as quenching. This idea has been applied to various optimization problems in science [16], including FPGA placement which is implemented in VPR [15]. VPR is an open-source FPGA place and route tool, and it has became the *de facto* standard

The pseudocode for VPR's simulated-annealing is shown in Figure 2.7. The algorithm begins with a few initialization tasks: generating an initial placement and

```
1: set initial placement()
2: set initial temperature: T
3: set iterations per temperature: inner_num
4: set swap range: R_limit
5: while !exit_condition do
6:    for n = 1 to inner_num · N^(4/3) do
7:       try_swap
8:    end for
9:    update T
10:   update R_limit
11:   update net criticality
12: end while
```

Figure 2.7: Simulated annealing pseudocode

```
1: identify swap candidates
2: calculate Δcost = CostBefore() - CostAfter()
3: if rand(0,1) < e^(-ΔC/T) then
4:    AcceptSwap()
5: else
6:    RejectSwap()
7: end if
```

Figure 2.8: Try_swap() pseudocode

setting up initial temperature, number of swaps per temperature and determining the range limit for each swap.

An initial random placement is obtained by randomly placing each cluster at a different (non-overlapping) location on the grid. Then, N (the total number of CLBs and I/O pads in a circuit) random swaps are performed, and the standard deviation of the cost of these N different configurations is computed [15]. The algorithm sets the initial temperature as 20·(standard deviation), to ensure that the initial temperature will adjust according to the circuit.

Temperature is used to control the probability that a bad move, a move which increases the overall cost, will be accepted; a good move, one that improves the

Table 2.1: Adaptive simulated-annealing schedule

Phase	Fraction of moves accepted (R_{accept})	α
Randomization	$0.96 < R_{accept}$	0.5
Initial Improvement	$0.8 < R_{accept} \leq 0.96$	0.9
Rapid Improvement	$0.15 < R_{accept} \leq 0.8$	0.95
Greedy Optimization	$R_{accept} \leq 0.15$	0.8

overall cost, is always accepted regardless of temperature. The number of swaps per temperature is calculated using Equation 2.1, where *inner_num* is a user-defined constant and *N* is the total number of CLBs and I/O pads in a circuit.

$$inner\_num \cdot \mathrm{N}^{4/3} \tag{2.1}$$

An adaptive annealing schedule is used which updates the temperature at the end of every swap iteration. The new temperature is computed as $T_{new} = \alpha T_{old}$, where α is defined by Table 2.1 [15].

R_{accept} is the ratio of moves that were accepted during the previous iteration. It is desirable to keep R_{accept} around 0.44 for high-quality results [8, 15, 17]. R_{limit}, a range limiter which constrains the Manhattan distance[2] which each pair of swap candidates are located, was introduced to facilitate the acceptance rate [15, 18, 19]. A small R_{limit} value implies that only 'local swaps' with its nearby neighbors are allowed, which tends to result in small placement cost changes. The opposite is true for large R_{limit} values. R_{limit} is updated according to Equation 2.2. It is initially set to the entire grid, and shrinks gradually during the middle stages and becomes 1 during the low-temperature parts of the anneal.

[2]Manhattan distance is the distance between two swap candidates calculated utilizing only horizontal and vertical paths (not diagonal paths).

$$R_{limit}^{new} = R_{limit}^{prev} \cdot (1 - 0.44 + R_{accept}^{prev})$$
$$\text{such that:} \quad 1 \leq R_{limit} \leq grid\ size \tag{2.2}$$

The simulated-annealing engine will terminate when the exit condition is met, shown by Equation 2.3, where *number_of_nets* is the total number of nets in the input circuit and *cost* for a timing-driven placement algorithm is a combination of bounding box (Equation 2.6) and critical-path delay (Equation 2.11). The cost calculation equation is shown in Equation 2.4, and λ is used to vary proportion of each of the components.

$$temperature < 0.005 \cdot \frac{cost}{number\_of\_nets} \tag{2.3}$$

$$cost = \lambda \cdot Timing\_cost + (1 - \lambda) \cdot Wiring\_cost \tag{2.4}$$

The following section explains the swap evaluation method. The core simulated-annealing engine consists of the *try_swap* function as shown in Figure 2.8. Prior to entering this function, the algorithm will have already identified a swap-from candidate, which is a random block anywhere on the grid. The first task is to identify a swap-to candidate, which is another random block within a Manhattan distance of R_{limit} away from the swap-from candidate. Next, the change in cost (Δc) will be calculated assuming the swap will take place.

For timing-driven placement algorithms, Δc is the sum of two components: wiring cost and timing cost weighted by a variable λ to vary the importance of

each component. A λ value of 0 yields a purely bounding box driven placement algorithm, and a λ value of 1 yields a purely timing-driven placement. This is shown in Equation 2.5.

$$\Delta c = \lambda \cdot \frac{\Delta Timing\_Cost}{Previous\_Timing\_Cost} + (1-\lambda) \cdot \frac{\Delta Wiring\_Cost}{Previous\_Wiring\_Cost} \tag{2.5}$$

The *timing_cost* calculation is described later in Section 2.4, the *wiring_cost* is calculated with Equation 2.6.

$$wiring\_cost = \sum_{i=1}^{N_{nets}} q(i) \cdot [\frac{bb_x(i)}{C_{av,x}(i)} + \frac{bb_y(i)}{C_{av,y}(i)}] \tag{2.6}$$

Here, N_{nets} is the total number of nets in the circuit. Variables bb_x and bb_y represent the horizontal and vertical span of nets connected to net i as illustrated in Figure 2.9. $C_{av,x}$ and $C_{av,y}$ are the average channel capacities (in tracks) in the x and y direction respectively, over the bounding box of net i. The $q(i)$ factor compensates for the fact that the bounding box wire length model underestimates the wiring necessary to connect nets with more than three terminals; q is 1 for nets with 3 or fewer terminals, and slowly increases to 2.70 for nets with 50 terminals [15].

If the calculated Δc is negative, indicating that the move is good, then the swap will be accepted. Otherwise, a random number will be compared with $e^{\frac{-\Delta c}{T}}$, where T is the current temperature, to determine whether the swap will be accepted. As seen, even swaps that result in a worse cost metric could be accepted. This is to ensure the algorithm will not be trapped at a local minimum solution. The ultimate goal is to minimize the cost function of the circuit through iterative placement stages.

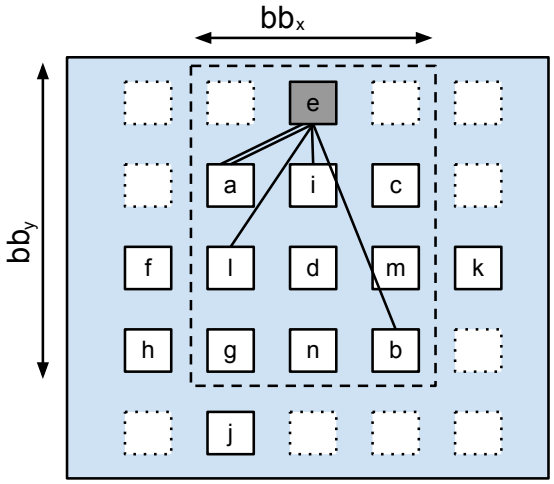


Figure 2.9: Example bounding-box of a 5 terminal net, based on [20]

2.3.5 Routing

As routing is the last step in the FPGA CAD tool flow occurring after placement, it has the most-direct impact on the quality of the final circuit. The purpose of routing is to find a suitable path for all of the interconnect signals utilizing the programmable routing switches. Most routers aim to minimize routing area (channel width), critical-path delay, congestion or any combinations of the aforementioned goals.

Traditional routing algorithms can be classified into two categories: two-step routers and single-step routers. A two-step router [21–25] separates the process into global routing, where signals are assigned pins and channels, and detailed routing, where signals are further assigned specific wiring tracks within the assigned channel. Single-step routers [26–29] combine the global and detail routing steps together. In general, FPGA routers are based on the latter approach to cope with its routing resource architecture, which is both limited and constrained. Two-step

routers are better suited for ASIC designs.

VPR employs a single-step router based on the PathFinder [27] algorithm. PathFinder's negotiated congestion algorithm first routes all wires using the shortest path, despite over-consuming resources. It will then iteratively increase the cost of the over-subscribed resource and rip-up and re-route the affected wires. In practice, this will cause some of the signals that do not absolutely need the over-subscribed resource to route around the congested region to eventually resolve the over-usage issue. In order for VPR to achieve a timing-driven result, critical-paths are routed with a higher priority over the other nets, giving it access to all of the resources. This effectively ensures the critical-paths are delay-optimized.

2.4 Timing Analysis

VPR can operate in two modes: wire-length driven or combined wire-length and timing-driven placement. It has been demonstrated that an average of 42% improvement in post-place-and-route speed can be achieved while sacrificing only 5% wire length by operating in the timing-driven mode [20]. Clearly, timing-driven placement is highly beneficial for FPGA devices and the following section will give a brief introduction to the timing-analysis engine used in VPR.

In order to perform timing analysis, the circuit needs to be converted into a graph. The nodes represent the input and output pins of circuit elements and edges are used to model the physical connection delays between the nodes. Flip-flops are removed, with the inputs treated as virtual outputs and their outputs being treated as virtual inputs. The number of nodes and edges is a constant for a given circuit, while values on the edges are the only variables which depend on the physical location of the nodes it is connected to.

Timing analysis starts after all blocks have been assigned a location. The edges are first updated to reflect the actual physical delays between every pair of nodes. Using these values, a breadth first traversal is then performed originating at all source nodes. Source nodes are assigned a $T_{arrival}$ value of 0; the $T_{arrival}$ value for the remainder of the nodes is calculated using Equation 2.7:

$$T_{arrival}(i) = \underset{\forall j \in fanin(i)}{MAX} \{T_{arrival}(j) + delay(j,i)\} \tag{2.7}$$

Where node i is the node being evaluated and $delay(j,i)$ represents the delay value of the edge joining node j to node i. The maximum arrival time is called the delay of the circuit, D_{max} [20].

Using this information, the amount of delay that can be added to a path before it becomes critical, *slack*, is calculated in two steps. First, $T_{required}$ is calculated starting at all sink nodes, which are assigned a value of D_{max}. The $T_{required}$ for the remainder of the nodes is calculated using Equation 2.8:

$$T_{required}(i) = \underset{\forall j \in fanout(i)}{MIN} \{T_{required}(j) - delay(i,j)\} \tag{2.8}$$

Then the *slack* of a path driven from i to j is defined by Equation 2.9.

$$slack(i,j) = T_{required}(j) - T_{arrival}(i) - delay(i,j) \tag{2.9}$$

Using this information, the criticality is calculated with Equation 2.10 and the *Timing_cost* of the circuit is calculated with Equation 2.11

$$Criticality(i,j) = 1 - \frac{slack(i,j)}{D_{max}} \tag{2.10}$$

$$Timing\_cost = \sum_{\forall i,j \in circuit} Delay(i,j) \cdot Criticality(i,j)^{Criticality\_Exponent} \quad (2.11)$$

The $Criticality\_Exponent$ is a constant for a given temperature used to heavily weigh connections that are on the critical-path of the circuit, while giving less weight to other paths. It starts off with a constant value of 1 and gradually increases to 8 as the R_{limit} value shrinks.

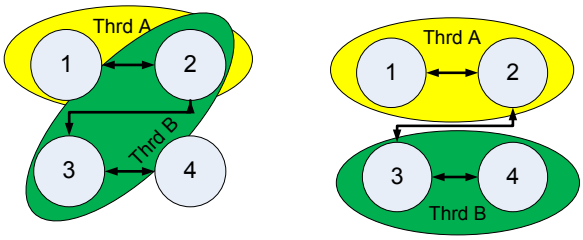
2.5 Parallel Placement - Conflicts

Two important terms in parallel placement algorithm development are *hard conflicts* and *soft conflicts*. Although the conflict term is used rather often, hard and soft conflicts have never been formally defined. In order to prevent confusion, a definition will be presented here which will be consistent with the usage throughout this thesis.

A simple circuit with four CLBs, numbered 1 through 4, is shown in Figure 2.10. The arrows represent nets that connect CLBs, the green and yellow (light) shades each represent a swap pair that two threads are considering respectively.

A *hard conflict* arises when the same CLB is considered by more than one thread concurrently. In the first example, threads A and B both consider block 2 for a move, thereby producing a hard conflict. Hard conflicts are problematic in multi-threaded programming since it may lead to race conditions if not considered carefully. In this example, if the order of swaps is not constrained, then the final outcome would be dependent on the order of the swaps that took place.

A *soft conflict* arises when different CLBs that are connected by the same net



(a) Hard Conflict (b) Soft Conflict

Figure 2.10: Illustration of a hard and soft conflict

are considered concurrently. In the second example, thread B now considers CLB 3 and 4 to resolve the hard conflict, but a soft conflict remains since blocks 2 and 3 are connected by the same net. The soft conflict arises because each thread makes cost-based decisions that depend on these shared nets, but the parallel movement of CLBs attached to these shared nets may invalidate the decision. High-fanout nets make the probability of a soft conflict to be high, but they are also less likely to result in a cost change.

2.6 Parallel Programming Background

2.6.1 Symmetric Multi-Processing

A Symmetric Multi-Processing (SMP) architecture system is a multi-processor machine which utilizes a single shared-memory memory address space amongst all processing nodes, and all of the processing units are identical. One of the main advantages of these machines is that all parallel processing units have an identical view of all memory elements, regardless where the memory is physically located. With the help of memory consistency protocols, a shared-memory multicore machine greatly simpliefies the process of parallel programming.

2.6.2 POSIX Threads

Pthreads, short for POSIX Threads, is a programming library used to implement parallel programs on SMP machines. Pthreads is a relatively light-weight library, as compared to *fork()* [30], and it can be added to any C based program by simply linking with the Pthreads library.

The Pthreads Application Programming Interface (API) is used to control the creation, management and destruction of threads. A thread is defined as an independent stream of instructions that can be scheduled to run by the operating system [30]. Pthreads also provides a few general purpose synchronization mechanisms, including barriers, locks, condition variables and mutexes, which are used for inter-processor synchronization tasks. However, they do not solve the typical parallel programming challenges such as load imbalance, race conditions, inter-thread communication latencies, etc.

2.6.3 Determinism

Determinism is a property of a program such that, for a given constant set of inputs, the outcome is identical regardless of the number of time the program is executed. While this is commonly the case for a single-threaded program, achieving determinism for a multi-threaded program is not straightforward. Various synchronization mechanisms must be employed to constrain the order of global write operations to shared memory, and special care must be taken into consideration when designing the algorithm in order to achieve this. Such an intricate implementation must offer significant benefits to offset the difficulties in its creation. Debugging and reproducibility are the biggest advantages of having a deterministic program, as it is otherwise very difficult to debug a program in which the execution (thus

the location of error) changes every time! Also, the reproducibility property is also helpful in many settings, such as in customer support setting where the reported bug needs to be reproduced, or in regression testing where the same answer must be produced.

Determinism is a subset of *serial equivalence*, which has an additional constraint such that the outcome is independent of the number of threads utilized. A program that is serially equivalent is deterministic, but not vice-versa. Serial equivalence may be useful under certain conditions, but for the purpose of this thesis, determinism is sufficient since the number of threads used can simply be treated as an input for reproducibility purposes. For example, the same 16-thread solution can be reproduced on a system with 16 processor cores, 8 processor cores, or even 1 processor core.

2.7 Previous Works

Parallel simulated-annealing placement algorithms can be loosely categorized into two groups: the shared and partitioned region.

- The first group of algorithms allows all processors to work within the same region (often the entire grid), but restricts swaps that are being evaluated in parallel, such as being from independent sets [6, 31–33]. Some algorithms employ a speculative move proposal to further accelerate the algorithm [6], and a dependency checker, executed serially, is used to ensure that no hard conflict has occurred and that soft conflicts are resolved with recalculation. The algorithm as reported in Ludwin's work [6] achieved a speed up of 2.1x and 2.4x on four and eight cores, respectively. Further speedup beyond eight cores does not look promising. The main reason is that the probability of a

soft or hard conflict increases with the number of processors, making it more difficult to find moves that do not conflict.

- The second group of algorithms allocates a specific region to each processor, with no or minimal overlap, and different methods are employed to allow blocks to migrate from one region to the other [9, 34]. Sun and Sechen [34] employed an algorithm based on dynamic region generation by dividing the chip vertically and horizontally in alternating iterations. It was implemented on multiple machines connected over a Local Area Network (LAN). During each iteration, each machine receives an independent region to work on, and is terminated by the first machine that completes a pre-determined number of moves. In order to minimize communication, all machines only update changes at the end of each iteration, resulting in move evaluation being based on the cell locations from the previous iteration. It achieved a speedup of 5.3x using six machines and yielded comparable results to a serial algorithm. While the dynamic region generation may have helped with cell movement, it greatly hinders the amount of parallelism the algorithm can achieve and thereby limits the overall speedup. In addition, this algorithm and all of the other ones in this group consider bounding-box cost only and are non-deterministic, making them unsuitable to be implemented in commercial tools.

A list of past parallel placement work is summarized in Table 2.2.

