Scalable and Deterministic Timing-Driven Parallel Placement for FPGAs

Chris Wang
Dept. of ECE
University of British Columbia
Vancouver, BC, Canada
chrisw@ece.ubc.ca

Guy G.F. Lemieux
Dept. of ECE
University of British Columbia
Vancouver, BC, Canada
lemieux@ece.ubc.ca

ABSTRACT

This paper describes a parallel implementation of the timing-driven VPR 5.0 simulated annealing engine. By restricting the move distance to a confined neighborhood, it is possible to consider a large number of non-conflicting moves in parallel and achieve a deterministic result. The full timing-driven algorithm is parallelized, including the detailed timing analysis updates done periodically while placement progresses. The limited move slightly degrades the placement quality, but this is necessary to expose greater degrees of parallelism. The overall bounding box metric degrades about 11% and critical path delay metric degrades about 8% compared to VPR's original algorithm, but we show the amount of degradation is independent of the number of threads. Overall, the parallel implementation scales to a speedup of 123x using 25 threads compared to VPR. With additional tuning effort, we believe the algorithm can be scaled to a larger number of threads, perhaps even run on a GPU, with little additional quality degradation.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Placement and routing*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Design, Performance

Keywords

Parallel placement, FPGA, Timing-driven placement

1. INTRODUCTION

As FPGA logic capacity steadily increases at the rate provided by Moore's Law, FPGA CAD must synthesize, place and route more logic blocks and more nets every generation. Keeping runtime and quality of results (QoR) constant while the number of objects continues to grow is a demanding task because uniprocessor performance improvements do not track FPGA capacity growth. To keep runtime in check, 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. A promising solution to the runtime crisis is to employ parallel CAD algorithms so the number of working processor cores and FPGA capacity can both scale at similar rates. However, in addition to the raw numbers scaling, the algorithms must also demonstrate good scaling results in terms of QoR and runtime. With multicore processors now common, Altera[1] and Xilinx[2] have both started to implement parallel algorithms that offer some runtime improvement.

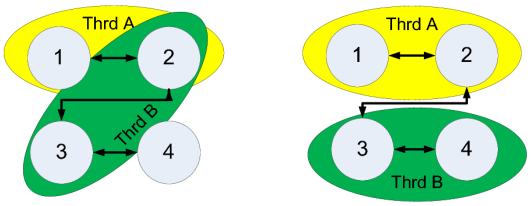
One of the most time-consuming steps in the FPGA CAD flow is placement. A good quality placement is essential to reduce interconnect delay, localized congestion, wirelength and power. Simulated annealing, the placement engine used in VPR and Altera's Quartus II tools, is widely regarded as producing 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 [3]. The conclusion was that "simulated annealing based placement would still be in dominant use for a few more device generations." In VPR, the number of moves needed to find an optimized solution grows as $O(N^{4/3})$ [4], where N is the number of CLBs being placed. As more objects must be placed, the VPR placement algorithm slows down due to nonlinear scaling.

In recent work, Altera described a parallel timing-driven annealing-based algorithm in Quartus II [5] which evaluates many moves in parallel, but serially commits the moves to achieve a deterministic placement. A speedup of 2.2x was demonstrated on 4 processors, and QoR is equivalent to the serial version. However, this algorithm may not runtime-scale to a large number of cores because multiple cores increases the probability of both *hard* and *soft conflicts*[1] between moves. A conflict of any type requires a speculated move to be abandoned, or its cost to be recomputed serially at commit time.

In contrast, this paper parallelizes the full timing-driven annealing-based algorithm in VPR5. However, unlike the Altera work, we show that our approach can scale to a larger number of processors. We do this by allowing moves and commits to occur in parallel. The key is to avoid generating moves that have hard conflicts by greatly restricting the range of motion of CLBs for each move. By using stale (or imprecise) placement information for dis-

[1]Conflicts are defined in Section 2.

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.



(a) Hard Conflict (b) Soft Conflict

Figure 1: Illustration of a hard and soft conflict

tant CLBs handled 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 the number of processors nor race conditions. While Altera emphasizes serial equivalence for easier regression testing and customer support, we believe that an algorithm that is deterministic is sufficient.[2] These changes introduce about 11% and 8% lost QoR on bounding box cost and critical path delay, respectively. However, the QoR does not degrade as the number of threads is increased, and the algorithm exhibits good self-speedup up to 25 threads. The algorithm achieves a speedup of 123x with 25 threads compared to the original sequential VPR algorithm. With additional tuning, we believe that it is possible to achieve additional speedup with no further lost QoR if more threads are used. Finally, we think it may be possible to implement a GPU-hosted version of this parallel placement algorithm as well.

2. PREVIOUS WORKS

Parallel placement research has been underway for more than two decades. To the best of our knowledge, all published work with the exception of [5] have targeted wirelength only. Furthermore, determinism, an extremely important feature for bug reproducibility and testing [5], has not been considered except in [5, 6]. 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.

Before we review placement algorithms, we must first define two types of conflicts. A simple circuit with four CLBs is shown in Figure 1, where arrows represent nets that connect CLBs. 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, producing a hard conflict. A *soft conflict* arises when different CLBs that are connected by the same net are considered concurrently. In the second example, thread B now considers block 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.

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 to be from independent sets [7, 8, 9]. Speculative move proposal was employed to further accelerate the algorithm [5], and a dependency checker, executed serially, is used to ensure no hard-conflict has occurred. The algorithm as reported in [5] achieved a speed up of 2.2x on four cores. However, it likely suffers in terms of scalability beyond four cores. 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 allocate 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 [6, 10]. Sun and Sechen [10] employed an algorithm based on dynamic region generation by dividing the chip vertically and horizontally on alternating iterations. It was implemented on multiple machines connected by LAN. During each iteration, each machine would receive 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 would only update cells changes at the end of each iteration and move evaluation is based on the cell locations from the previous iteration. It achieved a speedup of 5.3x using six machines and yielded comparable results against the best serial algorithm known at the time. 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 considers bounding box cost only and are non-deterministic, making them unsuitable to be implemented in commercial tools.

