A SA-based parallel method for FPGA placement

Chengyu Hu\textsuperscript{a)}, Peng Lu, Meng Yang, Jian Wang\textsuperscript{b)}, and Jinmei Lai\textsuperscript{c)}

\textit{ASIC and System State Key Laboratory, Fudan University,}
\textit{Shanghai, P. R. China}
\texttt{a) 15110720009@fudan.edu.cn}
\texttt{b) wjian@fudan.edu.cn}
\texttt{c) jmlai@fudan.edu.cn}

\textbf{Abstract:} In this paper, we present a serially-equivalent parallel method to accelerate FPGA placement. Our method is based on Simulated Annealing (SA) Algorithm: moves of placement blocks are processed concurrently on multiple threads. Two strategies are adopted here to guarantee serial equivalency: task switch of the master thread is used to handle data conflicts aroused by parallel; an efficient SA-based parallel framework is designed to obtain orderly flow of data. Our method is tested by doing placement for Xilinx xc4vlx200 FPGA chip. In a quad-core processor, a speedup of 1.8$\times$, 2.7$\times$, 3.4$\times$ is achieved on 2, 3, 4 threads. Compared to serial placer, placement results of our parallel placer are deterministic and have no quality loss.

\textbf{Keywords:} parallel, simulated annealing, placement, FPGA

\textbf{Classification:} Integrated circuits

\textbf{References}

[1] W. Li, et al.: “UTPlaceF 3.0: A parallelization framework for modern FPGA global placement: (Invited paper),” IEEE/ACM International Conference on Computer-Aided Design (2017) 922 (DOI: 10.1109/ICCAD.2017.8203879).
[2] A. Ludwin and V. Betz: “Efficient and deterministic parallel placement for FPGAs,” ACM Trans. Des. Autom. Electron. Syst. 16 (2011) 22 (DOI: 10.1145/1970353.1970355).
[3] J. B. Goeders, et al.: “Deterministic timing-driven parallel placement by simulated annealing using half-box window decomposition,” Reconfigurable Computing and FPGAs (2011) 41 (DOI: 10.1109/ReConFig.2011.27).
[4] A. Ludwin, et al.: “High-quality, deterministic parallel placement for FPGAs on commodity hardware,” Proc. 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays. ACM (2008) 14 (DOI: 10.1145/1344671.1344676).
[5] S. Kirkpatrick, et al.: “Optimization by simulated annealing,” Science 220 (1983) 671 (DOI: 10.1126/science.220.4598.671).
[6] J. Luu, et al.: “Vtr 7.0: Next generation architecture and cad system for fpgas,” ACM Transactions on Reconfigurable Technology and Systems (2014) 6 (DOI: 10.1145/2617593).
[7] S. Birk, et al.: “Parallelizing FPGA placement using transactional memory,” Field-Programmable Technology (2010) 61 (DOI: 10.1109/FPT.2010.5681538).
1 Introduction

Placement is a compute-intensive part of FPGA CAD flow. In the past several decades, the capacity of FPGA has grown fast, leading to severe runtime problem of FPGA placement [1]. Fortunately, parallelizing existing placement algorithms is developed as a promising solution [1, 2, 3, 4, 7, 8]. Among available placement algorithms, Simulated annealing (SA) [5] is most widely used in FPGA placers, such as academic tool VPR [6] and commercial tool Quartus II [2].

Many parallel methods have been investigated for SA algorithm. They can be summarized into two categories.

Methods of the first kind aim at getting good speedup and are tested in academic tool VPR. To handle data conflicts in parallel, a strategy of randomly aborting is adopted in [7], while orderly aborting is used in [8]. Paper [3] partitions whole FPGA chip into equal-sized regions and placement is executed concurrently in each region. Although these methods achieve good speedups, some features hinder their applications. Placement results of them are nondeterministic [7] or vary as processor number changes [3, 8]. Otherwise, they all suffer unpredictable quality loss.

Methods of the second kind focus on applications in real FPGA chip and has been verified in commercial CAD tool. Pipelined way is used in [4]. Repairing conflicted placement data is adopted in [2]. These two methods are serial-equivalent, which means that results of parallel placement are the same as serial placement. A best speedup of $1.3\times$ is achieved on 2 threads in [4], and a speedup of $1.3\times$, $1.7\times$, $1.4\times$ is got on 2, 4, 8 threads in [2]. Some optimization can be done to improve their speedup.

According to the description above, speedup, quality loss and determinism are key points in parallel FPGA placement. In this paper, we design our parallel method to be serial-equivalent and hope to achieve a good speedup. The property of serial equivalence can bring two merits: (1) Placement results are deterministic and independent of number of CPU cores, which bring convenience for migration and debugging of FPGA programs. (2) There is no quality loss compared to serial placement.
2 Method description

In a typical flow of SA placement, there are lots of iterations (called move) at each temperature. One move means changing a placement block from current position to another [6]. Whole process of a move can be divided into three step: proposal, evaluation, and finalization. In proposal step, a placement block is selected randomly, and one target position in certain physical range is generated. In evaluation step, value fluctuation of cost functions is calculated. If the move makes placement quality better, it will be accepted. Otherwise, accepting or not is decided by a random number. In finalization step, data of the move will be submitted to the global placement data if move is accepted and will be rolled back if not accepted.

