Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 514754, 9 pages
doi:10.1155/2009/514754

Research Article

Speeding Up FPGA Placement via Partitioning and Multithreading

Cristinel Ababei

Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58108-6050, USA

Correspondence should be addressed to Cristinel Ababei, cristinel.ababei@ndsu.edu

Received 5 June 2009; Revised 15 October 2009; Accepted 12 November 2009

Recommended by Marco Platzner

One of the current main challenges of the FPGA design flow is the long processing time of the placement and routing algorithms. In this paper, we propose a hybrid parallelization technique of the simulated annealing-based placement algorithm of VPR developed in the work of Betz and Rose (1997). The proposed technique uses balanced region-based partitioning and multithreading. In the first step of this approach placement subproblems are created by partitioning and then processed concurrently by multiple worker threads that are run on multiple cores of the same processor. Our main goal is to investigate the speedup that can be achieved with this simple approach compared to previous approaches that were based on distributed computing. The new hybrid parallel placement algorithm achieves an average speedup of 2.5× using four worker threads, while the total wire length and circuit delay after routing are minimally degraded.

1. Introduction

In the field programmable gate arrays (FPGAs), design automation domain, placement, and routing are the most processing time intensive steps. The processing time problem has been somewhat alleviated by the advancements in processor speeds. However, the speedup of classic placement and routing algorithms obtained by using faster processors cannot keep pace with the rate at which the FPGA complexity increases.

The switch to parallel microprocessors represents a cornerstone in the history of computing [1], and the current trend is to continuously increase the number of cores per chip [2–7]. Despite the fact that parallel computing has been discussed for a long time [8, 9], it is still a challenging task to find the most appropriate parallelization technique and application transformation that would maximize the benefit of parallelism. Concurrency is now a pervasive topic [10] and one of the problems of parallel computing is that there are too many solutions [11]. In the FPGA domain, distributed computing and multithreading have been used as parallelization techniques for achieving efficiency. While our primary goal in this paper is to develop an efficient and practical hybrid parallel implementation of the simulated annealing placement algorithm of VPR [12], we also hope that our study will contribute toward a better understanding of the general question of which parallelization technique suits best certain placement or routing algorithms.

In the next two sections we discuss related previous work and outline our main contribution. Then, we present details of the proposed parallel implementation, which is based on mincut partitioning and multithreading. Finally, we will present our experimental results, discussions, and conclusions.

2. Related Work

There have been considerable efforts to develop parallel implementations of placement algorithms in the FPGA and VLSI domains. Routing algorithms have received less attention partly due to the fact that routing has been taking less computational time and partly due to the fact that routing is a more complex step from a parallelization perspective. Traditionally, simulated annealing (SA) has been one of the most popular placement approaches in the FPGA domain. Hence, most of the previous work has focused on parallelization techniques for SA.

From an algorithmic point of view, previous work techniques can be classified into *move acceleration* [13, 14] and *parallel moves* based [15–17] solutions. The parallel moves approaches include: (i) techniques using one copy of the main placement problem, (ii) techniques using multiple copies of the main placement problem, and (iii) techniques using placement subproblems of the main placement. The main limitations of these approaches are the degradation of the solution quality and the memory usage increase due to duplication of the database, which may lead to slowdowns. A remarkable characteristic of the techniques proposed in [14] is that serial equivalence is preserved (the parallel version of the algorithm gives the same result as the sequential version of it), which can be very useful for debugging and replication purposes.

Based on the parallel computing paradigm that is employed for parallelization, previous approaches in the FPGA domain can be classified into *distributed computing* [18, 19] and *multithreading*-based [14, 20] solutions. The distributed approach has the disadvantage of slower interprocessor communication, which can diminish the benefits of parallelization, especially for situations with significant and high-frequency intertask communication. Multithreading is using a different programming approach [21] that exploits concurrency offered by processors with multicores [2–7]. It needs only a multicore processor, which is readily available today and cheaper than a network of processors. Moreover, because all communications are within the same processor chip—via shared variables—multithreading is capable of achieving better speedups compared to those of distributed computing that can suffer from network delays. Multithreading is simple and offers an alternative implementation that complements previously proposed distributed implementations. That is, one can design a combined parallel implementation using distributed networked multicore processors, which can locally run multithreaded tasks to achieve further speedups. Distributed computing is now a more mature field [22] and has been extensively researched and used in many applications [23, 24]. Multithreading has been used also in applications such as circuit and transient simulation [25, 26].

There have been also efforts to parallelize non-SA-based placement algorithms. A parallel version of an analytical placement algorithm for FPGAs is presented in [18, 19] and is based on the negotiation paradigm. Other standard-cell parallel algorithms are reported in [27–30].