Interested readers are encouraged to consult [4] and [5] for more detailed summaries and other parallelization methods.

Work by Wrighton and DeHon [11] 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 worked 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 upon information staleness.

More recently, work by Smecher, Wilton and Lemieux [12] demonstrated that the algorithm used in [11] could be applied to placement of communicating tasks for massively parallel processor arrays (MPPAs). 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 [11] and [12] 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 [11] and [12] to the full timing-driven placement algorithm from VPR5 using pthreads so it runs on readily available shared-memory multicore computers.

[2] In our work, the algorithm is deterministic because T threads, where T is fixed, can be run on any number of processors (even $< T$) and produce the same result.

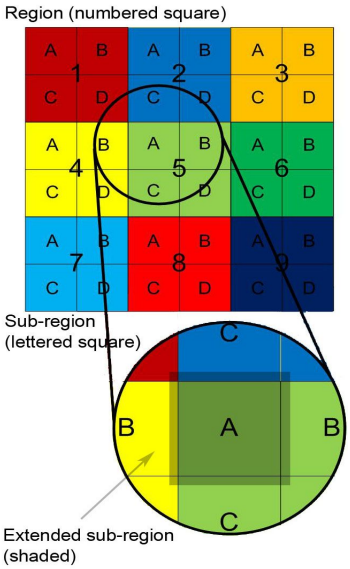


Figure 2: Equal-sized grid partition

For our experiments, we ran our parallelized VPR on an 8-core (64-thread) Sun Niagara 2 system, an Intel quad-core Nehalem and dual quad-core Clovertown, and an AMD quad hex-core Istanbul system. We report only Sun results due to the larger available parallelism, but Intel/AMD results were similar. Based on our results, we expect it is possible to develop an OpenCL or CUDA implementation that can also achieve good speedup. *The parallel placement code is freely available from the authors.*[3]

3. ALGORITHM DESCRIPTION

3.1 Algorithm Overview

Our parallel placement algorithm is based on partitioned regions, namely the second group of algorithms described in Section 2. The entire grid is first partitioned into approximately equal-sized private regions and assigned to a unique thread. An example is shown in Figure 2 with regions 1 through 9. The increased capacity (number of cells) in IOs around the peripheral of the chip can be ignored as the sparse IO cell occupancy balances this out. Therefore it can be estimated that each block, whether CLB or IO, contains approximately one cell. When the number of rows/columns in the grid cannot be evenly divided, the non-peripheral (not first, nor last) rows/columns will receive the extra rows/columns first. The number of columns and rows does not need to be same, *i.e.*, each region can be rectangular. Each thread will iterate sequentially through all CLB locations in its private region and considers the block at each position for a swap. To achieve a good quality versus runtime result, not every block is considered for the swap: a randomly generated number determines whether a particular block should be considered. If selected, this block will be paired with another randomly selected block within its legal swap region (defined below).

Placement data about blocks inside a thread's private region is precise, but placement data about blocks being managed by other threads is allowed to grow stale and updated only periodically. By further dividing each private region into sub-regions A through D (Figure 2), and placing only one sub-region at a time, we can ensure the resulting placement is deterministic. When placing a sub-region, blocks that reside inside will have the chance of swapping with blocks located near the edge of the adjacent sub-regions, a window we call the extended sub-region (Figure 2), to allow blocks to migrate. An adaptive schedule is used to guide the simulated annealing process.

To summarize, Figure 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]http://www.ece.ubc.ca/ lemieux/download/

3.2 Pseudo-code

The following is the thread-level pseudo code:

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

In this algorithm, `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 3.3.3), implying there is a 90% probability that each block is considered for a swap or simply, 90% of the blocks will be considered.

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.

3.3 Implementation Details

We implemented our changes directly into VPR 5.0.2 [13], which is based on the original VPR first published in [14], by parallelizing the *try_place()* function. We did not make major changes to the existing data structure in the software. To alleviate the demand on communication cost and create a deterministic program, we created local copies of global variables used 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 global memory to local memory. Conversely, after evaluating each sub-region, all threads update the global memory to reflect the changes made to their extended sub-region.

POSIX threads (pthreads) were used. However, the POSIX barriers may suspend and resume the thread, which can introduce process scheduling overheads. In our program, the overhead to suspend and resume is not justifiable because most threads have approximately equal work and hence similar execution time. Instead, we implemented tree-based polling barriers using shared global memory based on [15]. Our custom barriers performed much better than the POSIX barriers.

Prior to this shared-memory implementation, we initially wrote a message-passing implementation using MPI. This provided us with a great understanding of how to replicate local copies of the data structures without inadvertently sharing global structures. Ultimately, we discarded the MPI implementation because it did not achieve good speedups.

Except where noted, data presented in this paper are geometric means of all circuits normalized against values obtained from VPR.

3.3.1 Determinism Enforcement

We first subdivide each private region into four (2×2) approximately equal *sub-regions*. Each sub-region then is extended by two rows or columns along its boundaries to allow blocks to migrate between sub-regions, we name this combined region (sub-region and the two rows/columns) the *extended sub-region*. To achieve determinism, we must make sure all active extended sub-regions *do not* overlap with each other, thereby avoiding hard conflicts.