Main idea of our parallel method is to process multiple moves on multiple threads concurrently. In order to realize serial equivalence, two requirements are necessary for parallel placement: (1) Compared to serial placement, there should be no data error or loss over entire placement process; (2) Data of each move should flow in the same order as serial placement. If whole move is parallelized directly, these requirements are difficult to satisfy. On the one hand, conflicts of moves may arouse data error; on the other hand, moves are proposed and finalized disorderly, leading to nondeterministic placement results.

Based on analysis above, we design parallel scheme as followed: different threads execute different steps of moves, the master thread is used to propose and finalize moves, while multiple child threads are used to evaluate multiple moves. According to task complexity, evaluation step takes most runtime of a move, so this scheme is efficient. Two requirements for serial equivalence are satisfied by work of the master thread. Conflicts will be detected and handled in proposal step, a strategy of no data error and loss is adopted here. As for the same data flow as serial placement, it is guaranteed by controlling proposal order of moves and finalization order of moves.

In parallel implementation, it should be avoided that data change of one move affects another move. Thus, move is designed to be individual data structure shown in Fig. 1, which consists of data, tasks, and a flag indicates whether the evaluation task is completed. Data of a move is submitted to global placement data after finalization. The evaluation-done flag of a move is set by child threads after evaluation is completed and will be checked by the master thread to judge if a move is evaluated.

![Fig. 1. Data structure of a move](image)
2.1 Overall flow
Flow of our parallel SA algorithm is designed as Fig. 2. The difference from serial SA algorithm is that the master thread and multiple child threads will work together in inner loops.

The master thread has two tasks: proposal and finalization. It proposes a move and check if the proposing move conflicts with proposed moves. If conflicts occur, it starts to finalize all proposed moves that involves in conflicts. When performing finalization, the master thread will check the evaluation-done flag in a move, and only an evaluated move can be finalized. This way can handle conflicts among moves successfully. The detail will be illustrated in section 2.2.

Child threads execute evaluation task. Each child thread will try to evaluate a move. After evaluating a move, a child thread will set the evaluation-done flag of the move.

It is worth mentioning that some containers are needed to store active moves and control data flow of moves. A SA-based parallel framework is designed to achieve this goal, the detail will be introduced in section 2.3.

2.2 Conflict detection and handling
Parallel of moves can arouse three kinds of conflicts: (1) Different threads try to move the same block to different positions; (2) Different threads try to move different blocks to the same position; (3) Different threads evaluate different moves that share the same net. The first two can lead to placement error. The last one may mistake computation of cost functions and degrading placement quality [7].

Fig. 2. Flow of parallel SA algorithm
Conflicts are detected by “tag” approach. In proposal step, this approach is used to check conflicts between the proposing move and proposed moves. When proposing a move, the master thread will tag resources occupied by the move. These resources include blocks, positions, and nets. If the master thread tries to tag a resource that has been tagged, there is a conflict. By this way, three kinds of conflicts can all be detected. Tags of a move will be cleared after finalization.

Conflicts are handled through task switch of the master thread. As shown in Fig. 3, the master thread has proposed move M0,M1,M2,M3, when it starts to propose M4, conflicts occur between M4 and M0,M2, so the master thread switches to finalize M0,M2. If a conflicted move is not evaluated, the master thread will wait for its evaluation done and finalize it then. After M0,M2 is finalized, the master thread can propose M4 successfully, then it tries to propose the next move M5. This strategy can guarantee that all proposed moves are non-conflict, so no data error will occur. Besides, when a proposing move meets conflicts, the master thread will continue its proposal after handling conflicts, so no data loss will be produced.

Proposal step can still be subdivided to improve efficiency. In parallel flow of Fig. 2, conflicts are detected after entire proposal step. If conflicts occur, the proposing move will get wrong data and need to be re-proposed, leading to unnecessary workload. As shown in Fig. 4, proposal step is subdivided into three stages to solve this problem. Conflicts are detected and handled after each stage and will not affect data of the next stage.

2.3 Parallel framework
Parallel framework of our method is shown in Fig. 5. It is designed to guarantee orderly data flow of moves and improve parallel efficiency.
Queue1 is data Queue for storing proposed moves. When the master thread proposes a move, it will check if the new move conflicts with proposed moves in Queue1. If there is no conflict, new move will be submitted to Queue1 and evaluation task will be submitted to Queue2. Otherwise, the master thread will switch to finalization, it will take moves one by one from the tail of Queue1, until all conflicted moves are finalized. If evaluation-done flag of a tail move is not set, the master thread will wait for the move to be evaluated. Due to order property of Queue1, some moves that do not involve in conflicts will also be finalized over process of conflict handling.

Queue2 is task Queue for buffering evaluation tasks. An idle child thread will get and perform an evaluation task from the tail of the Queue2. The child thread will set the evaluated-done flag of move when evaluation is done.