3. Contribution

In this paper, we use mincut partitioning and multithreading for speeding up the simulated annealing placement algorithm of VPR [12]. Our solution can be classified as a technique in the (iii) category (using placement region-based subproblems) discussed in the previous section. The main difference (apart from using multithreading) between our implementation and previous region-based parallel solutions [16] is the fast *mincut* balanced partitioning that we use. This minimizes the number of nets with terminals in different partitions, and therefore, minimizes the amount of dependencies between tasks. This allows us to process tasks concurrently and independently from each other. As a result, the final quality degradation is minimal with a better overall speedup. Our main goal is to analyze how much speedup can be achieved using this technique, which requires minimal change to the code base of the sequential algorithm. Legacy algorithm implementations may be very complex, and thus, parallelization approaches that minimally modify them are desirable. Our implementation is the first one, in its category, to achieve better speedup using four threads, with less quality degradation. Our algorithm is intended to provide an alternative rather than a replacement to previous approaches. To this end, the main contribution of this paper is as follows.

(i) We propose a hybrid parallelization technique based on mincut partitioning and multithreading. We use hMetis [31], one of the best publicly available partitioning tools, to divide the main placement problem into tasks processed concurrently by different threads. This approach leads to better speedups with minimal degradation of the solution quality.

(ii) We are the first to report results for the largest new benchmarks of the latest VPR 5.0 package [32].

Preliminary results of this work were reported in a poster [20]. Here, we provide the details of our implementation and report additional results on larger test cases.

4. Mincut Partitioning and Multithreading-Based Parallel Placement

First, we review the classic simulated annealing-based placement for island-style FPGAs. This will help us to better introduce our ideas later in the paper.

4.1. Classic Simulated Annealing-Based Placement. The classic simulated annealing algorithm [33] was motivated by an analogy to annealing in solids. This algorithm simulates the cooling process by gradually lowering the temperature of the system until it converges to a steady, frozen state. The major advantage of SA is the ability to avoid being trapped at local minima. It employs a random search, which accepts not only changes that decrease the objective function but also some changes that increase it.

Simulated annealing has been applied successfully to the placement of both VLSI and FPGA circuits [12, 34]. In both cases, the solution space exploration—going from one feasible solution to another—is achieved by performing *moves*. A move typically means swapping two cells or relocating only one cell. These moves are accepted with decreasing probability as the temperature is decreased gradually (Figure 1). During placement for island-style FPGAs, combinational logic blocks (CLBs) are swapped to explore new solutions. These swaps are restricted within a distance rlim between blocks. The control parameter rlim is decreased during the annealing process from a maximum value to the minimum of 1. This way blocks that are located as far

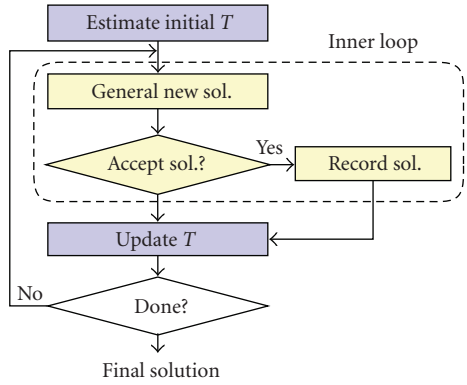

Figure 1: Block diagram of the classic simulated annealing (SA).

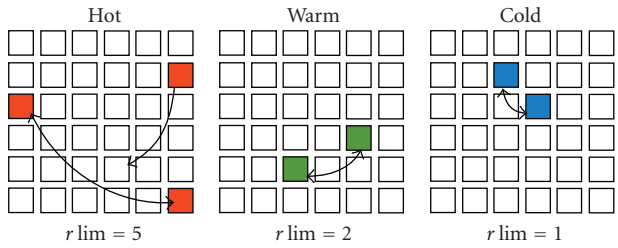

Figure 2: Illustration of how rlim controls moves inside the SA-based placement algorithm of VPR [12]. Initially, at high temperatures, blocks far away from each other can be easily swapped. Finally, at low temperatures, only blocks close to each other can be swapped.

as the entire chip width or height from each other can be swapped at the beginning of the algorithm, while toward the end of the algorithm only adjacent blocks can be swapped (Figure 2).

4.2. Parallel Simulated Annealing Placement. In this section we describe our new multithreading-based parallelization technique. The pseudocode of our algorithm is presented in Algorithm 1. The main placement problem is decomposed into multiple balanced regions using multilevel 4-way partitioning. These regions form tasks that are placed into a common queue. Then, the worker threads process these tasks in parallel and independently. The solution of each task is placed back into the corresponding task object from the queue. These results are then read in and assembled by the main manager run on the main thread. Finally, the top-level solution is further improved by an ultrafast low-temperature annealing refinement step. We use multilevel 4-way partitioning for its simplicity and because it helps in achieving balanced tasks as subproblems that resemble the original top-level problem. This allows us to reuse the same sequential annealing function for tasks processing. Next, we describe in more details the main steps of our technique.

Step 1 (Partitioning of the FPGA Chip Into Balanced Partitions). First, we perform multilevel 4-way partitioning of the top-level circuit netlist using hMetis [31]. This step

Partitioning and Multithreading-based Placement() {
(i) Step 1: Main thread
Multi level 4-way partitioning into subchips
Subchips added to queue of tasks
(ii) Step 2: Worker threads
Process tasks concurrently until queue becomes empty
(iii) Step 3: Main thread
Ultra fast top-level low temperature SA refinement
}

Algorithm 1: Pseudocode of the proposed multithreading-based parallelization algorithm.