As shown in Figure 2, each of the sub-regions is labelled A through D. All threads start by placing in sub-region A, and barriers are used to ensure no thread can proceed to sub-region B before every thread finishes with sub-region A. During this time, each thread updates placement data for its own extended sub-region, but allows placement data about all other threads grow stale. At the end of the sub-region placement, all threads broadcast their placement updates, giving a single consistent view of all placements. Using this synchronization as a new starting point, all threads proceed to the next sub-region. This allows each thread to work in its private region independently without costly fine-grain synchronization. As we will show in Section 5.3, stale data does not deteriorate the quality of the placed result.

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 associative and can produce different results due to round-off if the order of addition changes. We maintain a constant order for the final addition by allocating an array with one entry for each thread. The master thread performs the summation in sequential order only after all worker threads complete, producing a deterministic sum. Since the work division scheme (more details in Section 3.3.5) depends only on the connectivity of the nets, which is constant for a given circuit, the work allocated to each thread is also deterministic. This ensures partial results will also be the same. Thus, cost results are all deterministic.

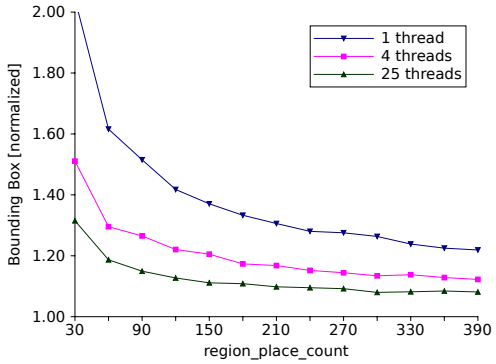
Finally, to keep the program deterministic, each thread tracks its own random number state. Thus, random numbers generated in one thread do not disturb the sequence of those made by other threads.

To help verify determinism, we ran each circuit at least 1000 times on different computer systems (eg, Intel and AMD) using the same binary executable. For a given number of threads, we observed the same bounding box, critical path delay, and placement CRC values across all the runs.

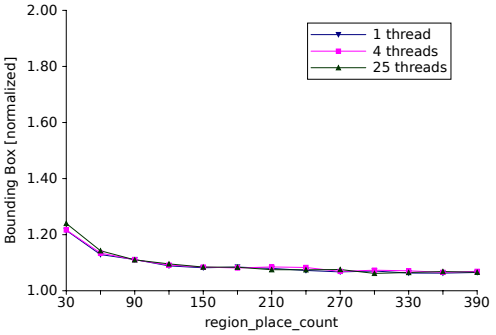
3.3.2 Legal Swap Region

The legal swap region is the neighborhood from which each block finds its swap candidate. To allow blocks to migrate between private regions, our algorithm allows each thread to consider blocks within its extended sub-region as swap candidates, provided they also fall within the legal swap region window. Therefore, the minimum dimension for each sub-region is 4×4, so that even if two neighboring threads decide to swap with blocks two blocks beyond its boundary, hard conflicts will always be avoided.

We first implemented the legal swap region as all blocks within a Manhattan distance of two as described in [12], however, it produced no significant advantage in the quality versus runtime graph when compared against VPR. Next, we extended the legal swap region to the entire extended sub-region and realized the result was even worse for certain number of threads and the result varied as depicted in Figure 3(a). Then we added the parameter *rlim* as used in VPR to limit the legal swap region as the placer begins to consider timing path delay. This dramatically improved the QoR. Finally, we experimentally determined that limiting the *rlim* value to 20 produces better QoR. The result is shown in Figure 3(b), and it is clear that we have successfully minimized the quality variation using an *rlim* value maxed at 20.



(a) with no restriction on size



(b) with rlim maxed at 20

Figure 3: Legal swap region quality variation

3.3.3 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 behavior of the other parameters. It dictates the probability each block will be *not* considered for a swap. A value of 100 is equivalent to not making any swaps, resulting in the initial placement. Figure 4 shows the runtime versus quality trade-off sweeping `PROB_SKIPPED` from 0 to 100 with 4 threads and an *region_place_count* value of 90. The experiment uses the *rlim* value maxed at 20 (Section 3.3.2) and the sequential block selection scheme (Section 3.3.4). 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 not consider every block and still maintain a good result. We see from Figure 4 that the quality of both parameters degrades rather slowly for small `PROB_SKIPPED` values. We selected a

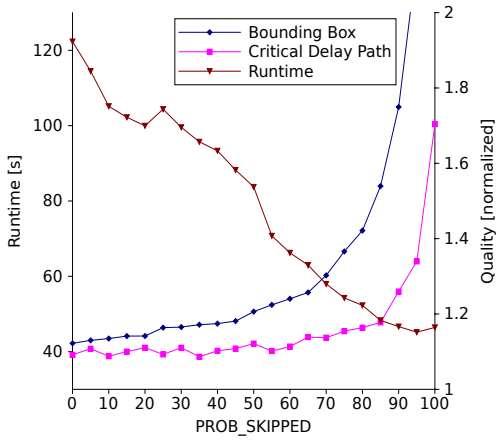


Figure 4: PROB_SKIPPED sweep

PROB_SKIPPED value of 10 as it gave us an approximately 14% runtime reduction for approximately 1% quality loss.

3.3.4 *Sequential vs Random Block Selection*

Unlike VPR, our algorithm sequentially iterates through each block position in the grid. To reach this conclusion, we ran an experiment with a controlled number of swaps per temperature range and compared the result from the sequential vs random block selection scheme using 1 thread and 25 threads. The program also incorporates *rlim* maxed at 20 (Section 3.3.2) and PROB_SKIPPED value of 10 (Section 3.3.3). The result is shown in Figure 5.

It can be seen that the quality of bounding box was clearly better in the case of sequential block selection. In the critical path delay graph, the quality is somewhat indistinguishable between the two block selection methods. Therefore, we adopted the sequential block selection scheme as it performed no worse than the alternative method.