In recent work, Ludwin et al. describe a parallel timing-driven annealing-based algorithm implemented in Quartus II [6, 39] which evaluates many moves in parallel, but serially commits the moves to achieve a serially equivalent placement. A

Table 2.2: Summary of past parallel placement work

Reference (Year)	Algorithm Category	Hardware Targeted	Deterministic	Timing-Driven	Result
Casotto (1987) [18]	2	Sequent Balance 8000 (8-processors)	No	No	6.4x on 8 processors
Kravitz (1987) [31]	1	VAX 11/784 (4-processors)	No	No	less than 2.3x on 4 processors
Rose (1988) [35]	1 & 2	6 National Semi. 32016 processors	No	No	about 4 using 5 processors
Banerjee (1990) [32]	1	Hypercube multiprocessors	No	No	about 8 on 16 processors
Witte (1991) [36]	1	Hypercube multiprocessors	Yes	No	3.3x on 16 processors
Sun (1994) [34]	2	Networks of machines	No	No	5.3x on 6 machines
Wrighton (2003) [37]	2	FPGAs	No	No	500x-2500x over CPUs
Smecher (2009) [38]	2	MPPAs	No	No	1/256 less swaps needed with 1024 cores
Choong (2010) [33]	1	GPU	No	No	10x on NVIVIDA GTX280
Ludwin (2008/2011) [6, 39]	1	Multiprocessors	Yes	Yes	2.1x and 2.4x on 4 and 8 processors
This work	2	Multiprocessors	Yes	Yes	161x using 25 processors

speedup of 2.1x and 2.4x was demonstrated on 4 and 8 processors respectively, and QoR is equivalent to the serial version. However, this algorithm does not runtime-scale to a large number of cores because multiple cores increase the probability of both hard and soft conflicts between moves. A conflict of any type requires a speculated move to be abandoned, or its cost to be recomputed serially at commit time.

Work by Wrighton and DeHon [37] presented a distributed annealing algorithm for a systolic architecture. While the architecture was prototyped on an FPGA, it was only capable of computing placement for a much smaller array size than the "host" FPGA. Still, it demonstrated that hardware acceleration could obtain significant speedups from 500x to 2500x. The algorithm works by restricting the swap range of each block to its 4 immediate neighbors only. This allowed purely local placement decisions to be made between adjacent elements, thus exposing vast amounts of parallelism. One notable characteristic of this work was the use of stale (old) placement information when making local decisions – this was done deliberately to avoid the overhead of broadcasting updates after every move. Instead, placement information was updated infrequently using a daisy-chain. Overall, this approach suffers from 36% quality degradation in the final placed circuit. The QoR did not depend on information staleness.

More recently, work by Smecher, Wilton and Lemieux [38] demonstrated that the algorithm used in [37] could be applied to placement of communicating tasks for a Massively Parallel Processor Array (MPPA). Since MPPAs contain reasonably powerful CPUs, they can "self-host" or place themselves. The paper further shows that expanding the neighbor region to 8 cells (ie., includes diagonals) or 12 cells (within Manhattan distance of 2) improves QoR to within 5% of traditional

simulated-annealing while still offering significant speedups. One limitation with both of the aforementioned works is that only bounding-box cost is considered. Another limitation is that specialized hardware such as a very large FPGA or an MPPA is required. In this work, we apply techniques from Wrighton et al. [37] and Smecher et al. [38] to the full timing-driven placement algorithm from VPR 5.0 using Pthreads, allowing it to run on readily available shared-memory multicore computers.

Chapter 3

Algorithm Description

3.1 Algorithm Overview

The algorithm presented in this thesis is based on partitioned regions, namely the second group of algorithms described in Section 2.7. The main benefit with this type of algorithm is its ease of scalability due to the distinct region assigned to each processor. The key here is to minimize communication between processors, while maintaining a deterministic result. Achieving good quality is also one of the challenges since this region partitioning scheme constrains block movement. Careful tuning is required to achieve the ideal combination between runtime and quality.

The parallel placement algorithm is implemented directly into VPR 5.0.2 [10], which itself is based on the original VPR first published in [15], by modifying the *try_place()* function and sub-functions. Data structures in the software have been mostly left unmodified, except where needed to create private per-thread data and parallelize the timing update. To alleviate the demand on communication cost

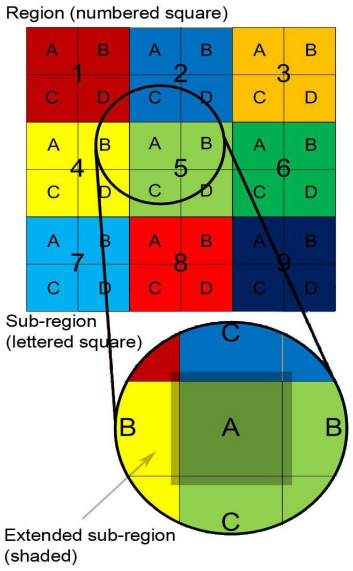


Figure 3.1: Equal-sized grid partition

and to create a deterministic program, local copies of global variables are created to keep track of block location, timing cost and bounding-box cost data. These data are updated frequently enough that they do not risk becoming too stale. For example, prior to evaluating each sub-region, all threads retrieve the latest data from each other.

3.2 Grid Partition

The entire grid is first partitioned into approximately equal-sized private regions and assigned to a unique thread. An example is shown in Figure 3.1 with regions 1 through 9. The increased capacity (number of cells) in IOs around the periphery of the chip is often balanced by the sparse IO cell occupancy. Therefore, it can be estimated that each block, whether CLB or IO, contains approximately the same amount of work.

When the total number of rows/columns in the grid cannot be distributed evenly, the interior regions rows/columns will receive the extra rows/columns first. The reason for this allocation scheme is due to the increased capacity in the IOs as previously mentioned; allocating these extra cells to the peripheral rows/columns could potentially worsen what may already be a somewhat imbalanced distribution.

The number of columns and rows does not need to be same, *i.e.*, each region can be rectangular, thereby allowing a more diverse number of processors to be employed.

The algorithm then subdivides each private region into four (2×2) approximately equal *sub-regions*. Each sub-region then is extended by two rows and two columns in each direction along its boundaries, which are helpful in allowing blocks to migrate between sub-regions to enhance the overall placement quality. This combined region (sub-region extended to include the four extra rows/columns) is called the *extended sub-region*.

To summarize, Figure 3.1 shows 9 *private regions*, numbered 1 through 9 and 4 *sub-regions* for each *private region*, lettered A through D. The zoomed figure at the bottom shows the *extended sub-region* for *sub-region* 5A.

3.3 Parallel Placement Pseudocode

In this algorithm, each thread will iterate sequentially through all CLB locations in its private region and consider the block at each position for a swap or skipped over entirely. However, `PROB_SKIPPED` is used to randomly determine whether the selected block will be considered for a swap. We experimentally determined a good value to be 10% (Section 4.4), implying there is a 90% probability

```
1:  while !exit_condition do
2:    for iteration = 1 to region_place_count do
3:      timing_update_parallel()
4:      barrier()
5:      for n = 1 to num_of_subregion do
6:        global_to_local_data_update()
7:        for each block position in subregion do
8:          if rand[0,100) ≥ PROB_SKIPPED then
9:            try_swap()
10:         end if
11:       end for
12:       local_to_global_data_update()
13:       barrier()
14:     end for
15:     bounding_box_update_parallel()
16:     barrier()
17:   end for
18:   update_anneal_schedule()
19:   barrier()
20: end while
```

Figure 3.2: Thread-level pseudocode

that each block is considered for a swap. Hence roughly 90% of the blocks will be considered each iteration.

The parameter *region_place_count* is used to control the quality versus runtime in a way similar to *inner_num* in VPR. In the psuedocode described, the algorithm makes a constant number of iterations through the grid at each temperature. The constant is *region_place_count*. According to the original VPR equation, the number of swaps is equal to $inner\_num \cdot N^{4/3}$, where *inner_num* is the VPR control parameter for adjusting quality versus runtime. To scale this value as the circuit size changes at the same rate as VPR, Equation 3.1 is used to convert between *inner_num* and *region_place_count*:

$$region\_place\_count = \frac{1}{\texttt{PROB\_SKIPPED}} \cdot N^{\frac{1}{3}} \cdot inner\_num \tag{3.1}$$

The initial term, $\frac{1}{\texttt{PROB\_SKIPPED}}$, is used to account for the blocks that won't be moved. The size of the circuit N only needs to be raised to the $\frac{1}{3}$ power since each inner loop iterates through the entire grid which is already N^1.

Additionally, barriers are needed to enforce synchronization between the threads. Barriers help ensure deterministic behaviour; very little data structure locking is necessary, and of the locks we used, none of them lead to race conditions or non-deterministic behavior.

3.4 Determinism Enforcement

Determinism is achieved partly by ensuring all active swap regions being evaluated concurrently *do not* overlap with each other, thereby avoiding hard conflicts. In another words, no block can be considered by more than one thread at any instance. It is also achieved through several other considerations. Both of these issues are discussed below.

3.4.1 Swap Regions Restriction

The following section explains two ways of restricting the swap_from and swap_to region to achieve determinism. Only the descriptions are presented here, the QoR comparison between the two is shown in Section 4.1.

Method 1 The first method restricts the swap_from region to each sub-region, and the swap_to region to the extended sub-region.

This method starts with every thread placing sub-region A (refer to Fig-

ure 3.1). A barrier is used to ensure no thread can proceed to the next sub-region B before every thread finishes with sub-region A. Each thread will start at the top-left corner of each sub-region and work its way down to the bottom-right corner of its sub-region and consider each block (the swap-from block) for a swap. The swap_to candidate is randomly selected from the swap_to region. After region A completes, all threads will move onto sub-region B, then C, then D, and the process repeats. As all extended sub-regions with the *same* letter do not overlap, race conditions cannot occur, and thus determinism is established.

Method 2 The second method, suggested by Goeders et al. [2], is named the Half-box Window Decomposition (HBWD) method. It considers a slightly different swap-from and swap-to region.

The swap-from and swap-to region considered by HBWD is depicted in Figure 3.3. The goal here is to increase block migration, to improve quality, and to decrease the number of barriers to improve runtime. In this method, the swap-from region spans two of the original sub-regions used in method 1. The swap-to region is simply the swap-from region plus half of one of its neighbor's private regions.

Using the labeling system in Figure 3.1, let's go through the regions considered by thread 5 (in the middle). This method starts with the swap_from region set to the combination of 5A & 5B, and the swap_to region consisting of 2C, 2D, 5A & 5B. Similarly to the Method 1, this thread would start at the top-left corner of 5A and work its way down to the bottom-right corner of 5B. After finishing with the swap_from region, the thread will wait in the

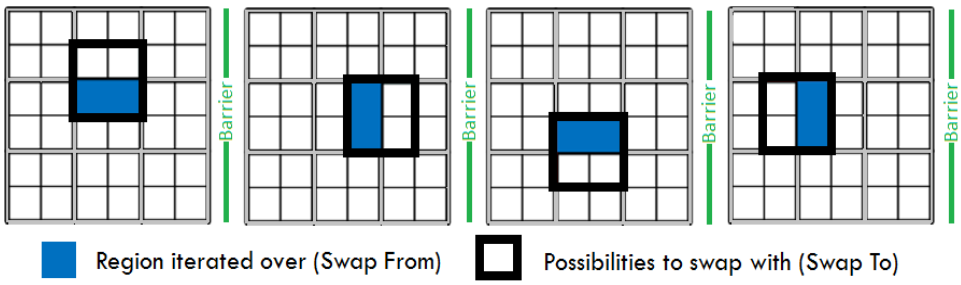


Figure 3.3: Half-box window decomposition's swap-from and swap-to region [2]

barrier until all threads have finished their respective region. Then, it will move onto the next iteration with the swap_from region being 5B & 5D and the swap_to region being 5B, 5D, 6A & 6C. The region restriction will repeat according to Figure 3.3.

Regardless of which method is chosen, data update is only performed after each barrier. This means that during the period prior to reaching the barrier, each thread only updates placement data for blocks residing within its swap_to region in its own private data. This implies that placement data about all other threads will grow stale until the next barrier. In another words, from the perspective of each thread, all blocks outside of its swap_to region will not move until a data update is performed. At the end of each region placement, all threads broadcast their placement updates, giving a single consistent view of all placements across all threads. Using this synchronization as a new starting point, all threads proceed to the next sub-region and so on. This allows each thread to work in its private region independently and to make movement decisions deterministically without costly fine-grain synchronization. However, during placement, decisions may be made based upon stale data. The effect of stale data is shown in Section 5.3.2.

3.4.2 Other Determinism Considerations

Calculations of timing, delay, and bounding-box costs are parallelized by dividing the netlist across multiple threads. Each thread returns a partial result which needs a final summation. These functions use floating-point values, which are not necessarily associative and can produce different results due to round-off if the order of addition changes. Therefore, in order to achieve a deterministic result, the order for the final addition must be enforced. This can be achieved by allocating an array with one entry for each thread. A master thread will then perform the summation in sequential order only after all worker threads have completed their partial sum, producing a deterministic sum. Since the work division scheme (more details in Section 3.5) depends only on the connectivity of the nets, which is constant for a given circuit, the worker threads will always be allocated a deterministic amount of work — this ensures partial results will also be the same. Thus, cost results are all deterministic.

Finally, to keep the program deterministic, each thread tracks its own random number state. Thus, random numbers generated in one thread do not disturb the sequence of those made by other threads.

While Ludwin [6] emphasizes serial equivalence for easier regression testing and customer support, we believe that an algorithm that is deterministic is sufficient. For our program, the algorithm is deterministic because T threads, where T is fixed, can be run on any number of processors (even more or less than T) and produce the same result.

3.5 Parallel Timing Analysis

Periodically, after a significant number of moves, the old timing analysis data becomes stale and must be regenerated. Hence, every 5 temperature iterations, the parallel placer perform a timing analysis to identify new critical paths. This is a time-consuming task that quickly becomes a bottleneck, so careful consideration must be taken to ensure that it is not invoked too frequently. Furthermore, parallelizing the existing algorithm is not straight forward. The function *timing_update_parallel()* on line 3 of the pseudocode in Figure 3.2 indicates when this task is invoked.

This function first calculates the *edge delay* for each connection in the circuit based on the current placement, and uses this to compute the *slack values*. The slack values are then used to calculate the *criticality values*, which are used in the calculation of *timing cost* and *delay* as described in Section 2.4. Barriers are used to enforce data dependencies between the aforementioned functions, as well as further enforce precedence in the calculations.

The parallelization of edge delay, timing and delay cost calculation is straight forward. As these functions loop through all the nets in the circuit, we could simply allocate an equal number of nets to each thread. Each thread stores its partial result in a pre-allocated location in an array, which is later summed by the master thread as described in Section 3.4. The summation of values is not parallelized, as it is a short operation which involves various data transfers; however, serialization is needed here to help achieve determinism.

The parallel slack and criticality computation requires more synchronization because it traverses the tree to perform breath-first modifications twice, once in

each direction. As there are no data dependencies within each horizontal level, work can only be distributed safely across all threads on a per-level basis.

A slight data structure change was made to assist with parallelizing the tree traversal process used in timing analysis as described in Section 2.4. In the original VPR code, only children of each node are stored, which is adequate for the calculation of $T_{required}$, since each node only traverses through its fanouts. However, the calculation of $T_{arrival}$ for each node requires the traversal through its fanins. In order to avoid saving the parents of each node and thereby minimizing the memory impact, VPR's implementation traverses the child nodes at each level and calculates the $T_{arrival}$ of each child by propagating forward the $T_{arrival}$ of the parent plus the edge delay. If the child node already has a $T_{arrival}$ value from a different parent, the larger value is stored. While this implementation works fine for a single-threaded program, as the child node can only be updated by one thread, this does not scale well to a multi-threaded implementation. Imagine the case where multiple parents are updating the $T_{arrival}$ of their commonly shared child: this creates a potential race condition that may lead to non-deterministic results. One possible solution is to use a synchronization mechanism such as locks to ensure that a child node will only be updated by one parent at a time. However, this is inefficient since nodes with multiple fanins are ubiquitous in a given circuit. Another solution is to add a pointer from each child to its parents, so that it can traverse through its own fanins to determine $T_{arrival}$. Hence, all parents of the child are considered in a sequential order by the same thread. This is parallelizable and ensures there are no data dependencies between nodes at the same level. It also load balances a bit better, since fan-ins are more balanced/limited than fan-outs. Therefore, the latter solution is implemented.

```
1: for inet = 1 to num_nets do
2:     ...
3:     for ipin = 1 to net[inet].num_sinks do
4:         ...
5:     end for
6: end for
```

Figure 3.4: Sample workload

In many cases, the amount of work depends on an attribute of the inner loop, which is not directly visible to the outer loop as shown in Figure 3.4. For example, loading the net delay matrix requires looping through all sinks of each net. Simply distributing an equal number of nets does not yield a good load balance, as some nets have only 1 sink, whereas some others may contain hundreds or even thousands of sinks. A dynamic workload distribution system is employed where each thread keeps track of the amount of work (ie, the number of sinks visited in this case) that is done, and returns that value when it has finished. The workload of two neighboring partitions is then compared, and the boundary is shifted to even out the workload. If the next region traverses through more sinks, then Equation 3.2 is used, otherwise, Equation 3.3 is used to calculate the number of nets shifted.

$$\Delta_{nets} = \frac{sinks_{next} - sinks_{this}}{sinks_{this}} * \frac{nets_{next}}{4} \tag{3.2}$$

$$\Delta_{nets} = \frac{sinks_{this} - sinks_{next}}{sinks_{next}} * \frac{nets_{this}}{4} \tag{3.3}$$

The first multiplicand indicates the relative percentage difference in workload between the two neighbors to control the number of nets being reallocated. The second multiplicand limits the number of nets that can be moved between regions to

25% of the entire region. This was added to compensate for the over-shifting which occurs often when the receiving region has much fewer sinks than the sending region.

The initial partition is simply an even distribution of nets; however, it will be dynamically rebalanced out as more calls to the function are done. During the first few runs of the workload distribution system, larger boundary movement is observed. However, as distribution begins to even out, the marginal improvement from each iteration gradually declines. Thus, only a limited number of dynamic distribution iterations will take place. Furthermore, the sequence of workload rebalancing is entirely deterministic and does not depend on timing or race conditions, only on the sink and net counts.