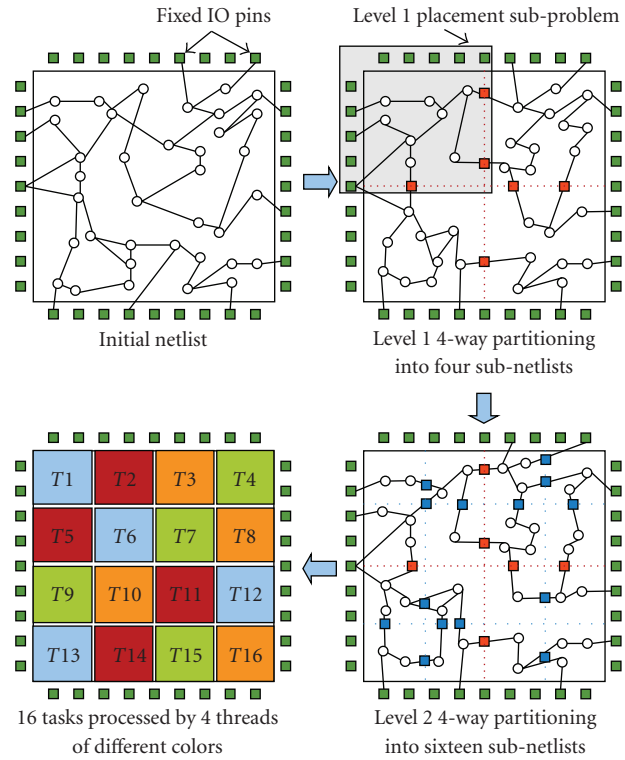

Figure 3: Illustration of Step 1. The FPGA is decomposed into multiple balanced regions using multilevel 4-way partitioning. These regions represent tasks to be processed concurrently.

is illustrated in Figure 3. Each partition is used to construct a smaller placement subproblem (that is a task), which has input/output (IO) pins of the top-level initial placement as well as *new IO* pins that account for nets which cross the partition boundaries. These nets represent the nets cut during the partitioning process and have terminals located in two or more different partitions. The IO pins of the top-level initial netlist are assigned fixed locations by VPR automatically, and they represent fixed nodes of the associated graph partitioned by the mincut hMetis. The *new IO* pins will represent *fixed anchors* (during the placement in Step 2) at the boundaries of the new placement subproblems (see right-bottom part of Figure 4). This is similar to the terminal propagation technique in standard-cell placement

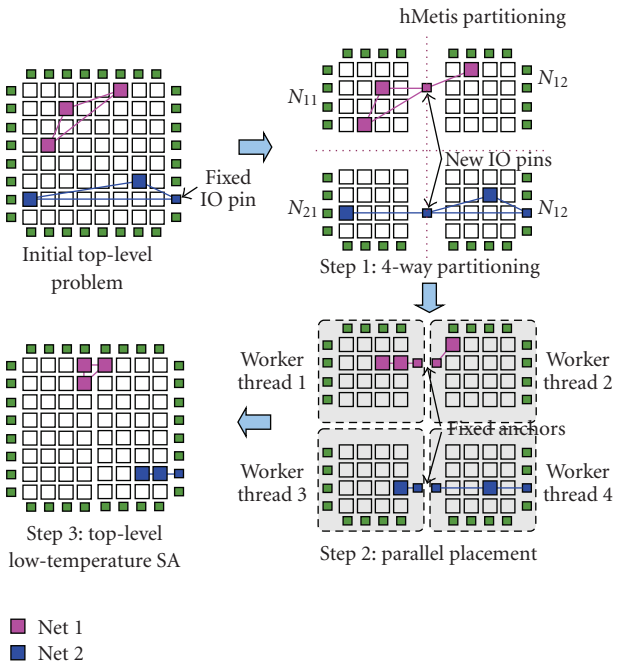

FIGURE 4: Subchip *new IO* pins are created during partitioning. They will act as fixed anchors during the parallel placement of all subchips.

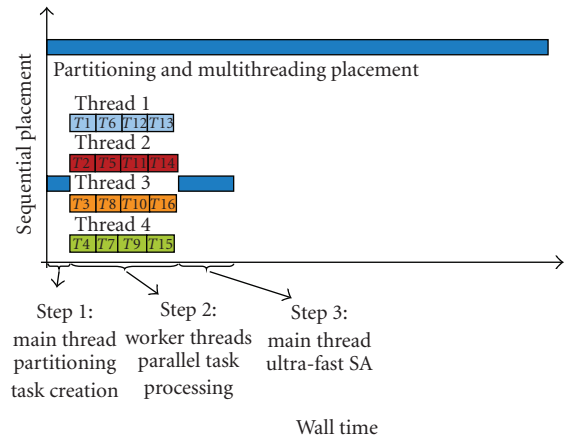

FIGURE 5: Illustration of Step 2. Sixteen tasks are processed concurrently by four worker threads (see also Figure 3). This step exploits parallelism to achieve overall speedup.