3.3.5 *Parallel Timing Analysis*

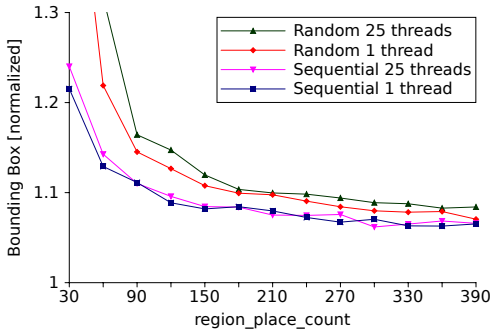
Periodically, after a significant number of moves, it becomes necessary to perform a timing analysis to identify critical paths. This time-consuming task quickly became a bottleneck, so we had to be careful not to invoke it too frequently, and we had to update it in parallel using multiple threads. The function *timing_update_parallel()* in the pseudo-code indicates this task.

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* and *delay cost*. Barriers were used to enforce data dependencies between the aforementioned functions, as well as further enforce precedence in the calculations.

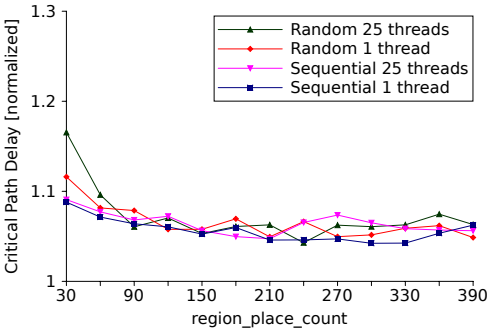
The parallelization of edge delay, timing and delay cost calculation are straight forward. As these functions loop through all the nets in the circuit, we could simply allocate an equal number of nets to each thread. Each thread stores its partial result in pre-allocated location in an array, which is later summed by the master thread as described in Section 3.3.1.

Parallel slack and criticality computation requires more synchronization because it traverses the tree to perform breath-first modifications twice. As there are no data dependencies within each level, we can process each level at a time and safely distribute the work of each level among the threads.

We made a slight data structure change to assist with parallelizing the tree traversal process. In the original VPR code, only children of each node were stored, which was adequate since the arrival



(a) Bounding Box quality comparison



(b) Critical Path Delay quality comparison

Figure 5: Sequential vs random block selection comparison

time for each node was calculated passively. Each parent calculates the earliest arrival time of the child with respect to itself and compares with the arrival value stored in the child node, which may have been computed earlier by another parent of the child. While this approach works for a single threaded program, we run into issues in the parallel version if two different parents are evaluating the same child concurrently. To resolve this, we decided to store the parent of each node, so that each node can determine its own latest arrival time by traversing through all of its parents. 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.

In many cases, the amount of work depends on an attribute of the inner loop, which is not directly visible to the outer loop. 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. We use a dynamic workload distribution system where each thread keeps track of the amount of work (ie, the number of sinks visited in this case) that was done, and returns that value when it has finished. We then compare the workload of two neighboring partitions, and shift the boundaries to even out the workload if there is significant imbalance. 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.

3.3.6 *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, we randomly select a neighbor from the legal swap region to be con-

Table 1: Adaptive simulated annealing schedule

Success Rate	Change in parameter	
	Temperature	region_place_count
> 0.96	old_t * 0.5	input_region_place_count
> 0.80	old_t * 0.9	input_region_place_count
> 0.15 or rlim > 1	old_t * 0.9	input_region_place_count/4
otherwise	old_t * 0.6	input_region_place_count/20

sidered 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. These 2 blocks will be assessed based on the difference in timing cost and bounding box cost and evaluated for a swap using identical functions as implemented in VPR.

3.3.7 Bounding Box Update

In the serial VPR program, bounding box update is done incrementally (except when needed to eliminate round-off error) 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 all 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 (Line 16 in Section 3.2). This is extra work that the serial VPR program does not need to execute. The bounding box calculation is parallelzed using the dynamic distribution scheme described in Section 3.3.5, and the final result calculation employs the method described in Section 3.3.1 to ensure determinism.

3.3.8 Adaptive Annealing Schedule

A slight modification from the annealing schedule of VPR is used in order to cope with the different move behavior of this new parallel placement algorithm. It allows for more iterations during high success rate phases, or equivalently, the high temperature phases. This alleviates the effect of restricted block movement. At high temperature, traditional VPR is capable of swapping two arbitrary blocks with only the *rlim* restriction on the distance between them. Initially, *rlim* encompasses the entire grid, and it gradually shrinks as the success rate drops below a certain value. To achieve the same effect while only using smaller sub-regions in our parallel VPR, the algorithm must consider more iterations so all blocks can have a chance of migrating toward any location on the grid. This inevitably leads to more iterations and hence more work than the original VPR algorithm. However, the ability to use more threads has enabled us to overcome this overhead and achieve a speedup as shown in the result section. Using this intuition, we experimentally determined the annealing schedule parameters as shown in Table 1.

3.3.9 Summary

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

- Partitioning each private region into four sub-regions and enforcing all threads to work on the same sub-region in its respective partition, thereby avoiding hard conflicts and enforcing determinism.
- Avoiding fine-grain synchronization by assessing moves using locally stored data, which may be stale.
- Limiting the range of swaps to all blocks with *rlim* capped at 20 or the sub-region boundaries to improve quality versus runtime trade-off result.
- Sequentially iterating through the grid instead of randomly selecting blocks to be swapped.
- In order to improve runtime for negligible quality loss, we modified the algorithm and only select 90% of the blocks to swap.
- More iterations are needed at higher temperature in order to allow blocks to migrate across the entire grid.