3.6 Parallel Move Evaluation

Each parallel worker thread must consider moves or swaps within its region (one sub-region at a time). The *try_swap()* function is executed by each thread with their respective local data or unchanged global data. Therefore, inter-thread interference is not possible in this function call. For each block that is considered, a randomly selected neighbor from the legal swap-to region is considered for a swap. In the case where the randomly selected spot is invalid, (eg, swap a CLB with an I/O pad) another neighbor is selected. This swap will be assessed based on the difference in timing cost and bounding-box cost, and evaluated using identical functions as implemented in VPR to mimic the simulated-annealing process.

3.7 Bounding Box Update

In the serial VPR program, bounding-box update is done incrementally as part of the swap cost evaluation function. However, in the parallel program, each thread calculates its incremental bounding-box cost based on stale data which contains imprecise data about the blocks that are not located within its private region. Therefore, a fresh bounding-box calculation using non-stale data is needed at the end of every iteration to ensure correctness and determinism. This is done at line 15 in Figure 3.2). This is extra work that the serial VPR program does not need to execute. The bounding-box calculation is parallelized using the same dynamic distribution scheme described in Section 3.5, and the final result calculation employs the method described in Section 3.4 to ensure determinism.

3.8 Summary

The parallel algorithm is based on the simulated-annealing algorithm in VPR with the following modifications:

- Dividing the entire grid into approximately equal sized private regions and assigning each region to a thread.
- Two different region decomposition methods to restrict the blocks being considered for swaps, thereby avoiding hard conflicts and enforcing determinism.
- Avoiding fine-grain synchronization by assessing moves using locally stored data, which may become stale but will periodically be refreshed.
- Parallelized timing analysis by storing the parents of each node to ensure

better load balance and determinism.

- Adding bounding-box update at the end of every temperature range to ensure correctness and determinism.

Chapter 4

Parallel Algorithm Tuning

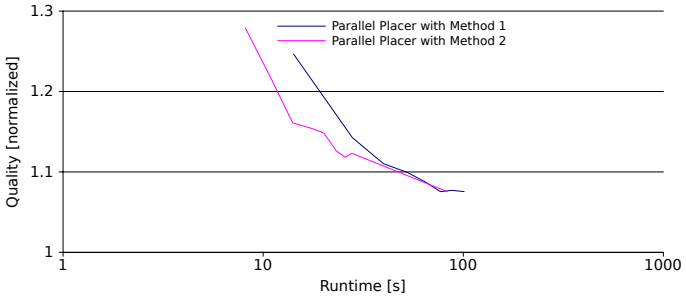
This chapter describes the tuning process for the various implementation choices and parameters needed to obtain the best QoR and best runtime.

Except where noted, data presented in this chapter are post-placement geometric means of all 7 circuits normalized against values obtained from default VPR 5.0.2 settings. Only post-placement metrics are used in this chapter since they can be generated quickly and correlate fairly well with post-routing metrics (as shown in Section 5.2).

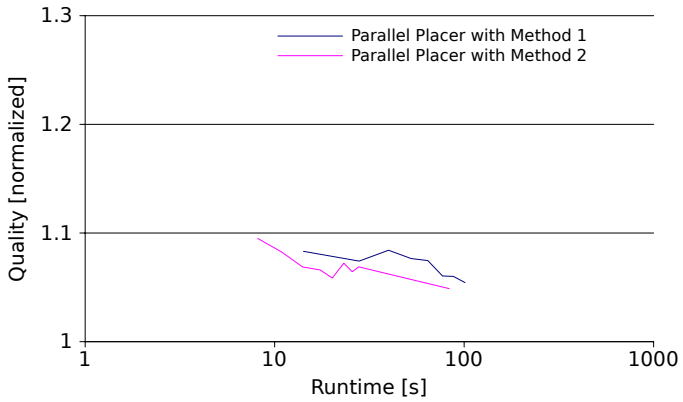
4.1 Swap Region

In Section 3.4.1, two different swap region methods are shown. The resulting QoR curve using 25 threads is evaluated here and presented in Figure 4.1. The HBWD method achieves better performance in both the bounding box and critical-path delay QoR curves. Therefore, it is employed for this thesis.

One reason the HBWD method achieves a better QoR is due to the amount of work accomplished between barriers. After four barrier executions in the HBWD



(a) Bounding box quality



(b) Critical path delay quality

Figure 4.1: Comparison between the two region decomposition methods

method, each of the sub-regions will have been considered twice. In comparison, the first method will consider each sub-region only once. The number of times both method enters the barrier is the same, thus the HBWD method effectively reduces the number of times the program enters a barrier by half.

The HBWD method also utilizes the grid better than the first method. In the first method, the active region, the region where blocks are being moved, at any given moment is the extended sub-region, which is approximately 25% of the entire grid. In the HBWD method, the active region spans the entire grid (100%). This

approach maximizes the size of the active region by fully utilizing the entire grid and eliminating blocks that would otherwise be idle. Another inherited benefit of the larger active region is that due to the larger swap-to region, blocks are able to swap with candidates located farther away, enhancing block migration.

4.2 Swap_To Block Choice Restriction

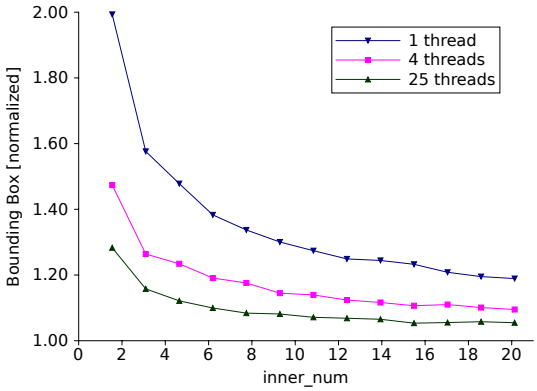
An algorithm was created by allowing each swap-from candidate to swap with any block residing within the swap-to region boundary. However, as shown in Figure 4.2(a) for 1, 4, and 25 threads, the quality varied wildly as the number of threads varied.

Adding a limit to the distance where swap-to blocks can be chosen from seems to alleviate the issue. Through experimentation, limiting all blocks to swap with candidate within a Manhattan distance of 10 produces the best QoR. This is effectively capping the R_{limit} value of 10 at the start of the program. The result is shown in Figure 4.2(b), and it is clear that the quality variation has improved dramatically compared to without the swap candidate constraint.

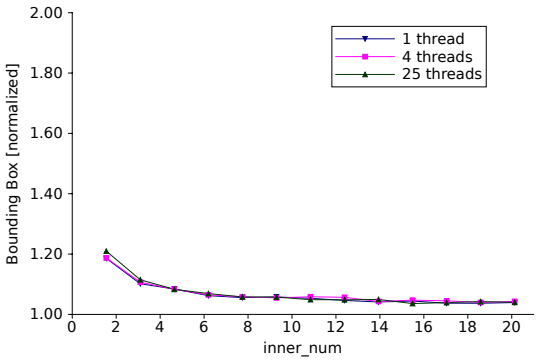
4.3 Sequential versus Random Block Selection

Unlike VPR, this algorithm sequentially iterates through each block position in the grid. To reach this conclusion, an experiment was designed with a controlled number of swaps per temperature range for both the sequential and random block selection schemes using both 1 thread and 25 threads. This experiment also incorporates the swap_to block choice restriction (Section 4.2) and `PROB_SKIPPED` value of 10 (Section 4.4). The result is shown in Figure 4.3.

It can be seen that the quality of bounding box is clearly better in the case of



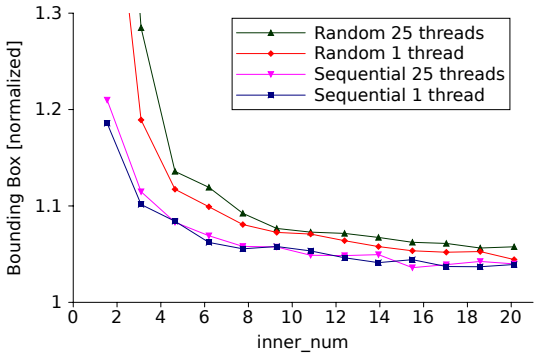
(a) with no restriction swap-to candidate selection



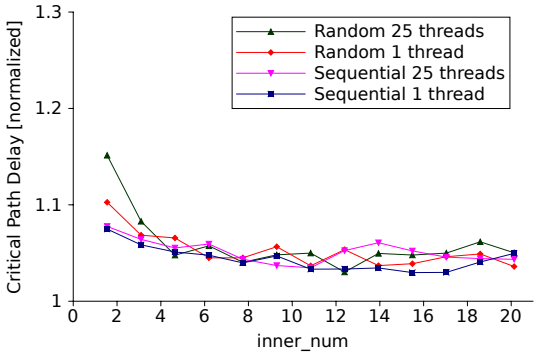
(b) with swap-to candidate limited to blocks within 10 Manhattan distance away from the swap-from candidate

Figure 4.2: Swap_to region quality variation

sequential block selection, for both 1 and 25 threads. In the critical-path delay graph, the quality is somewhat indistinguishable between the two block selection methods. Therefore, the sequential block selection scheme is used throughout this thesis as it performed no worse than the alternative method.



(a) Bounding Box quality comparison



(b) Critical Path Delay quality comparison

Figure 4.3: Sequential versus random block selection comparison

4.4 `PROB_SKIPPED` Characterization

The `PROB_SKIPPED` value adds an extra degree of freedom to tweak the program for quality and runtime trade-offs. It controls the inner most loop of the algorithm which makes it a standalone parameter that can be varied with only minimal disturbance to the rest of the parameters. It dictates the probability each block will *not* be considered for a swap when iterating through the swap_from region. A value of 100 is equivalent to not making any swaps, resulting in the initial placement. Figure 4.4 shows the QoR trade-off obtained by sweeping `PROB_SKIPPED`

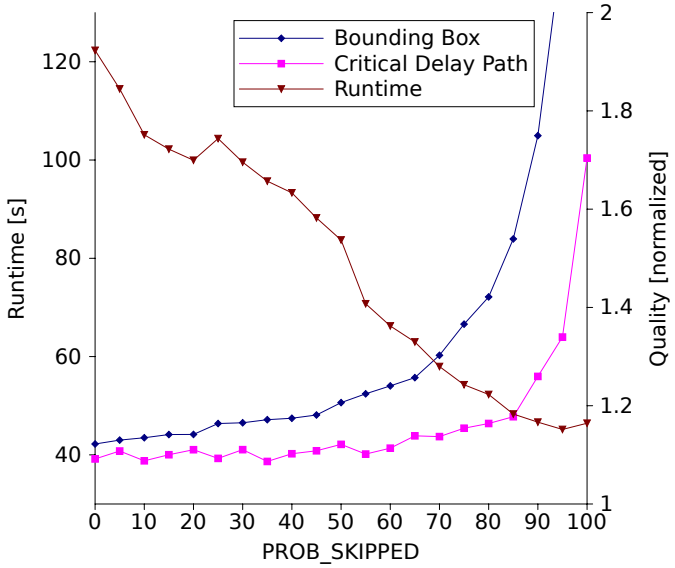


Figure 4.4: `PROB_SKIPPED` sweep

from 0 to 100 with 4 threads and an *inner_num* value of 4.6. The experiment uses the swap_to block restriction (Section 4.2) and the sequential block selection scheme (Section 4.3). The leftmost point (greatest runtime) is obtained with `PROB_SKIPPED` of 0, since every block is considered. As `PROB_SKIPPED` increases, fewer blocks are considered, resulting in shortened runtime and reduced quality.

The purpose of this experiment is to determine whether it is possible to avoid considering every block and still maintain a good result. It can be seen from Figure 4.4 that the quality of both metrics degrades rather slowly for small `PROB_SKIPPED` values. Therefore, a value of 10 is selected as it gives the best trade-offs — an approximately 14% runtime reduction for only approximately 1% quality loss.

4.5 Adaptive Annealing Schedule

This section describes the experiments conducted in an attempt to minimize the quality difference between varying number of threads. With the initial algorithm, the bounding-box quality mildly degrades by approximately 3% when scaling from 1 to 64 threads. While noise is may be one of the contributing factors, we believe improvements can be made here.

In able to better understand the potential causes that could result in different quality when a different number of threads are used, we first identify properties that change as the number of threads scales.

As the number of threads increases, the first property that changes is the size of the private regions. As the grid is constant in size, each thread ends up with a smaller proportion allocated for its private region. This in turn affects the average swap distance since blocks may only swap with other candidates within its private region boundary. Limited swap distance may potentially hinder block migration, which could potentially be the cause of the quality decay seen here.

The second property that changes with the number of threads is the amount of stale data, which has an effect on the swap decisions being made. Qualitatively examining this issue, the amount of stale data observed by each thread can be computed as $1 - 1/n$, where n is the number of threads being used. Thus, the largest stale data increase is from 1 thread (0%) to 4 threads (75%). However, as will be shown, the quality between 1 and 4 threads is relatively constant. Further scaling beyond 4 threads does not change the percentage of grid that is stale as dramatically. Also, the degree of staleness shrinks as the number of threads increase, since a block is restricted to move a distance no larger than the size of the swap-to region,

Table 4.1: Quality experiments

Experiment	Swap_from	Swap_to	*region_place_count*	PROB_SKIPPED
1	same as swap_to	same	divide by 2	same
2	same as swap_to	same	same	from 10 to 55
3	same as swap_to	same	multiply by 8	divide by 8

which is shrinking. Thus, we believe that stale data is not a primary contributing factor to the quality decay seen here.

4.5.1 Early Exploration

In this section, three experiments are conducted to investigate the effect of thread count on the quality. The parameter settings for each experiment are shown in Table 4.1. In all experiments, the premise behind using a larger swap_from region is to maximize the swap distance by setting the swap boundaries to the active region boundary. This change effectively doubles the number of blocks being considered at each inner loop. Thus, to keep the number of swaps constant, the *region_place_count* and PROB_SKIPPED values are adjusted accordingly in the first two experiments.

The third experiment is proposed with the intuition that in order to assist with block migration, more iterations are needed compared to the number of swaps per iteration. Each iteration, blocks can only move randomly within the constrained private region and cannot move beyond the boundary. Thus, we conduct an experiment using a large number of iterations to see if quality will improve, and hence whether the intuition is correct.

Results based on 7 circuits, normalized against VPR using *inner_num* value of 1, are shown in Figure 4.5. A constant *inner_num* value of 1.3 is used for the parallel placer. While experiments 1 and 2 show no improvement over the old

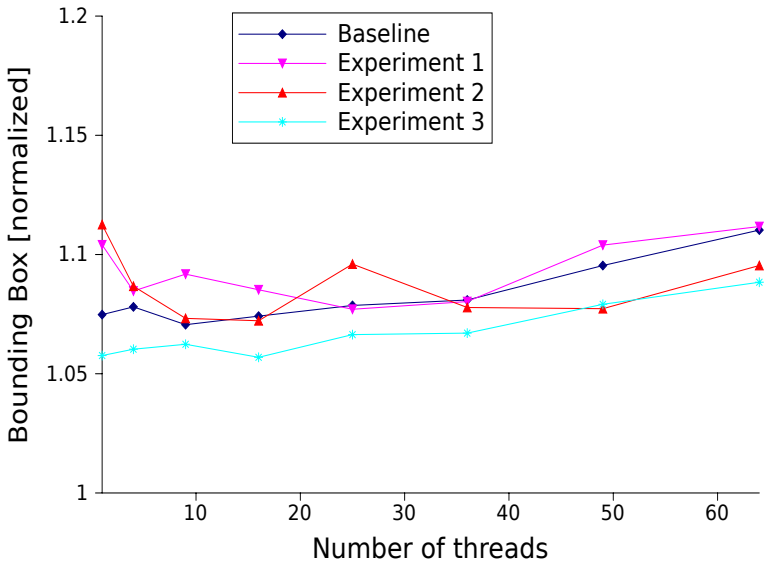


Figure 4.5: Quality with varying number of threads for various experiments

algorithm, experiment 3 seems to show improvements.

The first two experiments resulted in quality degradation at 1 thread. However, it is not clear if there is a subtle mechanism that could cause this or if it is simply noise. With more threads, these two experiments show similar results as the baseline algorithm. Although the second experiment may seem like it produces better quality at 49 and 64 threads, the result at 25 threads suggest this is simply noise.

The third experiment is convincingly better than the baseline algorithm. It is consistently about 1-2% better throughout the entire range of threads being used. While this experiment results in longer runtime since data broadcast and timing analysis are performed more frequently (between each iteration), it provides an important insight: it is not the number of swaps within the iteration that impacts quality, it is the number of iterations considered at each temperature that matters. That is, there must be ample ability for blocks to migrate across region boundaries.

While none of the methods presented in this section are implemented in the final algorithm, the insights gained are important.

In summary, the first two experiments show that increasing the swap_from region makes no significant effect to the overall quality. However, increasing the *region_place_count* results in a consistently better result. The important take away points from these experiments are 1) having a large number of swaps within a constrained region does not appreciably improve quality since blocks are simply moving randomly within the region; 2) increasing (or decreasing) the size of the swap_from region has little influence on quality; and 3) block migration between regions is important, thus increasing the number of iterations is an effective way to improve quality.

4.5.2 Cost versus Temperature Evaluation

With increasing the number of iterations in mind, an experiment was conducted to better align the QoR characteristics of the parallel placer with VPR. Specifically, the cost versus temperature curve for the stdev000 circuit using 4 threads was compared to the original VPR, both using an *inner_num* value of 1, to see if anything can potentially be modified. Both VPR and the parallel placer use the VPR cooling schedule as shown in Table 2.1. Figure 4.6 shows the cost versus temperature curve for VPR and the parallel placer. It can be seen that the two curves do not overlap each other well. This suggests the VPR's annealing schedule may not necessarily be the most suitable choice for the parallel placer.