algorithms [35]. The location of these fixed anchors will be established on the fly as each subnetlist will be initially randomly placed in the corresponding region of the FPGA. For example, after the level 1 4-way partitioning, a net with terminals in the upper left and right partitions will have an anchor on the vertical partitioning boundary. The location of the anchor is the closest to the center of gravity of the net. If the net has terminals in diagonally opposite partitions, then the anchor will be located at the center. This is illustrated in Figure 4, where, for example, the net N_1 is split by the partitioning process into two subnets N_{11} and N_{12}, representing new local nets for the corresponding placement subproblems. In our experiments we also tried using floating anchors but the final quality of results was worse. We suspect that fixing the anchor points leads to better results because anchors act as attractors for terminals of the same net from different partitions to the same fixed locations. In this way, the bounding box of the top-level net will be smaller at the end of Step 2.

The use of hMetis partitioning algorithm provides a minimum number of cut nets, which translates into a reduced number of anchor points. The main benefit of this approach is that it minimizes the required synchronization between tasks and allows the threads to be run independently with a minimal negative impact on the final quality of results. The number of partitioning levels is determined by the size of the FPGA as well as the number of available worker threads to be run on different cores. If the multicore processor has, for example, four cores, then four independent worker treads can be launched and used to process concurrently four tasks.

In this case, for the example shown in Figure 3, each of the four threads will end up processing four tasks. However, increasing the number of partitions beyond the number of available cores may improve the load balancing among threads but will also result in lower quality of results due to the smaller areas in which simulated annealing will be restricted to in each task. This partitioning step is performed by the main thread, which is also responsible for the creation and launching of the worker threads, using a manager-worker multithreading strategy described in detail in [21, 36]. Because hMetis is very fast, the processing time of this step is usually less than 1% of the processing time of the sequential VPR placement tool.

Step 2 (Multithreading-Based Parallel Placement). The result of the previous step is a list of tasks stored in a *queue* data structure, where each task represents a placement subproblem corresponding to a different region of the initial top-level FPGA. Note that the task objects stored in this queue represent the so-called *shared variables* in the commonly used terminologies in [21, 36]. In the second step of our technique, the worker threads pickout and process these tasks concurrently until all tasks are exhausted. Every worker thread performs the SA annealing described in Figure 1, but this time on placement subproblems of smaller size than the size of the initial top-level FPGA placement. The result of each thread is deposited back in the task object, which is marked as *placed*. It is this step where parallelism by multithreading is exploited. This step is illustrated in Figure 5. In our implementation, during this step the main thread also retrieves the placement result from each task that is marked as being *finished*. The result of each task is copied to the top-level data structure that represents the top-level FPGA placement. That is, *the location of each block from each placement subproblem is mapped back onto the corresponding location on the main initial top-level FPGA chip*. In our experiments, the processing time of this step is usually 25% of the processing time of the sequential VPR placement tool.

Step 3 (Low-Temperature SA Refinement). In the last step of our technique, the main thread runs the ultrafast low-temperature simulated annealing. The fast cooling scheme is realized by starting with a low initial temperature of 0.1 found by a set of calibration experiments over a set of representative test cases. This initial temperature offered a good tradeoff between speedup and degradation of solution quality. The cooling scheme is also controlled by the rlim parameter (see Figure 2). The cooling rate and the number of inner loop iterations are determined inside the modified algorithm similarly to the original VPR tool. The purpose of this sequential refinement step is to further improve the solution quality by correcting the moves that could not be explored during Step 2. These moves are restricted and involve mostly blocks located alongside the partition boundaries. During this step, nets that had terminals in different partitions will have their bounding boxes minimized (see left-bottom part of Figure 4).

5. Experimental Results

We implemented our technique using C++ by changing the VPR code base, which can be downloaded from [37]. We used the hMetis partitioning tool that can be downloaded from [38]. The modified VPR tool with our implementation can be downloaded from [39]. We introduced a new option called *-mt_place [int]* which can be used in order to run our parallel implementation of the placement algorithm. *[int]* specifies the desired number of worker threads to be created and used (its default value is equal to the number of cores detected on the current processor). The other options of the modified tool are the same as those of the original VPR tool. All experiments were performed using the VPR option of fixed IO pins. All our experiments were performed on a Linux machine running on an 2.4 GHz Intel Quad processor and 2GB memory.

5.1. VPR 4.3 and VPR 5.0 Test Cases. In this section we present experimental results obtained using our modified parallel VPR tool versus the standard sequential VPR for all twenty test cases of VPR 4.3 [37] as well as for the largest eleven benchmarks included in the latest VPR 5.0 package [32]. The FPGA architecture that we used is *arch4* of the VPR package, which is used by the majority of previous works. It contains a mix of wire segments of length one, two, six, and chip-width long wires. We ran our parallel VPR placement algorithm using four threads because our processor is an Intel Quad with four cores. Because of that and because the test cases are not too big and partitions obtained with hMetis are very well balanced we used only level one 4-way partitioning. The flow diagram of our experimental setup is shown in Figure 6. Each test case is basically processed using two different design flows. In the first design flow, each test case is placed using the proposed parallel VPR placement algorithm and then routed using the timing-driven sequential VPR routing tool. In the second design flow, each test case is placed using the standard sequential VPR placement algorithm and then also routed using the timing-driven sequential VPR routing tool. During each flow, we record the CPU runtime (processing time) and the wire length (WL) after the placement step and the wire length and the circuit delay after the routing step.