4. BENCHMARKING METHODOLOGY

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

4.1 Benchmarking Circuits

The traditional Toronto20 MCNC benchmark circuits are too small to use for parallel placement experiments and large FPGA circuits are rare. Therefore, we used the benchmarks provided with the Un/DoPack flow which are fully described in [16]. These large synthetic circuits are built using the GNL tool in hierarchical mode, using 20 subcircuits which are 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 were clustered using T-VPack 5.0.2 with 6-input lookup tables, a cluster size of 10, and a maximum of 35 inputs per cluster.

The architecture file used is obtained from the iFar repository[17], which models realistic FPGA architectures. We selected the file that matched our clustering specifications using 65nm CMOS technology.[4]

4.2 Hardware Environment

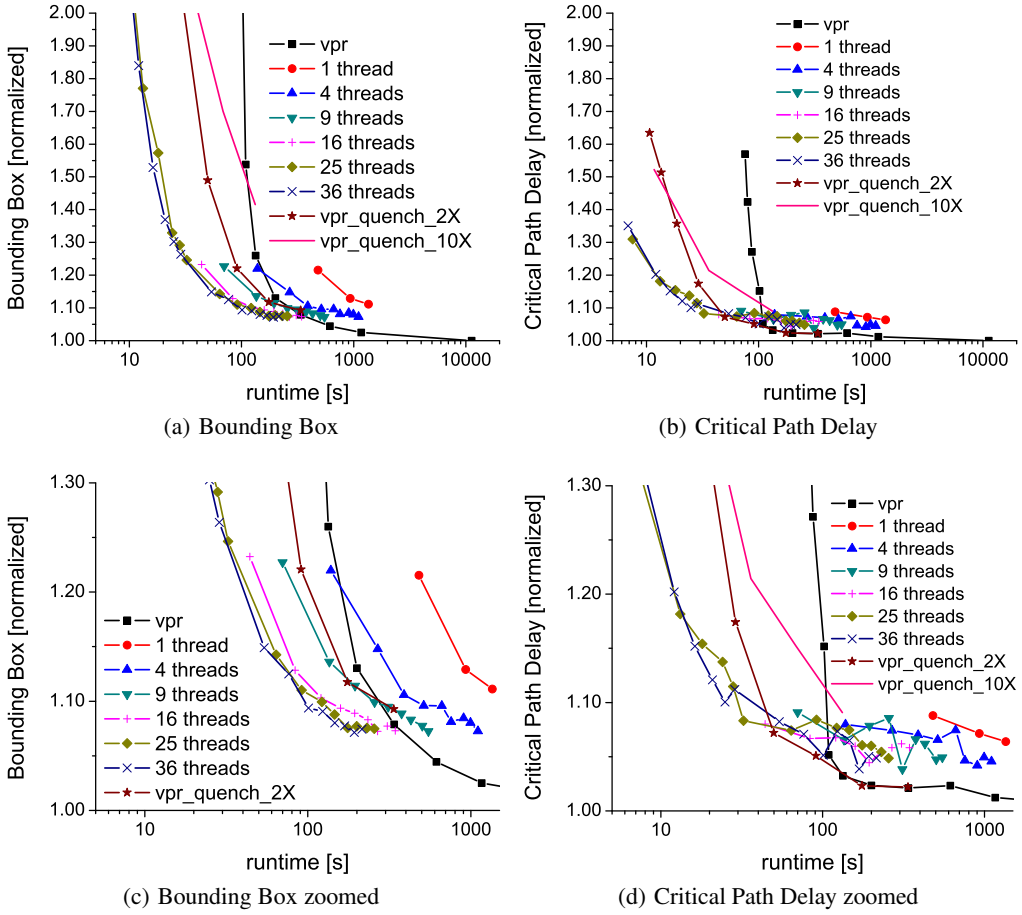
We evaluated the performance of our program using an UltraSPARC T2 (Niagara 2) machine from Sun Microsystems. It contains 8 SPARC cores, threaded 8 ways, to support a total of 64 threads, running at 1.2 GHz [18]. The system has 32GB of memory and the operating system is Sun Solaris 5.10. We also performed a number of experiments on Intel quad-core Nehalem and dual quad-core Clovertown, and AMD quad hex-core Istanbul systems. Except for Section 5.2, we only report the Sun results here to demonstrate scaling to a larger number of threads that should soon be available in future Intel/AMD systems.

4.3 Experimental Methodology

Our 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 our circuits. Our quality values in this section are compared against default VPR and not VPR '-fast'.

The runtime reported in the results (Section 5) includes placement time only, which is primarily the pseudo-code shown in Section 3.2. Time excluded from measured time, including netlist loading and precomputation of delay tables used for costing, is the same between VPR and our parallel placer. This initialization overhead

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



(a) Bounding Box (b) Critical Path Delay

(c) Bounding Box zoomed (d) Critical Path Delay zoomed

Figure 6: Runtime vs Quality (geomean of 7 benchmark circuits)

is considerable, but we were able to dramatically reduce it with a few very small changes. It can be further optimized, perhaps even parallelized, to avoid it from becoming a bottleneck. This common initialization runtime is not part of our measurement.

5. RESULTS/DISCUSSION

In this section, we compare the quality and runtime of our parallel placement algorithm against VPR. Also, we present the self-speedup of the parallel algorithm versus a single-thread implementation of the same parallel algorithm and discuss its runtime scaling limitations. Then, we show that the quality of placed circuits is preserved while scaling.

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

5.1 Quality vs Runtime

We compare the final post-placement bounding box cost and critical path delay between the two programs. These should correlate with the quality of routed result. We had insufficient time to generate and compare post-routing quality metrics in this paper.

We ran all the circuits and obtained a quality/runtime trade-off curve by varying the *region_place_count* values and the number of threads. Figure 6 shows the quality vs runtime comparison for our parallel program versus VPR. The runtime is plotted on x axis in log scale, the result is shown on the y axis, and a separate graph is shown for bounding box and critical path delay. 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.