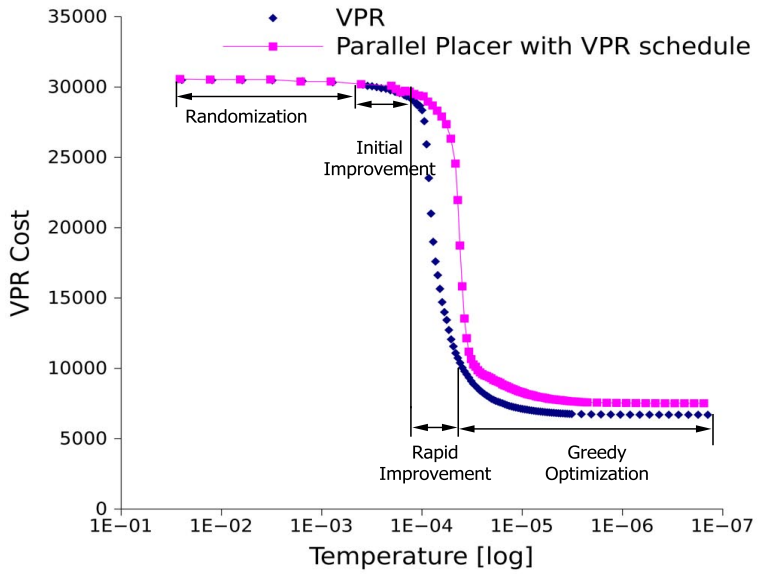


Figure 4.6: Initial cost versus temperature comparison

Modification 1

The first noticeable difference is that the transition from the randomization stage, where the cost remains very high, to the initial improvement stage occurs at a lower temperature in the parallel placer. The alpha value is 0.5 and 0.9 in the randomization and initial improvement stages, respectively. A smaller α value leads to faster temperature drop. Entering the initial improvement stage a single iteration later means valuable annealing steps needed for 'hill-climbing' are skipped. Therefore, to better align the parallel placer with VPR, *mod 1* is introduced: the randomization stage is limited to portions where the success rate is greater than 0.98 (versus 0.96 for VPR). The resulting curve from *mod 1* is shown in Figure 4.7; the resulting curve is a bit closer to the curve generated from VPR.

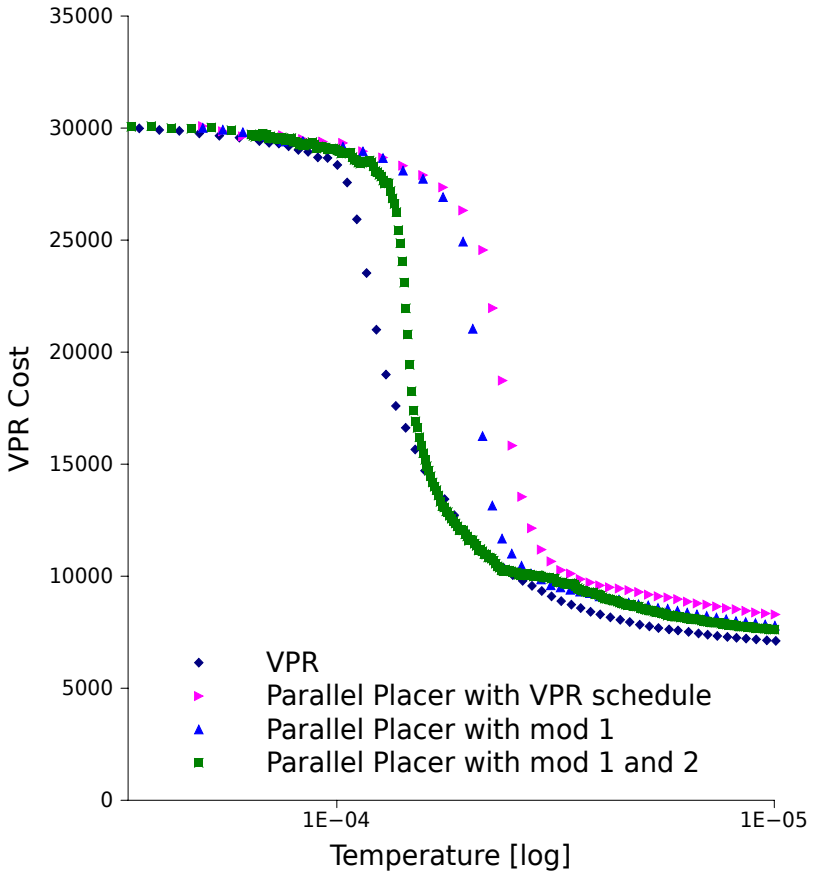


Figure 4.7: Cost versus temperature comparison with mod 1 & 2

Modification 2

The second noticeable difference is that during the rapid improvement phase, where the data points form a near-vertical line, the parallel placer has much fewer datapoints than VPR. This suggests that not enough time is spent during the cooling process, which may result in the final placement being trapped at local minima. This is a serious issue for the parallel placer since block movement is already constrained to the swap region boundaries; a quenching-like annealing schedule would

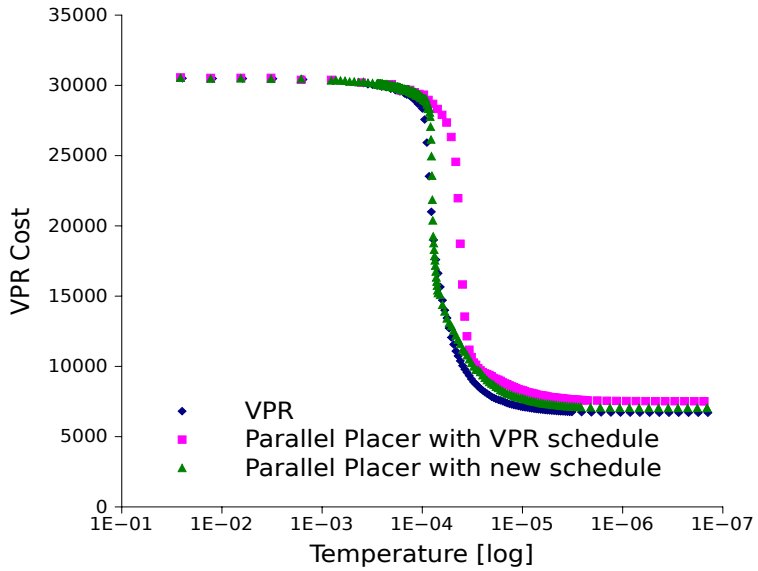


Figure 4.8: Cost versus temperature comparison with the new schedule

further inhibit block migration.

To mitigate this issue, finer annealing steps (a larger α value) are taken to ensure more iterations are considered at each temperature step. Modification *mod 2* includes a) adjusting the transition to the rapid improvement stage when the success rate falls below 0.94 (versus 0.80 for VPR) and b) increasing α to 0.99 (versus 0.95 for VPR) during the rapid improvement phase. These two changes become the 'Fine Rapid Improvement' phase in Table 4.2. The resulting curve is shown in Figure 4.7 with square markers. It can be seen that it is much closer to the original VPR curve with these changes. It also converges to a higher quality result than before these changes, but it is still not as good as VPR.

Modification 3

The final modification further targets the transition stage between randomization and the rapid improvement as there is still an apparent gap between the parallel placer and VPR. The α value is further increased from 0.99 to 0.995 to alleviate this issue. In addition, the slow annealing schedule (where $\alpha \geq 0.99$) is limited to the phase when R_{limit} spans the entire grid. Once R_{limit} begins to shrink, the original α value used in VPR is restored. This is done because the swap region sizes between the parallel algorithm and VPR begins to converge at similar rates as R_{limit} shrinks. In another words, as R_{limit} becomes the limiting factor for the swap region size, there is no longer a difference in the swap distances between the parallel placer and VPR. Hence, as the swaps considered in both programs are now similar, it is logical to restore the α value in VPR. These two changes become the 'Fine Initial Improvement' and 'Fine Rapid Improvement' phases in Table 4.2.

The entire new simulated-annealing schedule is shown in Table 4.2. For comparison, VPR's original schedule is also shown in this table.

The resulting cost versus temperature curve for the parallel placer using the improved annealing schedule is shown in Figure 4.8. The new parallel annealing schedule tracks VPR's curve much better, and achieves a better QoR than the original schedule.

Quality Scaling Comparison with Improved Schedule

With the improved simulated-annealing schedule, the quality versus number of threads curve compared to VPR is shown in Figure 4.9. In comparison to Figure 4.5, the upward quality degradation trend is no longer present, and quality variation due to varying the number of threads has been minimized.

Table 4.2: New simulated-annealing schedule

Phase	New schedule			VPR schedule		
	R_{accept}	R_{limit}	α	R_{accept}	R_{limit}	α
Randomization	> 0.98	–	0.5	> 0.96	–	0.5
Initial Improvement	> 0.94	–	0.9	> 0.80	–	0.9
Fine Initial Improvement	> 0.83	–	0.995	—		
Fine Rapid Improvement	> 0.15	== entire grid	0.99	—		
Rapid Improvement	> 0.15 or > 1		0.95	> 0.15 or > 1		0.95
Greedy Optimization	otherwise		0.8	otherwise		0.8

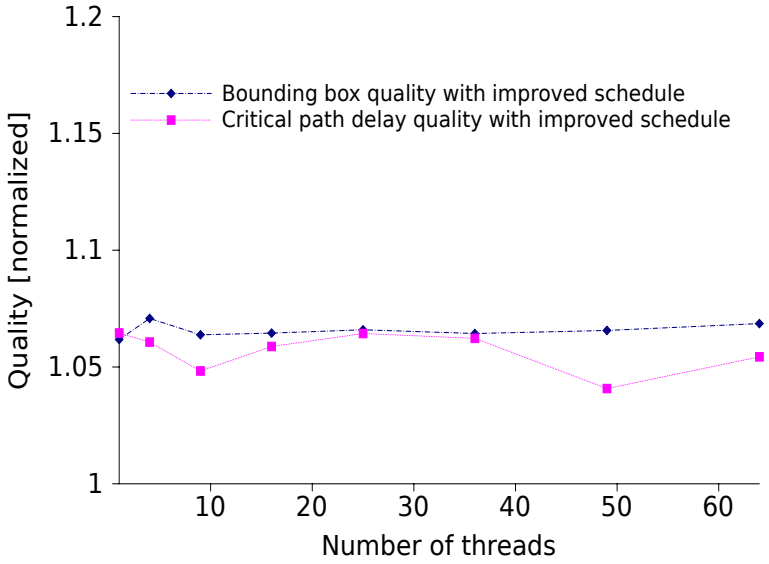


Figure 4.9: Quality scaling

QoR Curve

The QoR curve with the new simulated-annealing schedule for 25 threads is shown in Figure 4.10. Although the finer annealing steps result in more swaps taken and thus longer runtime for a given *inner_num* value, the overall QoR curve turns out to be better. This suggests that the improvement in quality outweighs the extra runtime, thereby shifting the QoR curve below the original one.

In summary, we did not explicitly increase the number of iterations at each tem-

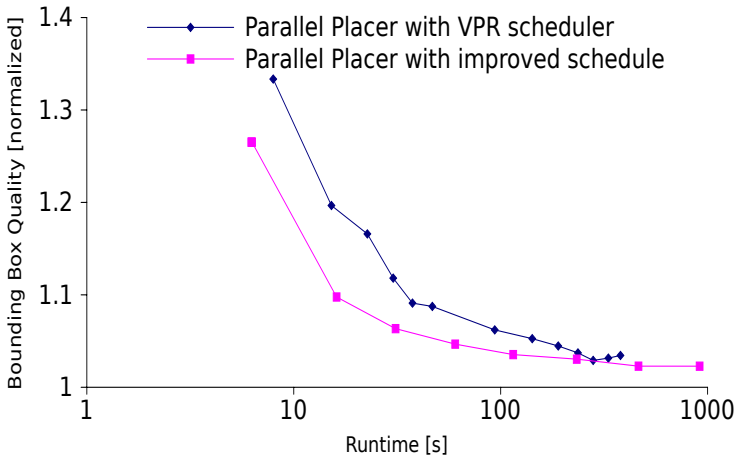


Figure 4.10: QoR improvement due to the new schedule

perature stage. Instead, we achieved a similar effect by slowing down the anneal process, which effectively increases the number of temperature stages, and hence the number of iterations. As a result, the new annealing schedule makes two improvements: a) eliminates the upward trend in quality degradation as more threads are used and b) improves the QoR curve.

4.6 Parallel Programming Optimizations

4.6.1 Parallel Memory Allocation

This work makes use of the Hoard memory allocator [40], which is a fast, scalable and memory-efficient memory allocator. This memory allocator addresses three main issues:

- Contention due to multi-threaded programs sharing the same heap. This is especially a problem when allocate functions are executed simultaneously by multiple threads. A conventional memory allocator would simply serialize

Table 4.3: Hoard library runtime comparison

Circuit	Without Hoard[s]	With Hoard[s]	Runtime Reduction
stdev000	51.89	37.23	28%
stdev002	51.94	38.55	26%
stdev004	51.59	36.12	30%
stdev006	50.23	36.12	28%
stdev008	44.68	32.38	28%
stdev010	56.48	39.81	30%
stdev012	61.08	44.81	27%
		average:	28%

these allocate functions.

- False sharing due to multiple threads sharing the same cache line to store its distinct variable. Frequent writes by other threads cause unnecessary invalidate requests from the CPU, requiring the data to be re-read from memory, which dramatically slows down memory accesses.

- Fragmentation due to inefficient utilization of freed memory. This is not a major issue for this work since memory de-allocation does not occur until the conclusion of the program. This would more beneficial for a program that contains significant dynamic memory re-allocation.

An experiment was conducted to experimentally determine the effectiveness of Hoard on the parallel placement program. Identical source code was compiled twice, once linking the Hoard library and once without the library. The average runtime for all 7 circuits is computed in Table 4.3. Hoard improves runtime by 28% on average.

```
1 struct aligned_mutex{
    pthread_mutex_t mutex;
3 } __attribute__ ((aligned(64));
```

Figure 4.11: Macro used to align data to cache lines

4.6.2 False Sharing

In addition to using Hoard to alleviate false sharing effects, many of the memory arrays and matrices are created with cache line size taken into consideration. Compiler macros ensure data structures are aligned to the cache line sizes of 64 bytes for the machines used in this work. This manual optimization is likely not needed in the presence of Hoard, however, it is left in the code in the event that the code is ported to a platform where Hoard is not available. A sample macro used is shown in Figure 4.11.

4.6.3 Processor Affinity

In a multi-threaded program, the operating system scheduler has the responsibility to assign processor(s) to execute each thread. On a SMP machine where all processors are identical, a thread could potentially be assigned to any of the processors by default. Processor affinity is an additional constraint which restricts the processors a given thread can be scheduled on.

For the parallel program described in this thesis, the ideal platform should have at least an equal number of processors as the number of threads. Therefore, each thread should be assigned uniquely to a processor without any time-sharing. This has a few benefits:

Processor Caching: the main issue with threads hopping from one physical processor to another is the contents of each processor's cache. Consider the

```
1  core_affinity = pow(2, input->id);
   pthread_setaffinity_np(pthread_self(), sizeof(core_affinity), &core_affinity)
```

Figure 4.12: Core affinity source code

example when a thread moves from being executed on processor A to processor B. Processor A's cache is warm, meaning it is full of useful memory contents. However, as caches do not move with the thread, the cache content is not transferred to Processor B when the thread moves. As a result, the thread will experience numerous cold (aka compulsory) cache misses as it begins to execute on processor B. To eliminate this unnecessary flushing of cache contents, it's best to lock threads to processors, so the cache stays warm at all times. Table 4.4 shows the runtime for a 25-threaded program run 10 times with and without processor affinity. It can be seen that program runtime reduces by approximately three quarters of a second when processor affinity is turned on, which contributes to about 2% in runtime savings.

Predictable Runtime: without setting processor affinity, the program runtime varies from run to run. From Table 4.4, it can be seen that the standard deviation is significantly lowered from 0.69s to 0.28s having processor affinity turned on.

The code shown in Figure 4.12 indicates how processor affinity is assigned, each thread numbered 0 through $n-1$ is matched on a one-to-one basis to a distinct processor 0 through $n-1$.

Table 4.4: Runtime comparison with and without processor affinity

Circuit	Runtime without affinity[s]				Runtime with affinity[s]			
	avg.	max	min	std dev.	avg.	max	min	std dev.
stdev000	38.26	37.12	40.35	0.93	37.12	36.82	37.53	0.26
stdev002	39.38	38.54	41.22	0.77	38.81	38.24	39.53	0.39
stdev004	37.24	36.65	38.00	0.36	36.67	36.20	37.01	0.25
stdev006	36.93	35.95	39.72	1.03	36.11	35.50	36.50	0.35
stdev008	33.45	32.70	34.15	0.51	32.76	32.52	33.20	0.19
stdev010	40.39	39.63	41.00	0.42	39.88	39.51	40.22	0.22
stdev012	46.00	44.88	49.49	1.31	45.11	44.57	45.68	0.32
average	38.65			0.69	37.90			0.28

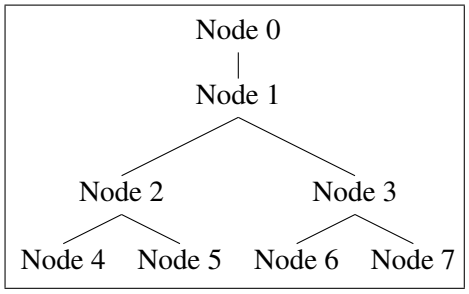


Figure 4.13: Sample custom polling barrier tree with 8 nodes

4.6.4 Custom Polling Barrier Implementation

While barriers in the Pthreads library are functionally correct, they are too slow. The POSIX barriers suspend and resume the thread as it arrives at the barrier, which introduces process scheduling overheads. The overhead is not justifiable for the purpose of this parallel program since most threads have approximately equal amount work and hence similar execution time. Instead, a custom tree-based polling barrier is implemented using shared global memory based on [41]. Figure 4.13 shows a sample barrier for an 8-thread program.