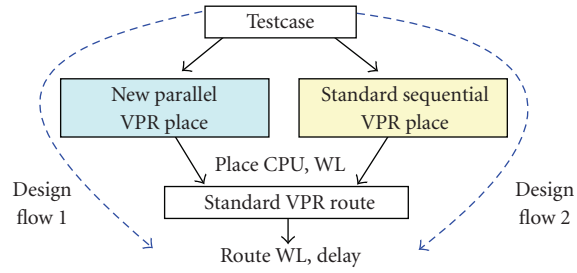

Figure 6: The flow diagram of the experimental setup with two design flows: first design flow uses the proposed parallel VPR placement algorithm and the second design flow uses the standard sequential VPR placement algorithm.

In order to have a better confidence in our results, all test cases are run four times corresponding to four different seeds of the random number generator (the seed value can be set using the available VPR options). The results are presented in Table 1 and are reported based on the averages of the four different runs. Due to space limitations, we report in Table 1 only the results obtained using the first design flow (uses the proposed parallel VPR) and the improvement in runtime or degradation in terms of place WL, route WL, and route delay compared to the results obtained using the second design flow (uses the sequential parallel VPR). Because we do four different runs for each test case in both design flows, we also report the standard deviations (as percentage % of the corresponding mean) of the placement CPU, place WL, route WL, and route delay.

We observe that our new parallel VPR tool achieves an average speedup of 2.51×. The last three columns, under the label *Degradation* [%], report the degradation (as percentages) of the WL after placement and WL and circuit delay after routing. Note that in several cases the results obtained using the proposed parallel VPR are actually improved. In such cases, the results are reported as negative percentages in Table 1. It can be noticed that the wire length after placement degraded on average with 2.89%, which translated into 2.36% degradation of the wire length and 3.2% circuit delay degradation after routing. The routing algorithm runtime remained the same.

We note an interesting trend: the speedup for individual test cases tends to increase proportionally with the circuit size. This is illustrated in Figure 7, which shows the speedup for all the test cases from Table 1. In this figure, the x-axis represents all the test cases ordered in nondecreasing order of the number of CLBs (i.e., circuit size).

5.2. Discussion. While the idea of region-based partitioning as a parallelization technique is not new, the merit of our paper consists in the improved speedup and smaller degradation of the quality of results. In this paper, we

Table 1: Comparison of sequential and parallel VPR placement algorithms. Negative numbers from the last three columns signify actual improvements and not degradations. The first twenty test cases are VPR 4.3 test cases and the remaining eleven test cases are VPR 5.0 test cases.