5.1.1 *Comparison Trends*

Figure 6 shows the runtime versus quality comparison between VPR and our placer. The black line with solid square is VPR's runtime curve; the right-most data point (at ~10,000s) is VPR and the 2nd data point from the right (at ~1,000s) is VPR '-fast'.

It can be seen that the single-threaded version of the parallel algorithm is slower than VPR '-fast'. This is mainly due to the associated overheads needed to make the algorithm parallelizable. As more threads are used, our program begins to outperform VPR. The 25 threaded version is able to achieve better bounding box and critical path delay results faster than VPR.

Figures 6(c) and 6(d) shows the same graph as Figures 6(a) and 6(b) with re-adjusted x and y axes to focus on the interesting portion of the graphs. It can be seen that VPR's QoR degrades sharply at approximately 100s; our algorithm is still able to sustain a decent QoR as runtime shrinks. We improved VPR's QoR in this extremely fast runtime region by using a technique known as quenching, where the temperature is decreased rapidly so the program spends more time making good moves greedily. *vpr_quench_2x* and *vpr_quench_10x* is achieved by multiplying the temperature by 0.5 and 0.1 respectively at every temperature update, achieving a 2X and 10X speedup in cooling speed. As seen on the graph, in the extremely fast runtime region, this produced a QoR superior to standard VPR in both bounding box and critical delay quality

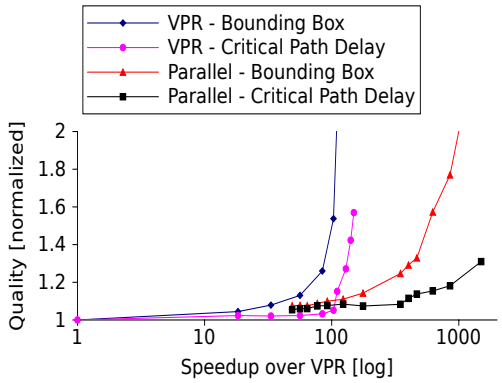


Figure 7: Quality loss from speeding up VPR and Parallel (25 threads)

Table 2: Parallel (25 threads) compared to VPR

	# luts	# clbs	bb cost	crit. delay	speedup
stdev000	40013	5036	1.09	1.11	133
stdev002	40013	5051	1.10	1.04	130
stdev004	40013	5037	1.13	1.08	121
stdev006	40013	5023	1.09	1.05	124
stdev008	40013	5041	1.11	1.06	114
stdev010	40013	5043	1.18	1.14	124
stdev012	40013	5060	1.07	1.12	116
25 threads	Geo. Mean		1.11	1.08	123
VPR-superfast	Geo. Mean		2.09	1.15	110

metrics, however, it is still inferior to our parallel program using 25 threads. This makes our algorithm an attractive candidate for users that are willing to trade some quality for extremely fast runtime. It can also be noted while the QoR for our parallel program converges, it is still unable to match the quality of VPR '-fast'. The QoR gap is approximately 3-4% on average and minimizing this gap would be an interesting future work.

5.1.2 *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 in Figure 6, we selected the first point where our parallel version with 25 threads outperforms VPR in *both* bounding box cost and critical path delay; this occurs around 85 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 a *region_place_count* value of 90. To be fair, we did not compare VPR_quench against our parallel algorithm in this section, as quenching could potentially affect the performance of our parallel algorithm as well.

At this selected point in the runtime/quality space, the parallel placer and VPR '-superfast' have similar speedups, but the parallel version beats VPR in quality. Table 2 shows the actual quality and speedups obtained of the 25 threads on each benchmark, as well as the geometric mean. In comparison, VPR '-superfast' is slightly outmatched in critical path delay (1.15 vs 1.08), and already vastly outmatched in bounding box quality (2.09 vs 1.11) by the parallel version.

A more general quality vs speedup trade-off comparison is plotted in Figure 7. It can be seen that VPR's speedup plateaus at approximately 100X, but our parallel algorithm can achieve speedups around 1000X for similar sacrifices in quality.

5.2 Runtime Scaling

Figure 9 shows the self-speedup obtained using up to 64 threads relative to the single-threaded version of the parallel algorithm with *region_place_count* value of 90. The Niagara system scales well up to 25 threads, beyond which only a minor additional speedup is observed. In this figure, we also show the self speedup up to 16 threads using a quad hex-core AMD system running at 2.4 GHz with 64 GB of system memory. It can be seen that the self speedup obtained on the AMD machine tracks Niagara curve quite well up to 16 threads. We were unable to run with more than 16 threads on the AMD system due to sharing of the machine with other users. However, this result gives us confidence to believe that speedup obtained on Niagara can also be obtained on AMD or Intel machines when the hardware becomes available.

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 (at most 4 barriers) interference and little inter-thread communication (broadcast updates at each barrier). This enables good scaling for a well load-balanced workload distribution, since all threads will progress at similar rates and they'll all arrive at the barrier at approximately the same time. These aspects of the algorithm lend themselves well to a GPU-based implementation. The other parts of our algorithm include the parallelized timing and bounding box updates. These require more barriers to enforce precedence, which may limit speedups and cause GPUs some trouble (it may have to remain on a multicore host processor). 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 contention.

5.2.1 *Runtime Breakdown*

Table 3 shows the runtime breakdown using *stdev000* circuit with *region_place_count* = 90. 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 incremental 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 3.3.7). 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 consumed doing data broadcasts at the end of each sub-region evaluation (Line 7 and 13 in Section 3.2). Finally, the barrier column shows the average time each thread spent idling at the barrier on Line 6 in Section 3.2. This quantifies load imbalance in our approach.