Nodes are numbered from left to right and top to bottom. For a given program with n threads, there'll be a total of n nodes in the barrier tree. The barrier tree

almost forms a binary tree, with the exception of node 0 and 1. This is done to make it mathematically simple to calculate the children of each node without needing a function to convert node id from thread id; the children of each node are simply $node \cdot 2$ and $node \cdot 2 + 1$. As each thread arrives at the barrier, it'll first poll until all of its children have arrived, if it has any, before writing to a shared memory to signal its arrival to its parent. Then it'll wait for the *release* signal from its parent to exit out of the barrier. This is very efficient since only two memory transfers are needed for each node, one for arrival and one for release, for an entire barrier cycle, with the assumption that there is no false sharing amongst different nodes. This custom barrier performed much better than the POSIX barriers. Figure 4.14 shows the average runtime comparison over the seven circuits running with 4 to 25 threads. It can be seen while runtime is about the same for four threads, the difference grows dramatically as 25 threads are used. With 25 threads, the runtime obtained using Pthreads barrier more than doubles the runtime with the custom polling barrier. The custom barrier provides better runtime and scales better.

In the event that the number of threads are less than the number of partitions, where the processors need to be time-shared, the native Pthreads barrier implementation outperforms the custom barrier. The reason is that in the custom barrier, each thread is constantly doing work, including the time that it is stalled waiting at a barrier. This makes it difficult for other threads to get work done, since spinning threads do not yield to threads that need to perform real work. In this case, the Pthreads barriers should be used instead.

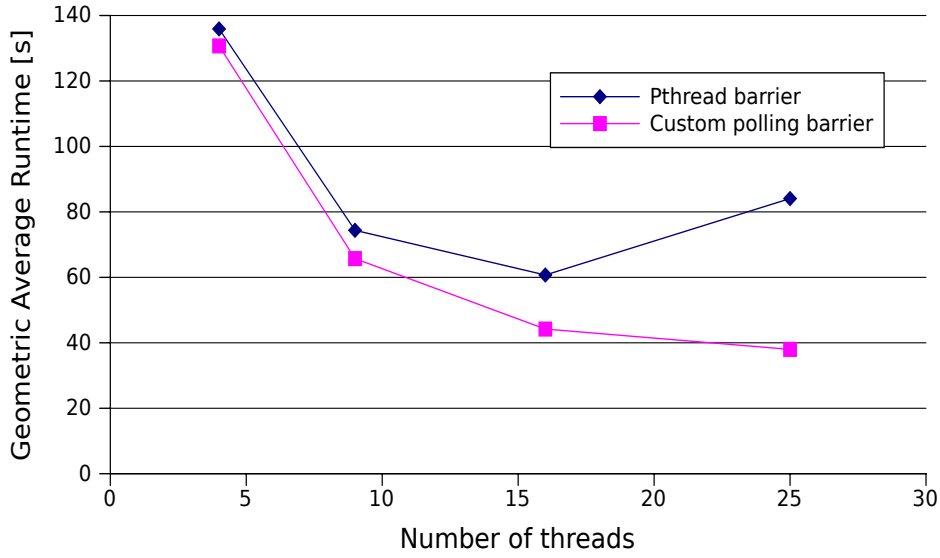


Figure 4.14: Custom polling versus Pthreads barrier

4.7 Closing the Quality Gap

This section describes experiments conducted in an attempt to minimize the quality difference between VPR and the parallel placer. The goal here is to investigate whether it is possible to achieve the same quality obtained by VPR using the parallel algorithm. All datapoints obtained in this section are with an *inner_num* value of 10 to obtain the asymptotic quality.

While none of the techniques here are used in the final parallel placement algorithm, it is documented here for the interest of the readers.

4.7.1 Forced Block Migration

To encourage blocks to migrate outside of the swap_from region, this experiment sometimes constrains the swap-to position to be selected outside of the swap-from region (refer to Figure 3.3). This experiment is designed with the intuition that the

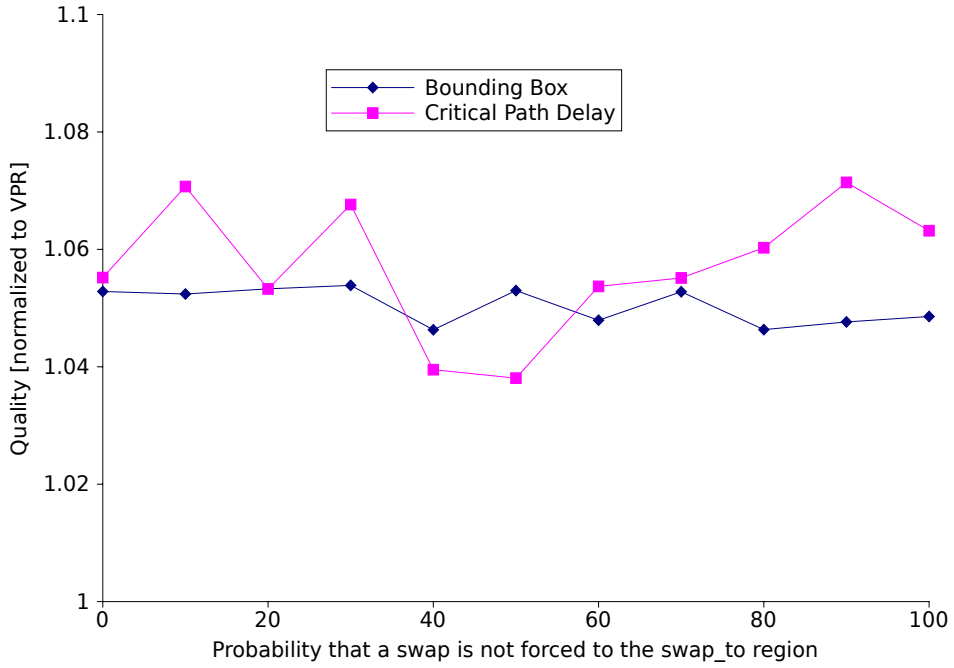


Figure 4.15: Forced block migration

block migration issue could be caused by a) choosing swap neighbors too close and b) being selected again due to the sequential block selection scheme. Thus, artificially forcing the block to move outside of the swap_from region may encourage block migration.

Results are shown in Figure 4.15. We sweep the probability a given block is forced to be outside the swap_from region. A value of 0 implies all swap-to candidates are outside the swap_from region and a value of 100 means no swap is being constrained, thus being equivalent to the original algorithm. In summary, forced block migration does not seem to have an influence on the QoR.

4.7.2 Reject Good Moves

In this experiment, we intentionally reject good moves that decrease the overall cost metric. The idea behind this experiment is that we suspect insufficient 'hill-climbing' is done, thus the result is being trapped in local minima. Since it does not make sense to do this across all temperature ranges, as it would take an extremely long time to converge. We set the move rejection rate equal to the acceptance rate (R_{accept}), so it would scale down as annealing progresses. We also restrict this experiment to the period where the R_{accept} is between 0.98 and 0.50, so it does not interfere with the decisions during the low temperature range. With this rejection rate, the `PROB_SKIPPED` value is varied from 10 to 90 to evaluate its effect on QoR.

This also leads to a significant runtime increase since good swaps must be abandoned and another swap must be re-proposed and the cost metrics need re-calculated. To make things worse, if the recalculated move is good, there's the same probability that it will be rejected as well! Nevertheless, the goal here is to see if rejecting good moves has any effect on quality.

The results are shown in Figure 4.16. The horizontal axis indicates the value of `PROB_SKIPPED`: a high value indicates large number of blocks had been skipped and a value of 10 is equivalent to the original algorithm where only the rejection rate changes at the high temperature. As seen, the QoR seems to degrade slightly as `PROB_SKIPPED` is increased. This makes sense since fewer blocks are considered for swaps. In summary, the result does not suggest that QoR is influenced by the percentage of good moves that are accepted at the high temperature phase regardless of what the `PROB_SKIPPED` value is.

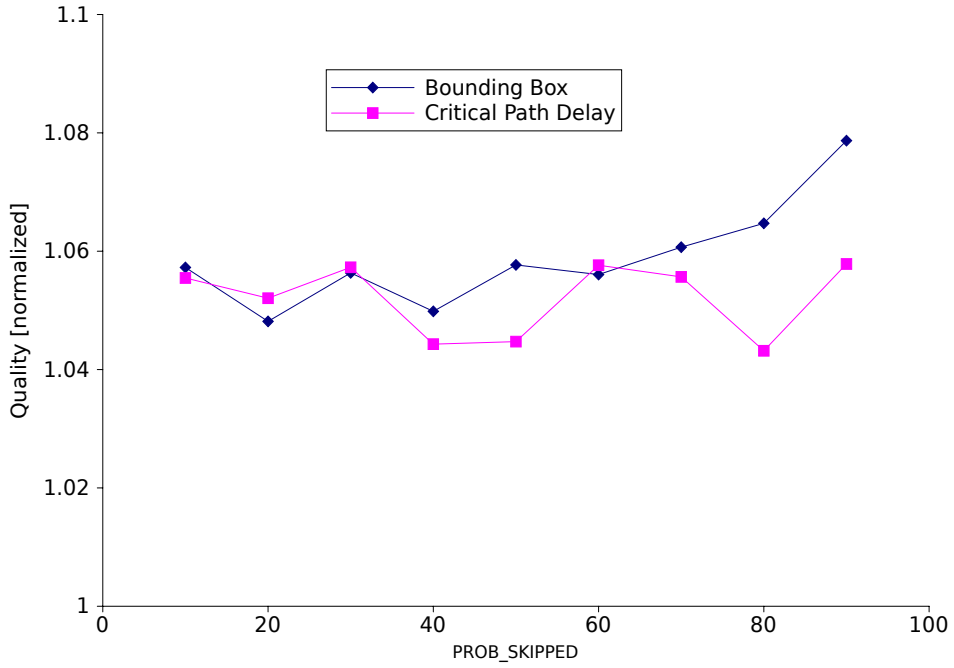


Figure 4.16: Reject good moves

4.7.3 Hybrid Parallel & Serial Placement

We investigate the effect of alternating between a serial and parallel version VPR placement. It is known that the serial placer yields superior quality than the parallel placer, and it is believed that the non-restricted swap region which allows freedom for distant block migrations is likely the cause of this. The idea behind this experiment is to see whether it is possible to combine the serial and parallel placer, thus allowing unconstrained swaps take place during every temperature range when the serial placer is executing. This is achieved by appending a serial VPR annealing step at the end of every temperature stage.

We start by allowing only 1 swap per temperature during the serial phase stage and incrementally increasing it, as shown in Figure 4.17.

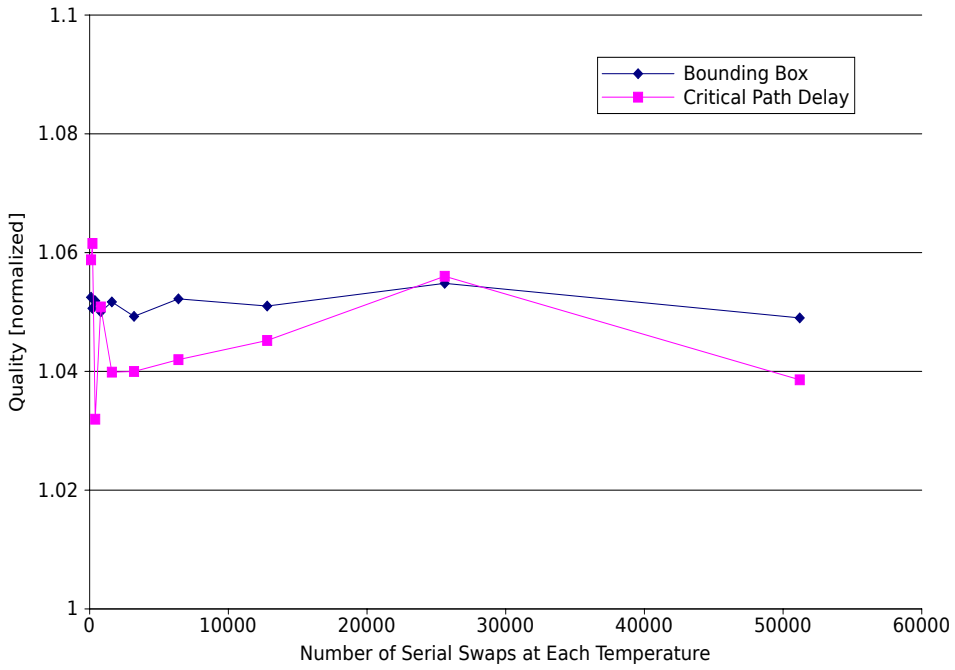


Figure 4.17: Serial and parallel hybrid placement

An issue here is that the unconstrained and constrained swaps may not actually work well with each other. For example, if one block makes a distant move during the serial phase, it could take the rest of the blocks which connect to it quite some time to move close to it during the constrained parallel swap phases. This apparent 'block-chasing' may lead to some quality degradation.

In summary, this hybrid placement scheme did not have an influence on the QoR. Even when the total number of serial swaps exceeded the conventional VPR, the hybrid placer was unable to achieve the QoR of the conventional VPR.

4.8 Summary

This section summarizes the algorithmic parameters and methods tuned to produce the final parallel placement algorithm with the best QoR.

- Utilizing the HBWD method for region decomposition to achieve determinism and good quality.
- Limiting the range of swaps to all blocks to a Manhattan distance of 10, or the swap region boundaries, whichever is smaller.
- Sequentially iterating through the grid instead of randomly selecting blocks to be swapped.
- To improve runtime for negligible quality loss, the algorithm only selects 90% of the blocks to swap.
- The new simulated-annealing schedule makes finer steps to better align cost versus temperature between the parallel placer and VPR.

Chapter 5

Experimental Evaluation

In this section, the benchmarking methodology is presented. Then the quality and runtime between the parallel placement algorithm and VPR is compared. Also presented is the self-speedup of the parallel algorithm versus a single-thread implementation of the same parallel algorithm and discussion regarding its runtime scaling limitations. Then, it is shown that the quality of placed circuits is preserved as the number of threads scales.

All quality values are geometric means of values which are normalized against VPR 5.0.2 run with default parameters. For example, a value of 1.06 implies 6% quality degradation while 0.97 corresponds to a 3% quality improvement.

5.1 Benchmarking Methodology

This section describes the circuits used for benchmarking and the experimental process.

5.1.1 Benchmarking Circuits

The traditional Toronto20 MCNC benchmark circuits are too small to use for parallel placement experiments. Furthermore, large FPGA circuits are rare. Therefore, the benchmarks provided with the Un/DoPack flow which are fully described in [42] are used here. These large synthetic circuits are built using the GNL tool in hierarchical mode, using 20 subcircuits which are each based on the Toronto20 MCNC circuits. For each of 7 synthetic circuits generated, the overall average Rent exponent is 0.62, but the Rent exponent in the inner subcircuits is varied to produce a standard deviation in Rent values from 0.00 to 0.12 in 0.02 increments. As a result, the synthetic circuit with the smallest standard deviation is easiest to route and has the most uniform packing of interconnect wires, while the largest standard deviation is hardest to route due to hotspots which need a much wider channel to route. Using GNL in this way is a bit better than simply stitching the 20 MCNC circuits at the I/O pins; the latter approach is more likely to have large independent subcircuits which are more amenable to parallel placement. The circuits are clustered using T-VPack 5.0.2 with 6-input lookup tables, a cluster size of 10 or 4, and a maximum of 35 or 15 inputs per cluster.

5.1.2 Hardware Environment

We evaluate the performance of our program using a Dell R815 machine. It contains 4 sockets, each with an 8-core AMD Opteron 6128 processor, to support a total of 32 processors, running at 2.0 GHz. The system has 32GB of memory and the operating system is Ubuntu Server 10.10. All timing measurements are for placement-related operations; both parallel and serial placer timing measurements are performed on this machine. The machine is load free with the exception of the

OS running in the background.

5.1.3 Experimental Methodology

The parallel placement work is compared against VPR running with the flag '-place_only'. There is a '-fast' flag in VPR that is faster (9.7x) than the default and achieves only 2.4% loss in bounding-box metric and 1.2% loss in critical-path delay metric on average for post-placement results and 2% and 0% for post-routing results using the benchmark circuits. The quality values in this Chapter are compared against default VPR and not VPR '-fast'.

The runtimes reported in this chapter include placement time only, which is primarily the pseudocode shown in Section 3.3. Time excluded from measured time, including netlist loading and pre-computation of delay tables used for cost calculations, is identical between serial VPR and our parallel placer, and is not part of the runtime measurement. The excluded portions can be runtime-optimized and/or parallelized quite easily.

We ran all of the circuits through the parallel placer first and obtained a quality/runtime trade-off curve by varying the *inner_num* values and the number of threads. Then each placed circuit is fed into the VPR router to obtain a minimum channel width with the flags shown in Figure 5.1. Then, 20% extra channels are added to the previously obtained minimum channel width to result in a low-stress routing solution which determines the critical-path delay of the circuit using the flags shown in Figure 5.2. This method is used for all circuits regardless of whether it is placed with the parallel or the original VPR algorithm.