		Design flow 1: parallel VPR place + sequential VPR route				Improvement [×]	Degradation [%]		
Test case	No. CLBs	Place CPU [s]	Place WL/Std. dev [%]	Route WL/Std. dev [%]	Route delay/Std. dev [%]	Place CPU	Place WL	Route WL	Route delay
ex5p	1064	16.80	178/1.22%	36723/1.47%	85.48/9.24%	1.82×	3.36	1.72	9.48
apex4	1262	16.40	186/0.38%	42108/0.58%	78.81/5.74%	2.16×	1.10	−0.06	−5.52
dsip	1370	18.60	308/1.14%	44111/1.36%	47.95/8.73%	2.44×	1.81	0.53	−2.29
misex3	1397	18.40	197/2.25%	44197/0.95%	78.56/11.73%	2.17×	2.47	0.66	3.81
tseng	1407	12.60	122/1.14%	22157/0.66%	54.42/8.74%	2.52×	3.58	2.38	3.15
diffeq	1497	19.00	175/2.7%	33446/2.07%	59.85/8.12%	2.51×	4.68	3.58	0.74
alu4	1522	19.20	199/0.93%	40366/1.36%	83.52/4.99%	2.24×	3.24	2.48	9.94
des	1591	24.40	399/1.06%	78897/0.36%	93.07/4.83%	2.50×	3.03	1.01	−1.75
bigkey	1707	24.00	317/0.59%	53433/0.9%	54.74/7.56%	2.53×	1.62	2.47	−16.23
seq	1750	24.60	275/1.58%	59280/1.66%	81.97/8.01%	2.31×	4.22	2.32	3.51
apex2	1878	27.40	288/0.69%	65697/2.53%	87.90/6.79%	2.26×	4.81	2.84	−2.55
s298	1931	20.80	209/1.69%	43746/1.61%	125.23/2.3%	2.63×	2.12	0.39	−0.11
frisc	3556	65.00	602/1.65%	120264/1.27%	116.50/4.44%	2.46×	9.09	5.08	5.87
elliptic	3604	63.60	580/1.03%	109437/0.66%	123.83/9.1%	2.47×	7.53	4.85	17.18
spla	3690	66.40	667/0.62%	159518/0.92%	126.43/6.1%	2.54×	5.88	2.68	−0.03
pdc	4575	88.20	942/1.12%	213418/1.3%	161.77/5.84%	2.54×	3.96	1.41	5.15
ex1010	4598	79.60	674/1.33%	144458/2.62%	140.44/6.79%	2.77×	2.03	1.85	11.76
s38417	6406	115.00	724/1.08%	138241/1.11%	73.59/3.4%	3.07×	2.76	1.73	−13.82
s38584	6447	119.60	824/1.1%	131252/1.75%	79.90/6.07%	2.99×	2.35	3.09	2.41
clma	8383	171.20	1535/0.91%	313180/0.91%	142.96/5.46%	3.02×	4.21	2.72	12.31
c1	3439	74.00	471/2.21%	103934/0.85%	161.98/2.62%	1.99×	4.07	5.66	1.67
c2	3831	88.00	572/1.61%	101830/1.21%	82.63/5.47%	2.21×	0.83	−0.59	6.56
c3	3876	85.67	557/2.18%	105189/0.2%	182.66/3.31%	2.11×	7.35	4.72	−13.02
c4	4859	104.00	600/1.18%	124975/0.84%	331.19/11.73%	2.32×	4.09	1.89	8.24
c5	5013	113.33	667/1.19%	134341/0.33%	331.33/4.65%	2.30×	2.31	1.15	−2.96
c6	6580	164.00	808/2.7%	148621/2.87%	58.70/12.1%	2.17×	2.54	2.67	9.34
c7	9467	254.33	1326/0.52%	256394/0.48%	189.56/9.87%	2.56×	6.48	4.21	18.33
c8	10099	255.00	1778/1.2%	355124/1.5%	341.11/4.2%	2.51×	7.64	3.40	1.86
c9	20326	720.33	2856/1.18%	569767/1.37%	295.46/0.33%	2.62×	1.85	2.99	5.00
c10	59917	4028.50	5032/1.68%	1152403/1.31%	157.45/7.12%	3.45×	−22.00	1.04	18.86
c11	63722	4360.00	9095/2.5%	1650583/1.57%	372.38/3.47%	3.48×	0.51	2.37	2.44
					Avg.:	2.51×	2.89%	2.36%	3.2%

investigated the speedup achieved using mincut partitioning as opposed to direct partitioning into vertical and horizontal strips [16]. Moreover, our implementation is based on multithreading and run on the same chip rather than on distributed processors in a shared memory network architecture [16]. Our approach offers a better speedup and smaller quality degradation than previous region-based parallelization attempts. Because of the slight WL and delay degradation, our new modified VPR placement tool is intended to be used primarily for faster and better area and wire length estimations or as a faster placement solution when users are willing to sacrifice performance for runtime, as suggested in [40] and as demonstrated in Figure 8. In this figure, we plot the normalized wire length-runtime envelope curves for the old sequential and our new parallel VPR placement tools. Normalization is done with respect to the best sequential VPR wire length result, while runtime is controlled via the number of moves performed inside the inner loop of the annealing engine. Both curves contain five data points. The right most data point corresponds to the default states of the sequential and parallel VPR tools. The remaining data points, moving from right to left on both curves, are obtained by limiting the number of inner loop iterations to a fraction of only 0.5, 0.1, 0.05, and 0.001 of the number of inner loop iterations of the default state. We observe that the new modified VPR placement tool achieves better quality for a given runtime budget, and therefore, can offer more accurate and efficient estimations.

Table 2: Qualitative comparison of the proposed parallel VPR placement algorithm against previously proposed parallel placement algorithms from the FPGA domain. NA stands for not available.

	Type of algorithm	Speedup [×]	Number of cores/ machines	Solution degradation [%]	No. test cases tested	Parallelization approach	Amount of code change	Domain	Availability
This work	Region-based SA	2.5×	4	WL: 2.36% Delay: 3.2%	31	Multithreading	Medium	FPGA	Public
Ludwin et al. [14]	Move accel. SA	2.2×	4	WL: 0.0%	8	Multithreading	Large	FPGA	Commercial '08
Haldar et al. [16]	Region-based SA	1–1.5×	2–6	WL: 7%–18%	5	Distributed	Medium	FPGA	NA '00
Haldar et al. [16]	Markov chain SA	1.2–4.0×	2–6	WL: 7%–40%	5	Distributed	Large	FPGA	NA '00
Chan and Schlag [18]	Analytical	2×	3	Delay: 1%	10	Distributed	Large	FPGA	NA '07

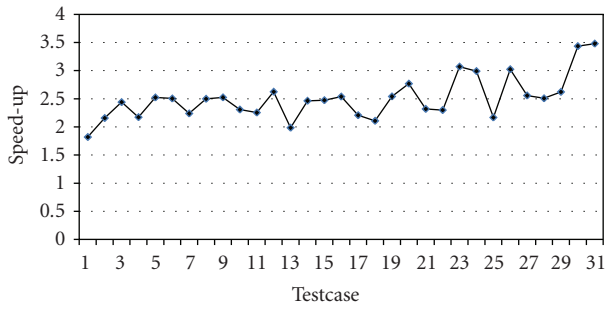

Figure 7: Illustration of the speedup variation proportional to the circuit size. The first circuit on the x-axis is the smallest and the last one is the largest.