It is interesting to note that the single-threaded version of the parallel algorithm considered more swaps (about 20% more) than VPR '-fast', but its inner loop's runtime actually shorter than VPR. Nevertheless, after adding the parallelization overhead, the single-threaded version is slower than VPR '-fast'.

As seen in the self-speedup curves in Figure 9, the program scales quite well up to 25 threads. However, efficiency drops when scaling beyond that point. We believe the following factors contribute to this performance degradation.

First, to paraphrase Amdahl's law, the speedup is limited by the non-scalable portions of the program. The non-scalable portions consist of initialization, global to local data copy, and data broadcast. These three components make up minimal runtime (<1%) at 1 thread. However, at 25 threads, their total runtime makes up approximately 10%. Furthermore, timing update has limited scalability due to the strong data dependencies discussed in Section 3.3.5; its speedup plateaus at just 16 threads. As the number of threads increase, the size of each private region decreases, leading to less

Table 3: Runtime breakdown

# thread	initialization [s]	initialization %	timing update [s]	timing update %	bounding box update [s]	bounding box update %	global to local data copy [s]	global to local data copy %	inner loop [s]	inner loop %	data broadcast [s]	data broadcast %	barrier (line 6) [s]	barrier (line 6) %	total [s]	self speedup
VPR	–	-	73.4	1	–	-	-	-	11045	99	–	–	–	–	11183	-
VPR-fast	–	-	55.0	5	–	-	-	-	1120	94	–	–	–	–	1190	-
1	0.2	0	290.5	20	62.8	4.4	3.3	0	1067	75	8	1	0	0	1431	1
4	0.2	0	78.1	18	18.3	4.2	3.6	1	317	73	6	1	9	2	432	3
9	0.2	0	37.5	17	8.5	3.8	3.8	2	155	69	7	3	12	5	224	6
16	0.2	0	17.7	16	3.8	3.4	3.1	3	73	66	5	4	8	8	111	13
25	0.4	0	13.7	16	2.8	3.3	3.9	5	53	62	4	5	8	9	85	17
36	0.7	1	11.8	16	2.2	3.0	6.0	8	41	55	4	5	8	11	73	20
49	1.1	2	12.5	18	2.2	3.2	7.9	12	33	49	10	15	8	12	68	21
64	1.4	2	14.3	19	2.4	3.1	7.0	9	31	40	3	4	13	17	77	19

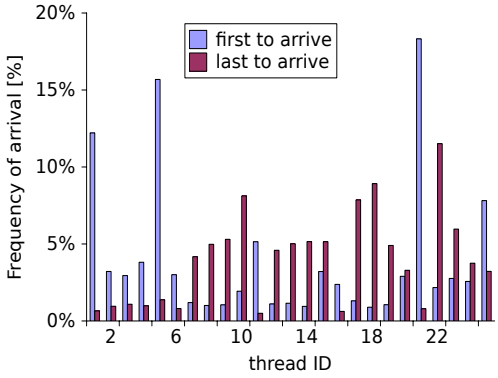


Figure 8: Probability of a region arriving first and last at a barrier due to workload imbalance

parallel work assigned to each processor. As a result, the inner loop work after being parallelized with 25 threads makes up only 62% of the runtime at this point.

Second, barriers, one of the most important mechanisms employed by our program to ensure determinism, can also be utilized to measure the amount of workload imbalance. We instrumented only the barrier on Line 6 of the pseudocode and measured the average barrier time loss due to load imbalance. Each thread ID is assigned a unique number in ascending order, starting from top left corner to the right bottom corner. We measure 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. The 25-threaded version was measured with *stdev000* circuit and the result is shown in Figure 8. It can be seen that corner regions with thread IDs 1, 5, 21 and 25 are often the first to arrive, while thread IDs in centre rows or columns are often the last to arrive. Although we attempted to assign similar work to each thread, execution time can still vary. For example, whitespace collects in the corners and requires fewer swap evaluations, while high-fanout nets may collect in the centre and need extra computation time. Although we clearly need better load balancing, we have not yet been able to improve it.

The inner loop continues to scale, though minimally, beyond 25 threads. However, since it no longer dominates the overall runtime, significant speedup is no longer achieved. We believe that increasing the amount of work allocated to each thread is necessary to obtain better speedups. Simply using bigger circuits may add more useful work to each thread, resulting in better speedups with more threads.

5.3 Quality Scaling

Figure 10 shows the quality compared to VPR using up to 64 threads for the parallel algorithm with a *region_place_count* value of 90. We can see the result is relatively constant, with a very mild

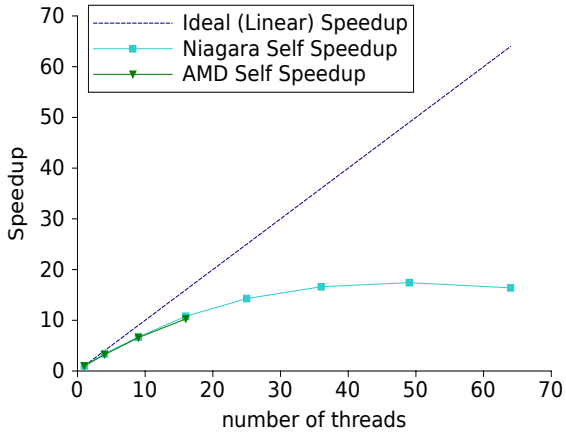


Figure 9: Self speedup

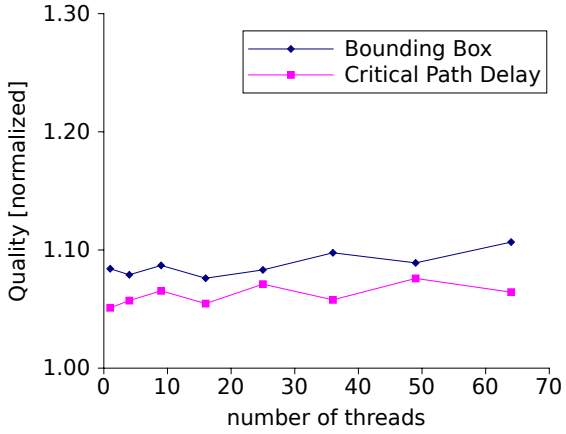


Figure 10: QoR by varying the number of threads

upward trend towards 64 threads. All quality values are within 3% of each other. This suggests that the QoR is relatively 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 49-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, limiting the staleness.