-max_route_iterations 100 -pres_fac_mult 1.3

Figure 5.1: VPR options for minimum channel width

-route_chan_width ($min\_chan\_width \cdot 1.2$) -max_router_iterations 300 -pres_fac_mult 1.1

Figure 5.2: VPR options for critical-path delay

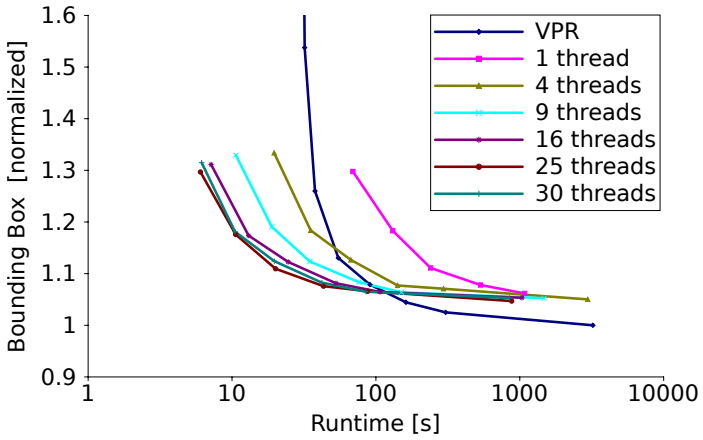
5.2 Quality versus Runtime

In the following section, the post-placement (PP) and post-routing (PR) metrics are shown. Post-routing results are ultimately the figures used to evaluate the result, however, as they are time-consuming to generate, much of the earlier algorithm exploration has been done using only post-placement metrics. It can be seen that the two curves track each other quite well, with PP bounding box correlating with PR wirelength and critical-path delays correlating with each other. Thus the early algorithm explorations done in Chapter 4 using PP metrics are valid.

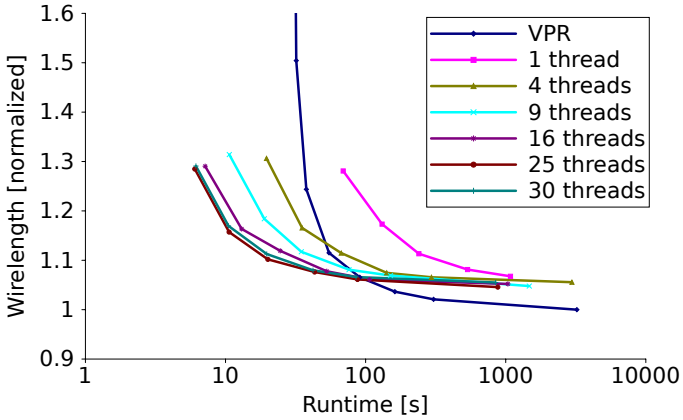
Figures 5.3, 5.4, and 5.5 show the quality versus runtime comparison for our parallel program and VPR. The runtime is plotted on the x-axis in a log scale and the normalized quality result is shown on the y-axis. The datapoints for VPR are obtained by starting with default VPR (slowest runtime, quality = 1.0). We then improve VPR runtime, first by adding '-fast' flag, then by further decreasing *inner_num* to values less than 1.

5.2.1 Comparison Trends

Figure 5.3 shows the runtime versus quality comparison for PP bounding box and PR wirelength metric. The dark blue line is VPR's runtime curve, and the rest of the curves are generated using the parallel algorithm with a varied number of threads.



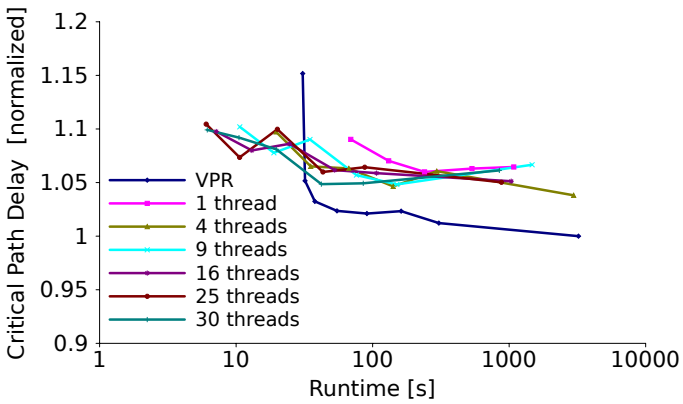
(a) Bounding Box PP



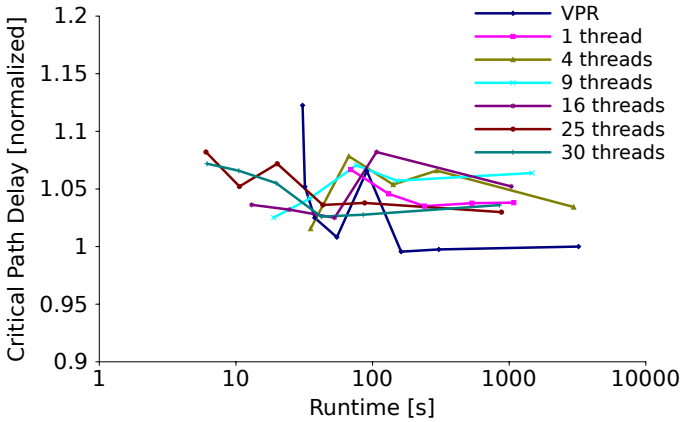
(b) Wirelength PR

Figure 5.3: Runtime versus quality of PP bounding box and PR wirelength

It can be seen that the single-threaded version of the parallel algorithm performs worse than VPR, since it is always above and to the right of the VPR curve. This is mainly due to the associated overheads needed to make the algorithm parallelizable. As more threads are used, the runtime of the parallel placer decreases while the quality stays constant. This is seen by the horizontal left shift of the



(a) Critical Path Delay PP



(b) Critical Path Delay PR

Figure 5.4: Runtime versus quality of PP and PR critical-path delay

curve. With just four threads, the parallel placer is able to outperform the serial VPR in the short runtime ($<$ 100s) region, but it includes mild quality degradation (about 20%).

Figure 5.4 shows the runtime versus quality comparison for the critical-path delay metric. While the magnitude of the critical-path delay degradation is smaller compared to the PR wirelength graph shown previously, the same trend is seen: the

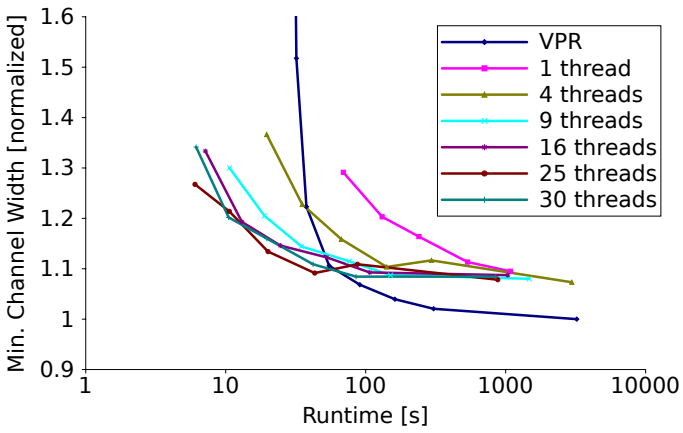


Figure 5.5: Runtime versus quality of PR minimum routable channel width

quality obtained by VPR degrades sharply at approximately 100s; in contrast, the parallel placer is still able to sustain a good QoR as runtime shrinks, outperforming VPR. A similar trend is seen with the minimum routable channel width curve as well, shown in Figure 5.5.

It can also be noted that although the QoR for the parallel program converges to an asymptotic bound as runtime increases, it is unable to match the quality of VPR. The PR-QoR gap of the best parallel placer quality compared to VPR is approximately 7.6%, 4.6%, 1.6% for minimum routable channel width, wirelength and critical-path delay respectively. We look into a different way of performing FPGA placement in Section 6.1, and show a promising technique that may be used to recover some of this quality loss.

5.2.2 Comparison with 25 Threads

To compare VPR and the parallel algorithm at a single data point, we need a strategy for selecting a point of comparison. Moving right-to-left along the quality/runtime curves, we selected the first point where our parallel version with 25 threads

Table 5.1: Parallel (25 threads) compared to VPR

	# LUTs	# CLBs	min. chan width	wirelength	crit. delay	speedup
stdev000	40013	5036	1.12	1.10	1.14	150
stdev002	40013	5051	1.14	1.09	1.13	153
stdev004	40013	5037	1.13	1.17	1.06	201
stdev006	40013	5023	1.13	1.09	1.02	168
stdev008	40013	5041	1.14	1.11	1.03	167
stdev010	40013	5043	1.15	1.09	1.09	151
stdev012	40013	5060	1.13	1.07	1.04	141
25 threads	Geo. Mean		1.13	1.10	1.07	161
VPR-superfast	Geo. Mean		1.96	2.09	1.12	105

outperforms VPR in *all* of the PP and PR quality metrics; this occurs around 30 seconds. We describe this point along the VPR curve as '-superfast', obtained by setting the *inner_num* to 0.015625. In comparison, at this point our parallel version with 25 threads uses an *inner_num* value of 0.25 with runtime of approximately 20 seconds.

At this selected point in the runtime/quality space, the parallel placer beats VPR '-superfast' in both runtime and quality metrics. Table 5.1 shows the actual quality and speedups obtained for each benchmark, as well as the geometric mean. In comparison, VPR '-superfast' is slightly outmatched in critical-path delay (1.12 vs. 1.07), but already vastly outmatched in minimum routable channel width (1.96 vs. 1.13), wirelength (2.09 vs. 1.10) and speedup (105x vs. 161x) by the parallel version.

A more general quality versus speedup trade-off comparison is plotted in Figure 5.6. It can be seen that VPR's speedup plateaus at approximately 100X with quality degrading more than 100% in wirelength and minimum routable channel width. In contrast, our parallel algorithm can run as fast as 6s on average while only sacrificing less than 30% on all PR quality metrics against VPR. This is equivalent

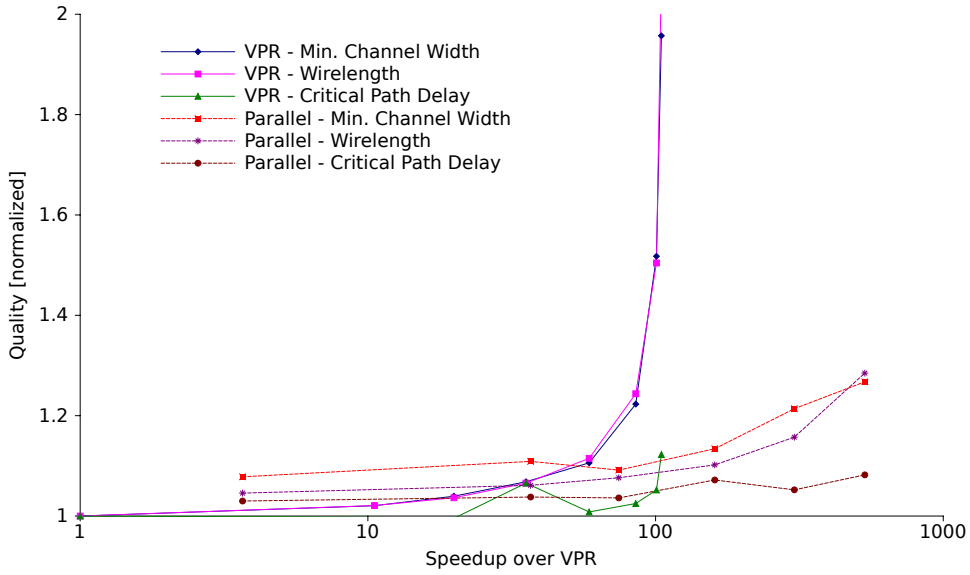


Figure 5.6: Quality from speeding up VPR and parallel (25 threads)

to a speedup of 500X.

5.3 Scalability

5.3.1 Runtime Scaling

Figure 5.7 shows the self-speedup obtained using up to 30 threads relative to the single-threaded version of the parallel algorithm with *inner_num* value of 1. The program scales well up to 25 threads, beyond which the speedup begins to plateau.

We attribute the scalable nature of our algorithm to the highly parallelizable inner loop. The inner loop, which iterates through the CLBs within each thread's private region, operates independently from all other threads with minimal inter-processor interference (eg, false sharing) and little inter-thread communication (broadcast updates at each barrier). This enables good scaling for a well load-

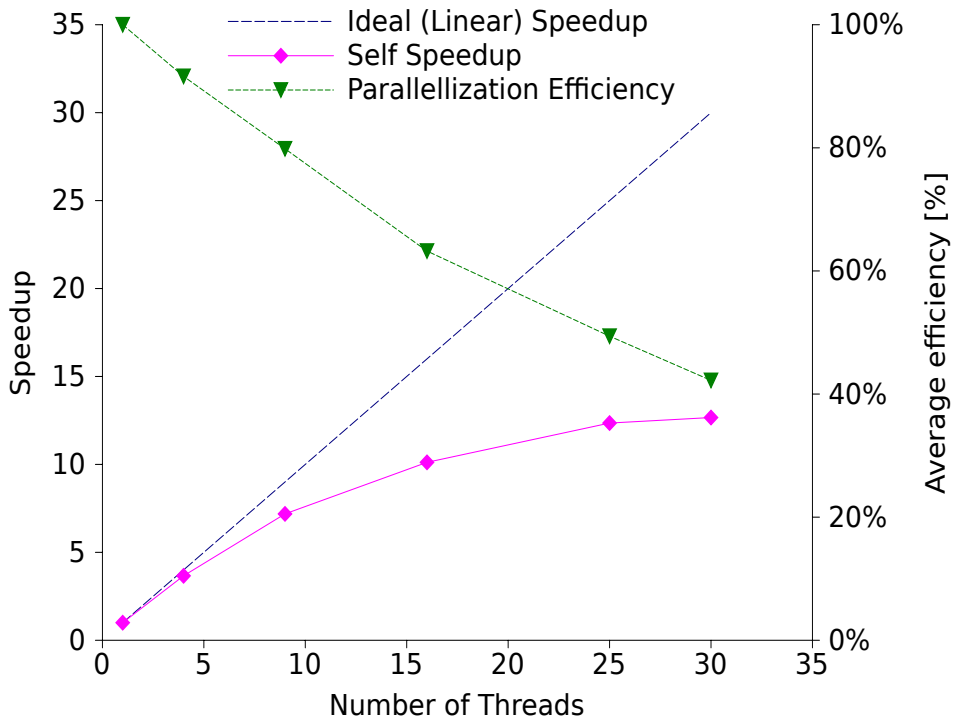


Figure 5.7: Self speedup

balanced workload distribution, since all threads will progress at similar rates and they'll all arrive at the barrier at approximately the same time. The other parts of our algorithm include the parallelized timing and bounding box updates which require more barriers to enforce precedence. Finally, the total amount of data copied between global and local data updates is constant regardless of the number of threads, but this part may become a bottleneck due to memory contention.

Also shown in Figure 5.7 is the per-processor parallelization efficiency obtained by dividing the speedup versus the number of threads used. It can be seen that as more processors are used, the efficiency drops off. The efficiency is about 42% when 30 threads are used. While it is difficult to achieve 100% efficiency for

our program due to the inevitable serial portions of the code, we breakdown the runtime of an individual run in the next subsection to identify factors that lead to this inefficiency.

Runtime Breakdown

Table 5.2 shows the runtime breakdown using the *stdev000* circuit with *inner_num* = 1. The initialization column includes memory allocation and data initialization for the parallel algorithm. The inner loop column measures time spent identifying swap candidates and the associated cost calculations needed for the swap evaluation. Incremental bounding-box calculation time is included in the inner loop as well. However, a separate bounding-box calculation must be done for the parallel program to ensure correctness, and is displayed in the bounding box update column (details in Section 3.7). The global to local data copy is time spent making local copies of global data structures, such as the timing information, needed for swap evaluation. The data broadcast column on the other hand measures the time spent updating the global data with local copies of data at the end of each sub-region evaluation (Line 6 and 12 in Section 3.2). Finally, the barrier column shows the average time each thread spent idling at the barrier. This quantifies load imbalance in our approach. The time is measured on a per-thread basis, and a geometric mean is taken from all threads to obtain the value as shown in the table.

As seen in the self-speedup curves in Figure 5.7, the program scales quite well up to 25 threads. However, efficiency drops when scaling beyond that point. Next, we investigate a few factors that may contribute to this performance limitation.

First, to paraphrase Amdahl's law, the speedup is limited by the non-scalable portions of the program. The only item on the list that grows in runtime is the

global to local data copy. It grew from about 3s to about 11s from 1 to 30 threads, increasing to 11% of the overall runtime. In addition, the barrier runtime is relatively constant from 1 to 30 threads, but progressively becoming a bigger proportion of the runtime. Furthermore, timing update has limited scalability due to the strong data dependencies discussed in Section 3.5. Although its runtime has been steadily decreasing up to 16 threads, its share of the total runtime increased by 12% from 1 thread to 25 threads. This suggests that the speedup obtained from timing analysis is less than the overall speedup, becoming a bottleneck.

Second, let's analyze the most parallelizable portion of the code — the inner loop. As the number of threads increase, the size of each private region decreases, leading to less parallel work assigned to each processor. Figure 5.8 shows the speedup curve for the *inner loop* only. It can be seen that this portion of the code scales extremely well, with nearly 100% processor efficiency up to 9 threads! At 25 threads, a speedup of 21 is achieved, which is equivalent to 83% processor efficiency. Workload imbalance is likely a contributing factor to the non-linear scaling of the program, and is investigated below.

