A Hypergraph Partitioning Algorithm with Logic Replication for Multi-FPGA Systems

Haichuan Liu[*]
Peking University
Haidian Qu, Beijing Shi, China

Peidong Yang[†]
Peking University
Haidian Qu, Beijing Shi, China

Hong Zhang[‡]
Peking University
Haidian Qu, Beijing Shi, China

Abstract

As integrated circuit (IC) designs grow in scale, the limitations of single FPGA platforms necessitate the use of multi-FPGA systems. Efficient partitioning algorithms are essential for optimizing timing and resource allocation in these systems. This paper presents a novel greedy partitioning algorithm that utilizes multiple scoring functions to improve both timing and hop path efficiency. The algorithm incorporates preprocessing, coarsening, initial partitioning, uncoarsing, and adjustment steps, ensuring effective FPGA resource utilization and satisfying problem constraints. We propose a multi-phase strategy that dynamically alternates between different scoring functions to balance partition quality and hop path optimization. The results demonstrate the algorithm's superior performance, achieving faster runtime and better score compared to baseline approaches. Our work also explores the potential for future optimizations, including the integration of genetic algorithms and enhanced logic replication techniques, to further improve scalability and computational efficiency. We make the source code privately available at https://github.com/EDAGroupProject/LoReHyPar

Keywords

FPGA, Partitioning, Greedy Algorithm, Hypergraph, Logic Replication

1 Introduction

As the scale of integrated circuit (IC) designs grows in complexity, hardware simulation platforms are no longer able to simulate designs with billions of gates on a single FPGA. Thus, multi-FPGA systems are needed to handle such projects. In this context, the quality and speed of partitioning algorithms become new challenges. A good partitioning algorithm can provide a result that is favorable for timing optimization and easy for FPGA internal layout and routing within a controllable time. If a reasonable partition cannot be ensured, the circuit's operating frequency will be severely affected. At the same time, the replication of certain specific circuit logics can also help optimize the partitioning results.

The core of the problem is to model the circuit as a hypergraph, perform hypergraph partitioning under given constraints, and apply knowledge from partitioning in the course to balance timing and hop paths in order to achieve higher scores. Based on the requirements of the problem, we propose an original greedy partitioning algorithm based on multiple scoring functions. The process includes preprocessing, coarsening, initial partitioning, uncoarsing, and adjustment, which will be introduced in detail in the following chapters.

2 Problem Requirements

The problem provides the following scoring functions (where x is the hop path and y is time, in seconds):

$$\text{Score} = x \times [1 + 0.2 \times (\frac{y}{3600})] \tag{1}$$

From the function, it is clear that our optimization objectives are the hop path and time.

The problem also proposes the following constraints:

- Hop path: must not exceed the given maximum hop path.
- Time: must not exceed 1 hour.
- Memory: must not exceed 32GB.
- Threads: at most 4 threads can be used.
- Resources: the number of FPGA resources used must not exceed the given upper limit.

The specific optimization objectives and resource constraints can be referred to in the Contest Guide (http://eda.icisc.cn/).

3 Algorithm Description

3.1 Preprocessing

In the preprocessing phase, we use two methods:

- Perform multiple rounds of locality-sensitive hashing (LSH) on the nodes and merge nodes with the same hash value.
- Use the Louvain algorithm to divide the nodes into multiple communities, with subsequent coarsening processes occurring within each community.

After experimentation, we abandoned the preprocessing steps for the following reasons:

- The hashing method we used did not significantly reduce the number of nodes. Relaxing the node merging requirements would exceed many constraints, causing trouble for the subsequent steps.
- The communities obtained directly using Louvain were of unequal size, and merging only within these communities did not help in reducing the problem size. Moreover, this approach could lead to local optima, whereas global coarsening would be more effective.

[*]Haichuan Liu participated in preparing the project presentation pre-slides, completed most of the code, prepared the paper reading pre-slides, contributed to the competition defense slides, and participated in writing this report.
[†]Peidong Yang contributed to the project presentation pre-slides, participated in coding, contributed to the competition defense slides, and participated in writing this report.
[‡]Hong Zhang contributed to preparing the paper reading pre-slides and this report.

3.2 Coarsening

We use a weighted scoring function to calculate the score between pairs of nodes.

$$\mathrm{r}(u,v) := \sum_{e \in \{\mathrm{I}(v) \cap \mathrm{I}(u)\}} \frac{\omega(e)}{|e|-1}.$$

The penalty mechanism ensures that edges connecting more nodes in the network contribute less to the score, preventing the merging process from overly relying on high-weighted edges. This ensures a more balanced coarsening process, avoiding over-reliance on certain parts of the network.