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 sub-regions also contributes to QoR and mitigates the impact of staleness. In particular, the limited range of movement within a sub-region keeps changes small relative to the length of long nets, thus mitigating the impact of stale data. Although very small nets have highest sensitivity, they often fall entirely within the current sub-region and are not subject to staleness. Slightly longer nets which extend just beyond the extended sub-region might also be sensitive to staleness. However, this is mitigated by 'buffer zone' gap of CLBs (located in between all sub-regions named A, for example) where the algorithm is not actively trying to move CLBs. This means that very short nets, where stale data has a large impact on cost, likely does not have stale data. On the other hand, medium length nets are unlikely to have their CLBs moved, and very long-length nets will have their CLBs move only a short distance.

6. FUTURE WORK

It can be seen in Figure 6(b) and the zoomed version Figure 6(d) that the QoR curve is somewhat 'twitchy', in another words, more runtime does not necessarily lead to better quality. One possible cause of the problem as explained in [19] is due to stale data information and the infrequent execution of timing analysis. More analysis could be done and implementing the incremental timing analysis update in [19] could be helpful.

Our algorithm was shown to scale up to 25 threads, but further scaling requires careful study to alleviate the bottlenecks. New data structures to support fully parallelizable timing update and 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 would be useful to remove load imbalance issues. More characterization could be done to elicit a better understanding of various tuning parameters that govern the trade-off between runtime and quality. It would also be helpful to analyze the algorithm using designs of various sizes to obtain the relationship between speedup and circuit size. Furthermore, it would be preferable to route our circuits to obtain the final quality metrics.

7. CONCLUSIONS

We presented a parallel placement algorithm that is both deterministic and timing-driven. VPR's simulated annealing can be accelerated by performing fewer moves per temperature, but quality of result degrades severely past 100X speedup. In contrast, at the point where the parallel algorithm beats VPR in quality, we achieved a speedup of 123X using 25 threads, with bounding box and critical path delay degraded by 11% and 8% respectively. While VPR cannot accelerate much beyond 100X, our parallel algorithm scales up to 1000X.

Varying the number of threads and using stale data does not have a dramatic effect on the quality of result. This is encouraging as it shows the algorithm can potentially scale to more threads without any further loss of quality. Our scaling bottleneck beyond 25 threads appears to be load balancing, but timing updates and other parallel overheads are also significant at this point. Our inner loop calculations may be suitable for GPU implementation, but timing update calculations are difficult and may need to remain on a multicore host CPU.

8. ACKNOWLEDGMENTS

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC, and WestGrid computing resources. The authors would also like to thank Tor Aamodt for interesting discussions regarding GPUs, Mark Greenstreet for allowing us to use the Niagara machine, Graeme Smecher for early advice, and Valavan Manohararajah for preliminary performance testing.

9. REFERENCES

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

[2] 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.

[3] H. Bian *et al.*, "Towards scalable placement for FPGAs," in *FPGA*, 2010, pp. 147–156.

[4] W. Swartz and C. Sechen, "New algorithms for the placement and routing of macro cells," in *ICCAD*, Nov. 1990, pp. 336 –339.

[5] A. Ludwin *et al.*, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *FPGA*, 2008, pp. 14–23.

[6] M. Haldar *et al.*, "Parallel algorithms for FPGA placement," in *GLSVLSI*, 2000, pp. 86–94.

[7] S. Kravitz and R. Rutenbar, "Placement by simulated annealing on a multiprocessor," *IEEE TCAD*, vol. 6, no. 4, pp. 534 – 549, Jul. 1987.

[8] P. Banerjee *et al.*, "Parallel simulated annealing algorithms for cell placement on hypercube multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 91–106, 1990.

[9] A. Choong *et al.*, "Parallelizing simulated annealing-based placement using GPGPU," in *FPL*, Aug. 2010, pp. 31–34.

[10] W.-J. Sun and C. Sechen, "A loosely coupled parallel algorithm for standard cell placement," in *ICCAD*, 1994, pp. 137–144.

[11] M. G. Wrighton and A. M. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *FPGA*, 2003, pp. 33–42.

[12] G. Smecher *et al.*, "Self-hosted placement for massively parallel processor arrays," in *FPT*, Dec. 2009, pp. 159 –166.

[13] J. Luu *et al.*, "VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling," in *FPGA*, 2009, pp. 133–142.

[14] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *FPL*, 1997, pp. 213–222.

[15] 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.

[16] M. Tom *et al.*, "Un/DoPack: re-clustering of large system-on-chip designs with interconnect variation for low-cost FPGAs," in *ICCAD*, Nov. 2006, pp. 680 –687.

[17] I. Kuon and J. Rose, "Area and delay trade-offs in the circuit and architecture design of FPGAs," in *FPGA*, 2008, pp. 149–158.

[18] Sun Microsystems, "UltraSPARC T2 Processor System On a Chip," http://wikis.sun.com/download/attachments/31400118/N2_Announce_Breakout_final.pdf?version=1&modificationDate=1212688201000,2007.

[19] K. Eguro and S. Hauck, "Enhancing timing-driven FPGA placement for pipelined netlists," in *DAC*, 2008, pp. 34–37.