We measure the amount workload imbalance by recording the frequency that each thread (a) arrives first at the barrier, meaning it takes the shortest time to complete its assigned work, and (b) arrives last at the barrier, meaning it is slowest to complete its assigned work. We ran the 25-threaded version three times using the *stdev000* circuit and the geometric mean result is shown in Figure 5.9. It can be seen that certain threads, such as thread 3, arrive first almost 20% of the time. The program could be more efficient if more work had been assigned to it. One possible cause for this imbalance is the private region assigned to this particular thread is very lightly populated, hence the amount of work is dramatically less than other

Table 5.2: Runtime breakdown

# thread	initialization		timing update		bounding box update		global to local data copy		inner loop		data broadcast		barrier (line 6)		total	self speedup
	[s]	%	[s]	%	[s]	%	[s]	%	[s]	%	[s]	%	[s]	%	[s]	
VPR	-	-	26.7	1	-	-	-	-	2832	98	-	-	-	-	2884	-
VPR-fast	-	-	17.5	6	-	-	-	-	283	93	-	-	-	-	303	-
1	0.05	0	268.8	25	22.3	2	3.1	0	787	73	1.84	0	0.0	0	1083	1.0
4	0.04	0	83.1	28	7.1	2	2.9	1	195	65	0.87	0	17.2	6	298	3.6
9	0.03	0	40.2	26	3.6	2	3.9	3	89	59	0.47	0	18.7	12	152	7.1
16	0.03	0	33.3	30	2.6	2	6.7	6	56	51	0.33	0	14.7	13	110	9.9
25	0.06	0	35.0	37	2.5	3	9.1	10	38	39	0.25	0	17.2	18	96	11.3
30	0.06	0	35.2	37	2.7	3	10.8	11	34	36	0.22	0	18.4	20	94	11.5

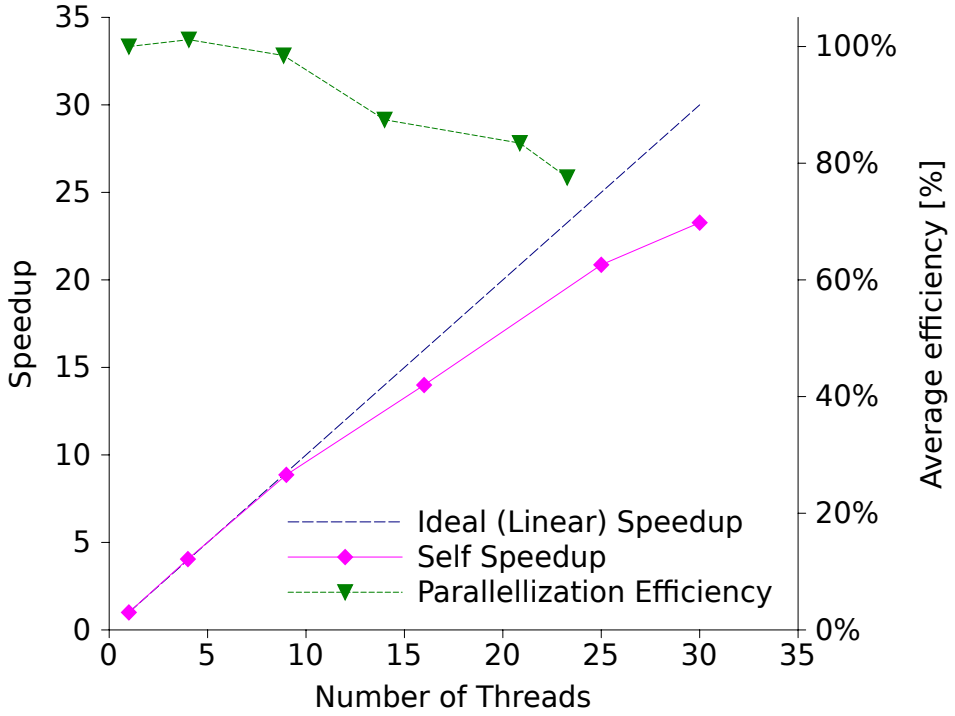


Figure 5.8: Speedup of the *inner loop*

threads. Another possibility is that the connectivity of the CLBs residing in this region is less than the others. Better load-balance schemes are an interesting topic that can be investigated in the future to further improve the scalability of the parallel placer.

It should be noted that time spent idle at the barrier could potentially be utilized to perform more iterations at each temperature, until all threads arrive at the barrier. This may improve quality without affecting runtime by allowing more opportunities for block migration. However, such an algorithm would be non-deterministic since it depends upon runtime data race conditions.

In summary, the inner loop scales quite well, even with up to 30 threads. How-

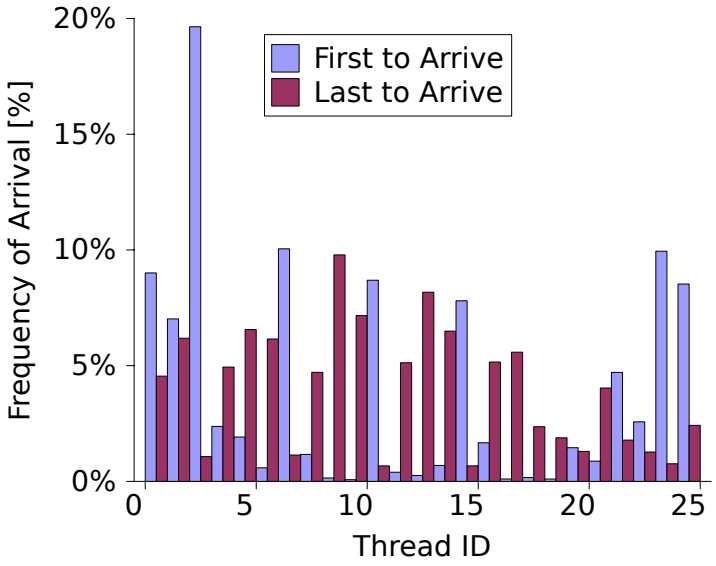


Figure 5.9: Probability of a region arriving first and last at a barrier due to workload imbalance

ever, since it no longer dominates the runtime, achieving more speedup becomes difficult as the hard-to-parallelize workload begins to dominate. We have shown that the *inner loop* of our parallel placer scales well up to 9 threads and achieves a per processor efficiency of 78% at 30 threads. Overall, however, the entire parallel placer achieves a per processor efficiency of 42%.

Our algorithm is scalable since the most time-consuming components scales rather well. While the runtime for data copying operations does increase with threads, different data-structures could perhaps alleviate this bottleneck and achieve even better scalability.

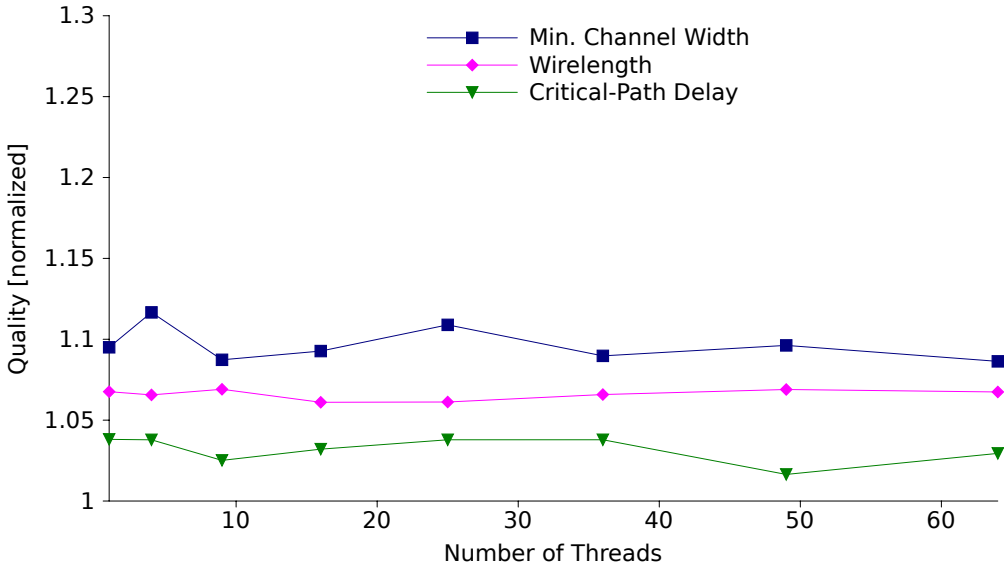


Figure 5.10: QoR by varying the number of threads

5.3.2 Quality Scaling

It is important to note that all of the benchmark circuits used here are the same size (roughly 40,000 luts). We anticipate that larger circuits will be scalable to an even larger number of threads as the amount of work done in the inner loop will increase.

Figure 5.10 shows the quality compared to VPR using up to 64 threads for the parallel algorithm with an *inner_num* value of 1. We can see the result is relatively constant all the way from 1 to 64 threads, showing that the QoR is unaffected by increasing the number of threads.

Another interpretation of these flat curves is that it is unaffected by (the amount of) stale data. Most importantly, the single-threaded version contains no stale data at all, yet it does not perform significantly better than the 64-threaded version which does contain stale data. As the number of threads increases, each thread

uses more stale data (since the region owned by a thread, which is perfect and not stale, shrinks in size). However, by the same logic, the use of more threads will also refresh the stale data more frequently, and limit the movement distance, which limits the magnitude of the stale data error.

One explanation for the good behaviour with stale data is due to the restricted local swap region size. Since a CLB is unlikely to move a great distance in just one iteration, its previous location (the location assumed by all other threads) becomes a rather good estimate for its new location. Even assuming a bad move has been made, due to the limited range of movements the amount of degradation to the placement is limited as well.

In addition, the way we divide work into regions also contributes to QoR and mitigates the impact of staleness. In particular, the limited range of movement within a swap-region keeps changes small relative to the length of long nets, thus mitigating the magnitude of the stale data error. Very small nets, on the other hand, have highest sensitivity, but they often fall entirely within the current swap-region and thus are not subject to staleness. This means that very short nets, where stale data has a large impact on cost, likely does not have stale data. Also, long-length nets will have their CLBs moved only a short distance. One limitation with our region division involves medium length nets which extend just beyond the swap-region boundaries; they may be involved in simultaneous swaps in multiple regions in opposite directions. There is currently no method to alleviate this, however, as the results suggest, this is not a dominating issue.

Table 5.3: Determinism verification runs

Circuit	# of threads	# of runs
stdev002	1	1000
stdev004	4	1000
stdev006	9	1000
stdev008	16	1005
stdev010	25	1000
stdev010	32	11524
Total number of runs		16529

5.4 Determinism

To help verify determinism, the parallel placer was run multiple times to ensure the same answer is generated each time. Due to time constraints, only one arbitrary circuit was selected for a specific thread configuration for testing. To facilitate this comparison, a CRC value is computed based on the placement output file; it encodes the placement location of all CLBs and, IOs. After each run, the bounding-box cost, the critical-path delay and the CRC value are compared to determine whether the result is identical. Table 5.3 shows the circuit used, the number of threads employed, and the number of verification runs completed. It can be seen that none of the runs produced a different result.

We also performed a small number of runs using an UltraSPARC T2 (Niagara 2) machine from Sun Microsystems, which support up to 64 threads, and Intel boxes with quad-core Nehalem processors, which support up to 8 threads with hyper-threading technology. The results obtained using the Sun and Intel box are identical to the results obtained using the AMD box.

5.5 Summary

In this section, we have shown that the QoR curve obtained from the parallel placer is superior to VPR in the fast runtime region. The maximum speedup obtainable with VPR plateaus at around 100x with quality degrading by as much as 100%. In contrast, our parallel placer can obtain speedups up to 500x with all quality metrics degrading less than 30%.

It is shown that the algorithm is both runtime and quality scalable. The algorithm exhibits good speedup up to 25 threads, with an approximate processor efficiency of 42%. With more optimizations and a machine with more processors, it would appear that the algorithm could scale beyond 30 threads. Then, it is shown that the quality is scalable, where all three PR quality metrics obtained are independent of the number of threads used from 1 to 64 threads.

The main limitation with the algorithm is that even with an extremely long runtime, it is still unable to achieve quality that is equivalent to VPR.

Chapter 6

Future Work

In this chapter, a few notable ideas for future work are presented.

6.1 Timing Analysis Speedup

The algorithm presented in this thesis simply parallelized the timing analysis algorithm originally developed for VPR. The main goal here was to preserve the decisions made in the original sequential algorithm to reduce the variation from the original algorithm. A natural extension is to investigate parallel-oriented timing analysis algorithms which load-balances a bit better, and scales regardless of circuit size. Incremental timing analysis update presented, in [43], may be helpful as well. Timing analysis as shown in Table 5.2 takes up 25 % of runtime, even at one thread. Compared to VPR, which allocates less than 6% of its runtime to timing update, this suggests that timing analysis in the parallel placer is invoked too frequently. Characterization to determine the optimal update frequency and perhaps an adaptive frequency which depends on the current temperature or target runtime can be designed.

6.2 Runtime Scaling

Our algorithm was shown to scale up to 25 threads, but further scaling requires careful study to alleviate the bottlenecks. New data structures to reduce the global to local data copying time and perhaps more efficient barrier designs could be part of the future work. A dynamic region allocation scheme where the net connectivity is considered in addition to number of CLBs may be useful to alleviate load imbalance. The `PROB_SKIPPED` characterization experiment described in Section 4.3 could be modified to identify the optimal `PROB_SKIPPED` value given a constant number of swaps; this can be achieved by adjusting both `PROB_SKIPPED` and *inner_num* simultaneously.

6.3 Serial Equivalence

Although not demonstrated in this thesis, it is possible to achieve serial equivalence with a moderate amount of code changes. Depending on the implementation, this may also have an effect on performance as well.

One way to establish a serial-equivalent algorithm is described as follows. Given a sufficiently large circuit, partition the grid for a large number of threads n, where n is at least equal to the number of threads in the largest multi-threaded machine used to run this code. Then, assign multiple partitions to each thread until all partitions are exhausted. Threads with multiple partitions constantly switch between partitions assigned to it while updating its memory contents as needed. This may also help alleviate load imbalance, as threads can trade partitions dynamically.

6.4 LUT Placement

While it is difficult for the parallel placer to match the quality of the serial placer, even with adding a substantial number of swaps, we take a slightly different approach here to see whether the quality can be recovered in another way.

The advantage of the parallel placer is its massive speedup and the potential for even more parallelism as newer hardware becomes available. With so much computational power, perhaps it's time to revisit the overall CAD tool flow to seek other potential improvements.

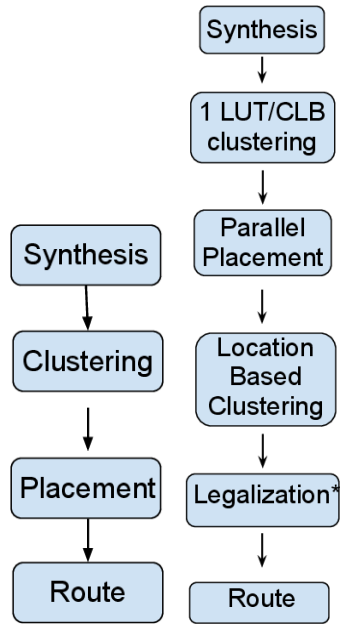
A conventional placer takes clustered circuits as its input and attempts to assign a unique location to each cluster. While the clustering of circuits dramatically decreases the amount of computation for the placer, it inhibits LUT movements between different clusters. In another words, the quality of placement is limited due to clustering since LUTs are locked into clusters already.

With the computational power enabled by the parallel placer, perhaps it is now possible to place LUTs directly within a reasonable runtime. This would avoid clustering of circuits and possibly achieve a superior quality compared to VPR. An experiment is conducted here to determine whether this idea is feasible.

6.4.1 Experiment Procedure

We devised an experiment to evaluate the potential quality gain from placing individual LUTs directly. Figure 6.1 shows the conventional flow on the left and the experimental flow on the right.

In the conventional FPGA CAD tool flow, the given circuit is packed into clusters containing up to 9 LUTs each. (The actual cluster size does not matter, as long as it is consistent between the tools being compared.) The clustered circuit is then



(a) Conventional Flow (b) Experimental Flow

Figure 6.1: Conventional CAD tool flow versus experimental flow

placed and routed with VPR to obtain post-routing metrics.

In the proposed experimental flow, the placer is free to move individual BLEs. The netlist is first clustered into individual CLBs using a cluster size of N=1. This step is utilizes existing FPGA CAD tools without making extensive code modifications. The circuit is then placed using the parallel placer. The output is a placement file is based on cluster size of 1, or equivalently with 1 BLE placed uniquely into each grid location. Based on this placement file, location-based clustering is accomplished after placement by clustering every 3×3 square (9 blocks) into a single CLB, achieving the equivalent cluster size of 9 as the conventional flow. Then, this circuit is routed using VPR to obtain post-routing metrics.

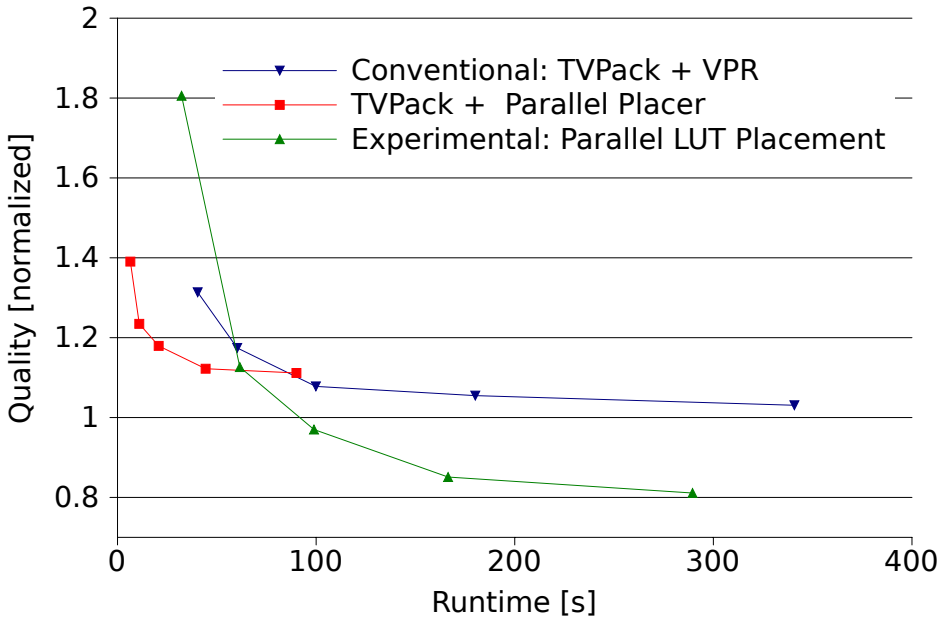
The legalization step in the experimental flow, as indicated by the asterisk (*), is not yet completed. One way to do it, as shown, is to clean up the solution afterwards by making small local swaps to correct the number of inputs per CLB and I/O block locations so they satisfy the clustered FPGA architecture. A better approach would be to rewrite VPR so that it can move individual BLEs while retaining the concept of a larger CLB, and enforce legalization during move evaluation. However, before making such extensive changes to the code, we wanted to run this easier experiment to verify whether it is a worthwhile path to investigate.