This parallel framework has three merits: (1) The entry and exit of Queue1 are sequential, so all data of moves are submitted to global placement data as the same order as serial placement. (2) Communication load is constant, because all threads only need to communicate with head or tail of queue. (3) Thread number only influences speedup, but does not influence placement results.

3 Experiments

3.1 Experimental requirements

We apply our parallel method in a serial SA placer, which can optimize wirelength and critical path delay [9]. As shown in Fig. 2, the serial placer is adapted into parallel placer by using multiple threads to process moves in inner loops. Placements results of serial placer and parallel placer are compared, and the data is shown in Table II and Table III. We do placement for xc4vlx200 FPGA chip, belonging to Xilinx’s Virtex 4 Series [10]. After placement, placed netlists are
imported into ISE 14.7 [11] to verify validity. Software environment is Windows 7 and hardware platform is a quad-core CPU, named Intel(R)Core(TM)i7-4790.

| Circuit name   | Blocks and nets number of each circuit |
|----------------|---------------------------------------|
|                | #IOBs  | #Slices | #DSPs | #BRAMs | #Nets  |
| BRAM_test      | 5      | 31137   | 0     | 300    | 57337  |
| Convolution    | 161    | 12841   | 48    | 21     | 26042  |
| DART           | 11     | 53792   | 0     | 240    | 104828 |
| GSM_Switch     | 98     | 52363   | 0     | 144    | 83728  |
| LU8PEEng       | 216    | 14615   | 32    | 44     | 29916  |
| Matrix         | 130    | 78769   | 32    | 258    | 153500 |
| Mux8_128bit    | 140    | 60494   | 0     | 0      | 109977 |

Benchmark circuits and corresponding resources are listed in Table I. They are elaborately selected and modified from three sources: VTR 7.0 [6], Titan [12], user circuits. Resources of them are heterogeneous and more than benchmarks in [2, 3, 4, 7, 8].

Thread optimization is used to improve parallel efficiency. From measure data of benchmarks, evaluation step takes over 95% runtime of a move. When our method is implemented in a quad-core processor, the master thread will be free most of the time, while child threads work busily. In programming implementation, if the master thread is idle, it will help performing evaluation task. After such optimization, utilization of the master thread increases.

### 3.2 Experimental results

In Table II, parallel placer is compared with serial placer, and a speedup of 1.8×, 2.7×, 3.4× is achieved on 2, 3, 4 threads. In Table III, critical path and wirelength of parallel placement results are the same as serial placement results. As for property of serial equivalence, it can be verified by placement results among different thread number.

| Circuit name   | Speedup of different thread number |
|----------------|-----------------------------------|
|                | 2 threads | 3 threads | 4 threads |
| BRAM_test      | 1.8        | 2.8        | 3.5        |
| Convolution    | 1.8        | 2.7        | 3.2        |
| DART           | 1.7        | 2.5        | 3.3        |
| GSM_Switch     | 1.9        | 2.8        | 3.2        |
| LU8PEEng       | 1.7        | 2.6        | 3.5        |
| Matrix         | 1.9        | 2.8        | 3.6        |
| Mux8_128bit    | 1.8        | 2.6        | 3.7        |
| Geomean        | 1.8        | 2.7        | 3.4        |
Compared with the first kind of methods [3, 7, 8] in Section 1, our method is serially-equivalent, thus the placement results are deterministic and have no quality loss.

Compared with prior serially-equivalent parallel method [2] (speedup is 1.3×, 1.7×, 1.4× on 2, 4, 8 threads), our method obtains a better speedup. The scalability is good within 4 threads. When thread number increase to 8, speedup of our method maintains at about 3.5×. It proves that problem of speedup degradation in [2] is modified here. This phenomenon can be explained from the parallel framework: communication load of each thread in [2] increases with total thread number, while communication load of our method is constant as explained in Section 2.2.

### Table III. Placement quality of different thread number

| Circuit name | Serial Placement | Parallel Placement |
|--------------|------------------|--------------------|
|              | WL | T_cri (ns) | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads | 2 threads | 3 threads | 4 threads |
| BRAM_test    | 16578 | 367.2 | Same | Same | Same | Same | Same | Same |
| Convolution  | 8360  | 26.4  | Same | Same | Same | Same | Same | Same |
| DART         | 46004 | 47.6  | Same | Same | Same | Same | Same | Same |
| GSM_Switch   | 32936 | 24.9  | Same | Same | Same | Same | Same | Same |
| LU8PEEng     | 9593  | 31.2  | Same | Same | Same | Same | Same | Same |
| Matrix       | 99556 | 83.0  | Same | Same | Same | Same | Same | Same |
| mux8_128bit  | 26130 | 110.0 | Same | Same | Same | Same | Same | Same |

### 4 Conclusion

A serially-equivalent parallel method for modern FPGA is introduced in this paper. We have applied it in doing placement for a commercial FPGA chip and good speedup is shown in experimental results. Compared with previous parallel method of serial equivalence [2] (best speedup is 1.7× on 4 threads), our method achieves a better speedup (best speedup is 3.4× on 4 threads). Otherwise, our method can support different placement targets and is easily implemented under different platforms.