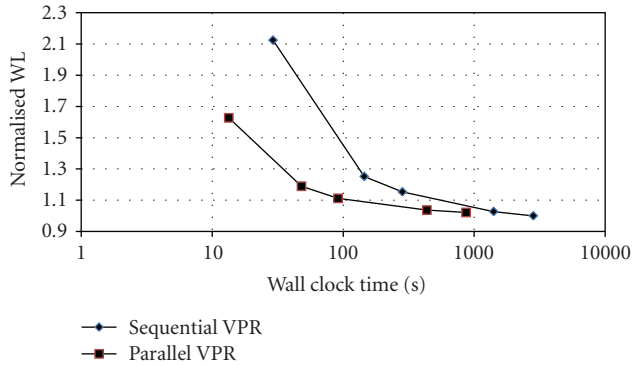

Figure 8: Placement quality-runtime envelope curves for the old sequential and new parallel VPR placement tools. Every data point is the average over all test cases.

The quality of the final placement at the end of the ultrafast low-temperature SA depends not only on the initial temperature, the maximum distance between swapped cells (i.e., rlim), and the total number of moves attempted during the inner loop but also on the quality of the starting placement. The placement achieved using the combination of hMetis mincut partitioning and simulated annealing of placement subproblems represents a high-quality starting placement for the last ultrafast low-temperature SA step. It is this combination of techniques that leads to better speedup and smaller degradation in the quality of results compared to previous similar approaches.

Our current parallel implementation is applied only to the wire length/congestion driven VPR placement algorithm. We are currently working on the timing-driven placement algorithm, which is more challenging because timing critical paths can span multiple partitions and requires synchronization between worker threads or between worker threads and the main thread in order to maintain an accurate top-level circuit delay information during the parallel processing of tasks. This additional required communication has a negative impact on the achievable speedup.

5.3. Related Work. In this section, we compare qualitatively the proposed parallel VPR placement algorithm with previous parallel implementations of placement algorithms from the FPGA domain. We cannot do a direct comparison because none of the previous implementations is publicly available. Nevertheless, because the main figures of merit for evaluation of a given parallel algorithm are the speedup and the degradation of the solution quality (compared to the sequential counterpart), which typically are reported as averages for a variety of test cases, a qualitative comparison is still possible. This comparison is presented in Table 2. We note that, among the simulated annealing-(SA) based approaches, the proposed parallel VPR placement achieves the best speedup of 2.5× on four cores. However, the proposed algorithm degrades the solution quality with 2.36% compared to the move acceleration-based SA from [14], which, however, is not as scalable, suffers from memory inefficiency, and requires considerable code change to the sequential algorithm. Among the implementations that use distributed computing as the parallelization paradigm, the analytical (not simulated annealing) placement from [18] offers one of the best speedup-solution degradation product.

6. Conclusion

In this paper, we implemented and studied a new parallelization technique for the simulated annealing-based FPGA placement algorithm of VPR. It is a hybrid technique that uses mincut partitioning and multithreading. The new parallel VPR placement tool achieves an average speedup of 2.5× using four threads on a four-core processor, while the total wire length and delay are degraded with about 3%.

Acknowledgments

This work was supported by the Electrical and Computer Engineering Department at North Dakota State University. The author would like to thank the anonymous reviewers whose suggestions have strengthened the final presentation.

References

[1] K. Asanovic, R. Bodik, B. Catanzaro, et al., "The landscape of parallel computing research: a view from Berkeley," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California at Berkeley, Berkeley, Calif, USA, 2006.

[2] C. McNairy and R. Bhatia, "Montecito: a dual-core, dual-thread itanium processor," *IEEE Micro*, vol. 25, no. 2, pp. 10–20, 2005.

[3] S. Borkar, P. Dubey, K. Kahn, et al., "Platform 2015: intel processor and platform evolution for the next decade," August 2005.

[4] "Intel Quad-Core Technology," 2006, http://www.intel.com/technology/#quad-core.

[5] J. Friedrich, B. McCredie, N. James, et al., "Design of the Power6 microprocessor," in *Proceedings of the International Solid-State Circuit Conference (ISSCC '07)*, pp. 96–97, San Francisco, Calif, USA, Febraury 2007.

[6] J. Dorsey, S. Searles, M. Ciraula, et al., "An integrated quad-core opteron processor," in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC '07)*, pp. 102–103, San Francisco, Calif, USA, Febraury 2007.

[7] UltraSPARC T2 Processor, 2007, http://www.sun.com/processors/UltraSPARC-T2/.

[8] S. Gill, "Parallel programming," *The Computer Journal*, vol. 1, no. 1, pp. 2–10, 1958.