Since our legalization step is unfinished, we instead chose to modify the conventional clustering parameters in order to establish a fair comparison. Hence, we did not place a limit on the number of inputs per cluster during T-VPack in the conventional flow, since the number of inputs is not constrained in the experimental flow either.[1] In addition, one of the circuits (stdev002) had an overflow of 4 I/Os at two different I/O locations for the experimental flow. We got around this by simply increasing the number of I/O blocks in the architecture file. It does give a slight advantage to this particular circuit, however, we believe that a few pins on one circuit are unlikely to have a significant impact on the overall quality.

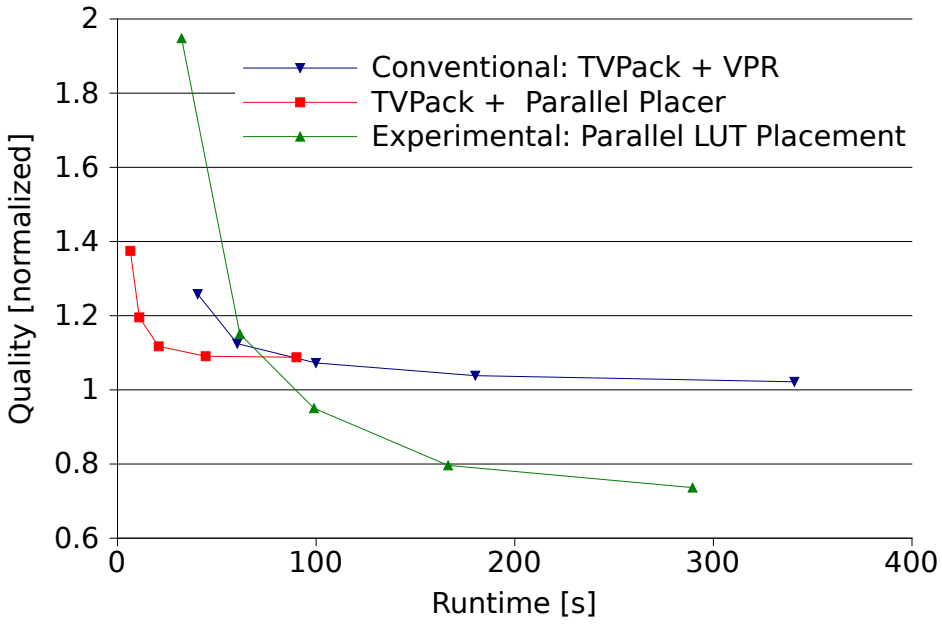
The parallel placer is run with 30 threads to perform all of the parallel placement experiments. T-VPack [14] is used to perform all of the clustering operations.

We route the placed circuits using VPR with flags shown in Figure 5.1 to obtain the minimum routable channel width. The wirelength at this channel width is recorded along with the wirelength metric. We then relax the channel width by 20% and obtain the critical-path delay metric shown in Figure 5.2.

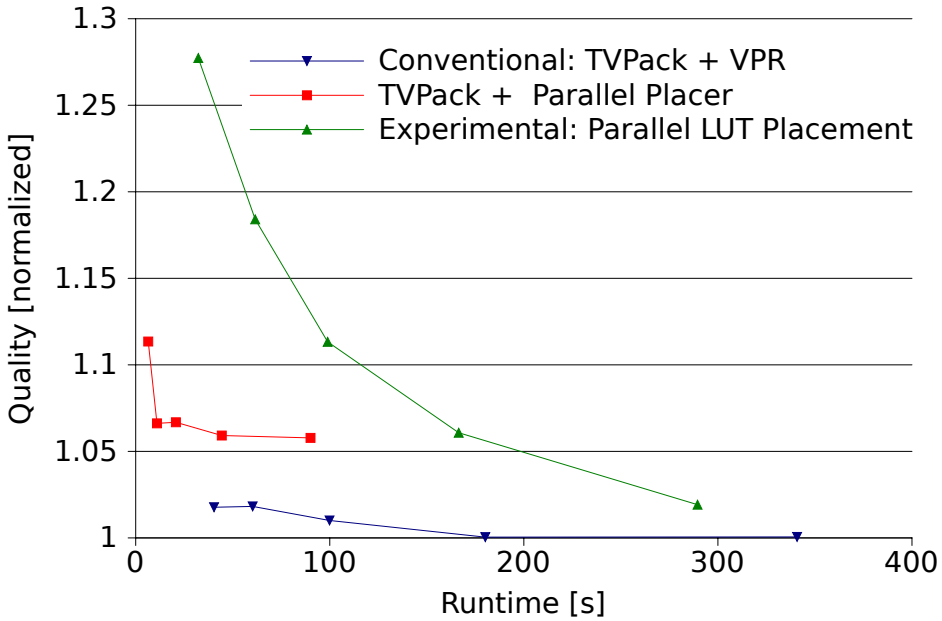
[1] We actually ran T-VPack in two modes, one with limited cluster inputs, and another with unlimited cluster inputs. We did not find an appreciable difference in trends.



(a) Wirelength



(b) Minimum channel width



(c) Critical path delay

Figure 6.1: Runtime versus quality comparison for post routing minimum routable channel width, wirelength and critical-path delay

6.4.2 Results/Discussion

In this section, we compare the quality and runtime of the various approaches described above.

All values are normalized against circuits packed with T-VPack (cluster size of 9 with an unlimited CLB inputs) and placed using VPR with the default inner_num of 10. Then the same method as previously explained is used to obtain wirelength and critical-path delay metrics. Finally, we compute the geometric mean across the 7 circuits to obtain 1 datapoint for each inner_num value.

The measured runtime in our results includes only the time spent performing placement operations. This does not include initialization time such as netlist load-

ing. All unmeasured time are identical between the serial and the parallel placer.

6.4.3 Quality versus Runtime

We compare the final post-routing metrics amongst all approaches in Figure 6.1.

We ran all circuits and obtained the quality/runtime curve by iteratively decreasing the *inner_num* value of VPR and the parallel placer. The right-most datapoint (at roughly 300s) is obtained using an inner_num value of 1, also known as VPR '-fast', and the left-most datapoint is generated using an inner_num value of 0.0625 for all curves shown.

Conventional versus Experimental Analysis

The experimental flow greatly outperforms the conventional flow in wirelength and minimum channel width metrics. However, the conventional flow is slightly better in critical-path delay. For wirelength, the experimental flow is better in the majority of the cases, decreasing the wirelength by as much as 28% at the right-most datapoint. The minimum channel width trend is similar to wirelength, with the experimental flow achieving up to 21% improvement. For critical-path delay, the quality of the experimental flow gradually approaches the conventional flow, but is still degraded by about 1.8% at the right-most datapoint.

One interesting observation is that the trend suggests that the parallel LUT placement requires *longer* runtime to achieve a better result. This does makes sense since LUT placement is a much more complex problem than the clustered placement. With shorter runtime, insufficient 'hill-climbing' is done, resulting in the rather quick quality degradation for the parallel placer. This could also be a potential algorithm-related property, since the partitioning method used in the parallel

placer may inhibit block migration, which may make it more vulnerable during the short runtime region. Nevertheless, this is the first time a parallel placer has outperformed the conventional state-of-art serial placer in absolute quality. With more tuning and care, we expect the parallel placement curve can shift towards the left side, or obtain better quality in the fast runtime region, with the serial placer staying constant.

Conventional with Parallel Placer

We also show the runtime/quality curve for the parallel placer put through the conventional flow, placing the benchmark circuits clustered with T-VPack and a cluster size of 9. (i.e., replacing the serial VPR placer in the conventional flow with the parallel placer).

The parallel placer has runtime advantages over the serial placer, as shown in Figure 6.1. However, the trend does not suggest that the quality obtained this way will be superior than using the conventional serial placer. Also, it does not seem promising that the critical-path delay quality will ever become equal.

In contrast, the curve obtained with the experimental flow exceeds the quality by a large margin in wirelength and asymptotically becomes close in critical-path delay. Although the LUT placement runtime is longer, since the LUT placement problem is more complex than placing clusters, it is believed that future advancements in parallel algorithm research could help to alleviate this. With the trend towards increasing parallelism, we believe placers should consider moving individual LUTs rather than clusters to the recover the valuable quality lost due to clustering. To address quality at the fast runtime region, a hybrid approach where clustering is used to produce an initial solution, or where the placer alternates be-

tween moving entire CLBs and moving individual LUTs, might be necessary.

6.4.4 Conclusions

This work conducted an experiment comparing the quality between a conventional CLB-based placer and a parallel LUT-based placer. It has shown that up to 28% in wirelength and 21% in minimum channel width improvement is achievable by placing individual LUTs without the need for clustering before placement. Critical-path delay is worse, but shows promise if enough runtime is given. Utilizing the computational power of multi-core machines available today in combination with a parallel placer, the LUT placement was able to achieve a shorter runtime, with a better quality of result, than the conventional placement based on clustering. While clustering may still be useful for certain purposes, such as obtaining an initial placement, we encourage future parallel placement research to consider placing individual LUTs directly to recover valuable quality lost due to the clustering algorithm.

One of the key limitations of our work is that we were unable to restrict the number of inputs per CLB during parallel placement. For a fair comparison, we gave the clustering tool the same freedom of using as many inputs as needed. This should be addressed in future work and evaluated for its effects on the quality of result.

Chapter 7

Conclusions

This thesis presents a parallel placement algorithm that is both deterministic and timing-driven. Although the original serial version of VPR's simulated-annealing placement engine can be accelerated by performing fewer moves per temperature, the quality of result degrades severely past 100X speedup. In contrast, at the point where the parallel algorithm presented in this thesis beats serial VPR in quality, we achieved a speedup ranging between 140X to 200X using 25 threads, with post-routing minimum channel width, wirelength and critical-path delay degraded by 13% , 10% and 7% respectively on average. While VPR cannot accelerate much beyond 100X, our parallel algorithm scales beyond 500X with all quality metrics degraded by less than 30%. However, given a sufficiently long runtime, the parallel placer cannot beat serial VPR in quality: it is 7.6%, 4.6%, and 1.6% worse than VPR on post-routing minimum channel width, wirelength and critical-path delay, respectively.

The algorithm is both quality and runtime scalable. Varying the number of threads and using stale data does not have an effect on the quality of result. This

is encouraging as it shows the algorithm can potentially scale to even more threads on longer circuits without any further loss of quality. With the fixed size of the Un/DoPack circuits, results do not improve beyond 25 threads. The limitations appear to be load balancing and timing updates. Global to local data copying is another bottleneck and we believe new data structure could potentially alleviate the runtime lost here.

We also conducted an experiment to investigate whether it is worthwhile to perform parallel placement based on individual LUTs. It was shown that quality improvements of up to 28% can be achieved over VPR with less runtime. This would be especially useful for individuals that cannot tolerate any quality loss but still demands speed up. We believe that this is a promising approach for future the parallel placer research in order to achieve superior quality than the serial VPR placer.

Bibliography

[1] C. C. Wang and G. G. F. Lemieux, "Scalable and deterministic timing-driven parallel placement for FPGAs," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2011, pp. 153–162.

[2] J. B. Goeders, G. G. F. Lemieux, and S. J. Wilton, "Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition," in *to appear in ReConFig*, 2011.

[3] Xilinx Inc., "Stacked silicon interconnect technology," http://www.xilinx.com/technology/roadmap/ssi-technology.htm, 2011.

[4] Altera Corporation, "Quartus II 10.0 Handbook," http://www.altera.com/literature/hb/qts/qts_qii51008.pdf, 2010.

[5] M. Santarini, "Xilinx Tailors Four Tool Flows to Customer Design Disciplines in ISE Design Suite 11.1," http://www.xilinx.com/support/documentation/ white_papers/wp307.pdf, 2009.

[6] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2008, pp. 14–23.

[7] H. Bian, A. C. Ling, A. Choong, and J. Zhu, "Towards scalable placement for FPGAs," in *Proceedings of the 18th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010, pp. 147–156.

[8] W. Swartz and C. Sechen, "New algorithms for the placement and routing of macro cells," in *International Conference on Computer-Aided Design*, Nov. 1990, pp. 336 –339.

[9] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Parallel algorithms for FPGA placement," in *Proceedings of the 10th Great Lakes Symposium on VLSI*, 2000, pp. 86–94.

[10] J. Luu, I. Kuon, P. Jamieson, T. Campbell, A. Ye, W. M. Fang, and J. Rose, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *Proceedings of the 17th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2009, pp. 133–142.

[11] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[12] I. Kuon and J. Rose, "Area and delay trade-offs in the circuit and architecture design of FPGAs," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2008, pp. 149–158.

[13] J. Lamoureux, "On the interaction between power-aware CAD algorithms for FPGAs," Master's thesis, University of British Columbia, Vancouver, Canada, 2003.

[14] A. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve FPGAs speed and density," in *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1999, pp. 37–46.

[15] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *IEEE International Conference on Field Programmable Logic and Applications*, 1997, pp. 213–222.

[16] S. Kirkpatrick, C. Gelatt, and M. Vecchi, in *Science*, vol. 220, no. 4598, May 1983, pp. 671 – 680.

[17] J. Lam and J. Delosme, "Performance of a new annealing schedule," in *Design Automation Conference*, 1988, pp. 306 –311.

[18] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," in *IEEE Transaction on CAD of Integrated Circuits and Systems*, September 1987, vol. 6, no. 5, pp. 838 – 847.

[19] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," in *IEEE Journal of Solid State Circuits*, April 1985, vol. 20, no. 2, pp. 510 – 522.

[20] A. Marquardt, V. Betz, and J. Rose, "Timing-driven placement for FPGAs," in *Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2000, pp. 203–213.

[21] J. S. Rose, "Parallel global routing for standard cells," in *IEEE Transactions on Computer Aided Design*, 1990, pp. 1085–1095.

[22] Y. Chang, S. Thankur, K. Zhu, and D. Wong, "A new global routing algorithm for FPGAs," in *IEEE Transactions on Computer Aided Design*, 1994, pp. 356–361.

[23] S. Brown, J. Rose, and Z. Vranesic, "A detailed router for FPGAs," in *IEEE Transactions on Computer Aided Design*, 1992, pp. 620–628.

[24] G. Lemieux and S. Brown, "A detailed router for allocating wire segments in FPGAs," in *ACM Physical Design Workshop*, 1993, pp. 215–226.

[25] G. Lemieux, S. Brown, and Z. Vranesic, "On two-step routing for FPGAs," in *ACM Symposium on Physical Design*, 1997, pp. 60–66.

[26] M. Placzewski, "Plane parallel A* maze router and its application to FPGAs," in *ACM Design Automation Conference*, 1990, pp. 691–697.

[27] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proceedings of the 3rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 1995, pp. 111–117.

[28] Y.-L. Wu and M. Marek-Sadowska, "An efficient router for 2-d FPGAs," in *European Design Automation Conference*, 1994, pp. 412–416.

[29] Y.-S. Lee and A. Wu, "A performance and routability driven router for FPGAs," in *ACM Design Automation Conference*, 1995, pp. 557–561.

[30] B. Barney, "POSIX thread programming," https://computing.llnl.gov/tutorials/pthreads/, 2011.

[31] S. Kravitz and R. Rutenbar, "Placement by simulated annealing on a multiprocessor," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 4, pp. 534 – 549, Jul. 1987.

[32] P. Banerjee, M. H. Jones, and J. S. Sargent, "Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors," *IEEE Transactions Parallel Distributed Systems*, vol. 1, no. 1, pp. 91–106, 1990.

[33] A. Choong, R. Beidas, and J. Zhu, "Parallelizing simulated annealing-based placement using GPGPU," in *IEEE International Conference on Field Programmable Logic and Applications*, Aug. 2010, pp. 31–34.

[34] W.-J. Sun and C. Sechen, "A loosely coupled parallel algorithm for standard cell placement," in *International Conference on Computer-Aided Design*, 1994, pp. 137–144.

[35] J. Rose, W. Martin Snelgrove, and Z. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," in *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 3, March 1988.

[36] E. Witte, R. Chamberlain, and M. Franklin, "Parallel simulated annealing using speculative computation," in *IEEE Transactions on Parallel and distributed systems*, vol. 2, no. 4, October 1991.

[37] M. G. Wrighton and A. M. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *Proceedings of the 11th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2003, pp. 33–42.

[38] G. Smecher, S. Wilton, and G. Lemieux, "Self-hosted placement for massively parallel processor arrays," in *Field-Programmable Technology*, Dec. 2009, pp. 159 –166.

[39] A. Ludwin and V. Betz, "Efficient and deterministic parallel placement for FPGAs," in *ACM Transactions on Design Automation of Electronic Systems*, vol. 16, no. 3, June 2011.

[40] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[41] M. L. Scott and J. M. Mellor-Crummey, "Fast, contention-free combining tree barriers for shared-memory multiprocessors," in *International Journal of Parallel Programming*, 1994, 22(4), pp. 449–481.

[42] M. Tom, D. Leong, and G. Lemieux, "Un/DoPack: re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs," in *International Conference on Computer-Aided Design*, Nov. 2006, pp. 680 –687.

[43] K. Eguro and S. Hauck, “Enhancing timing-driven FPGA placement for pipelined netlists,” in *Design Automation Conference*, 2008, pp. 34–37.