[9] G. V. Wilson, "The history of the development of parallel computing," 1994, http://ei.cs.vt.edu/~history/Parallel.html.

[10] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, New York, NY, USA, 1999.

[11] T. R. Halfhill, "Parallel processing with CUDA," 2009, http://www.nvidia.com/content/404/nvidia.asp.

[12] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for FPGA research," in *Proceedings of the 7th International Workshop on Field Programmable Logic and Applications (FPL '97)*, pp. 213–222, London, UK, September 1997.

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

[14] A. Ludwin, V. Betz, and K. Padalia, "High-quality, deterministic parallel placement for FPGAs on commodity hardware," in *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays (FPGA '08)*, pp. 14–23, Monterey, Calif, USA, February 2008.

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

[16] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "Parallel algorithms for FPGA placement," in *Proceedings of the 10th Great Lakes Symposium on VLSI (GLSVLSI '00)*, pp. 86–94, Chicago, Ill, USA, January 2000.

[17] J. A. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee, "An evaluation of parallel simulated annealing strategies with application to standard cell placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 4, pp. 398–410, 1997.

[18] P. K. Chan and M. D. Schlag, "Parallel placement for field-programmable gate arrays," in *Proceedings of the 11th ACM/SIGDA ACM International Symposium on Field Programmable Gate Arrays (FPGA '03)*, pp. 43–50, Monterey, Calif, USA, February 2003.

[19] P. K. Chan and M. D. Schlag, "Parallel FPGA placement with symmetric multiprocessors (SMPs) and vector functional units abstract," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '07)*, Monterey, Calif, USA, February 2007.

[20] C. Ababei, "Parallel placement for FPGAs revisited," in *Proceedings of the IEEE ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '09)*, Monterey, Calif, USA, February 2009.

[21] R. H. Carver and K.-C. Tai, *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*, Wiley-Interscience, New York, NY, USA, 2005.

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, Mass, USA, 2nd edition, 1999.

[23] G. Yang, "Paraspice: a parallel circuit simulator for shared-memory multiprocessors," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pp. 400–405, Orlando, Fla, USA, June 1990.

[24] S. Markus, S. B. Kim, K. Pantazopoulos, et al., "Performance evaluation of mpi implementations and mpi based parallel ellpack solvers," in *Proceedings of the MPI Developers Conference*, pp. 162–169, Notre Dame, Ind, USA, July 1996.

[25] X. Ye, W. Dong, P. Li, and S. Nassif, "MAPS: multi-algorithm parallel circuit simulation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '08)*, pp. 73–78, Anaheim, Calif, USA, June 2008.

[26] W. Dong, P. Li, and X. Ye, "WavePipe: parallel transient simulation of analog and digital circuits on multi-core shared-memory machines," in *Proceedings of the Design Automation Conference (DAC '08)*, pp. 238–243, Anaheim, Calif, USA, June 2008.

[27] A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A parallel simulated annealing algorithm for the placement of macro-cells," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 838–847, 1987.

[28] W.-J. Sun and C. Sechen, "A loosely coupled parallel algorithm for standard cell placement," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design of Integrated Circuits and Systems (ICCAD '94)*, pp. 137–144, San Diego, Calif, USA, June 1994.

[29] T. M. Nabhan and A. Y. Zomaya, "A parallel simulated annealing algorithm with low communication overhead," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 12, pp. 1226–1233, 1995.

[30] J. Rose, W. Snelgrove, and Z. Vranesic, "Parallel standard cell placement algorithms with quality equivalent to simulated annealing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 3, pp. 387–396, 1988.

[31] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in VLSI Design," in *Proceeding of the 34th ACM/IEEE Design Automation Conference (DAC '97)*, pp. 526–529, San Diego, Calif, USA, September 1997.

[32] J. Rose, et al., VPR 5.0 benchmarks, 2009, http://www.eecg.toronto.edu/vpr/.

[33] R. A. Rutenbar, "Simulated annealing algorithms: an overview," *IEEE Circuits and Devices Magazine*, vol. 5, no. 1, pp. 19–26, 1989.

[34] T. Taghavi, X. Yang, B.-K. Choi, M. Wang, and M. Sarrafzadeh, "Dragon2005: large scale mixed-sized placement tool," in *Proceedings of the International Symposium on Physical Design (ISPD '05)*, pp. 42–47, San Francisco, Calif, USA, April 2005.

[35] A. Dunlop and B. Kernighan, "A procedure for placement of standard-cell VLSI circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, no. 1, pp. 92–98, 1985.

[36] B. Lewis and D. J. Berg, *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[37] V. Betz, VPR 4.3, 2009, http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html.

[38] G. Karypis, hMetis, 2009, http://glaros.dtc.umn.edu/gkhome/metis/hmetis/download.

[39] C. Ababei, ParallelVPR, 2009, http://venus.ece.ndsu.nodak.edu/~cris/software.html.

[40] Y. Sankar and J. Rose, "Trading quality for compile time: ultra-fast placement for FPGAs," in *Proceedings of the 7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pp. 157–166, San Diego, Calif, USA, February 1999.