We merge the node pairs with higher scores until the stopping condition is met: $c(u) + c(v) \le \kappa$, where $\kappa := \lceil \frac{c(V)}{t \cdot k} \rceil$. Here, c represents the number of nodes and various FPGA resources, while t is the constraint parameter used to adjust the looseness of resource constraints. This helps avoid excessive coarsening and ensures that we obtain a relatively good solution in the initial partition.

4 Initial Partitioning

We utilize three partitioning methods:

- **BFS**: Randomly select k nodes and assign them to k different FPGAs. A BFS is then initiated, and the reached nodes are assigned to the same FPGA.
- **Label Propagation**: Randomly select k nodes and assign them to k different FPGAs. Their τ neighbors are then assigned to the same FPGA. After several rounds of random assignment, each node is allocated to the FPGA that maximizes either the FM or Hop gain.
- **Greedy Partitioning**: For each block, the algorithm maintains a priority queue (PQ) that stores adjacent nodes, ordered by a scoring function. The algorithm iteratively selects the highest-scoring node, moves it to the corresponding block, and updates the scores of adjacent nodes.

In the greedy algorithm, we tested various scoring functions and ultimately decided to alternate between two primary functions:

- **FM Gain**: Optimizes by minimizing the cut edges.
  $g_{\mathrm{FM}}(v) = \omega\left(\{e \mid e \in \mathrm{I}(v), \Phi(e, V_i) = 1\}\right)$
  $- \omega\left(\{e \mid e \in \mathrm{I}(v), \Phi(e, V_j) = 0\}\right)$
- **Hop Gain**: Directly minimizes the hop path.
  $g_{\mathrm{hop}}(v) =$
  $\sum_{v \text{ is source of } e} \sum_{w \in e} \left(Hop[V_i][V(w)] - Hop[V_j][V(w)]\right)$
  $+ \sum_{\substack{v \text{ not source of } e \\ w \text{ is source of } e}} \left(Hop[V(w)][V_i] - Hop[V(w)][V_j]\right)$

Alternating between these two scoring functions ensures a balance between partitioning and hop path optimization. This strategy helps avoid getting stuck in local optima and leads to a better overall solution.

During the process, we perform several rounds of BFS and label propagation based on the circuit's size, keeping the best initial partition. Then, a number of greedy partitioning rounds are performed on this initial result to obtain the final partition.

After the initial partitioning, we identify nodes in the graph that exceed the maximum hop constraint, activate these nodes and their neighbors, and proceed with several rounds of greedy partitioning to achieve a valid partition.

4.1 Uncoarsing and Adjustment

After uncoarsing a node, we activate the two nodes and calculate their gain if moved to adjacent FPGAs. Then, we move the node with the highest gain to the target FPGA and activate its neighboring nodes. This process continues until there are no positive gains after $\log_2 n$ steps, and the system returns to the state with the maximum gain.

To address exponential growth in computation, we reduce the computational workload for time efficiency. Although this sacrifices some solution quality in certain cases, this trade-off has a minimal impact on the results for large circuits. We employ the following strategies:

- Uncoarse multiple nodes at once to reduce the overall number of calculations.
- Activate only 20-50% of each neighboring node.
- Further restrict activation to only the neighbors of the initially activated nodes.
- Use a priority queue to store the gain, without updating the hop gain.
- End the process immediately when negative gain occurs.

4.2 Logic Replication and Multithreading

We search for all possible logic replication schemes and replicate according to the following order:

- Based on hop gain, from largest to smallest.
- Based on remaining FPGA resources, from largest to smallest.
- Random order.

Finally, we output the result with the optimal hop path.

Our algorithm releases unused intermediate variables multiple times during function execution, compressing memory usage, allowing the parallel program to run within memory constraints.

5 Running Results

As shown in Figure 1, the runtime for each case does not exceed 10 minutes, demonstrating that our solution is very fast.

Since only the specific results of the public cases from the pre-submission are available, we have chosen the average value of the publicly available data from the pre-submission as the baseline.

In Figures 2 and 3, the orange bars represent the results of our algorithm, while the yellow bars represent the baseline results. As can be seen, our algorithm outperforms the baseline in terms of solving time for larger instances. On average, our algorithm is 3.5 times faster than the baseline, and the score improves by 59% compared to the baseline.

Figure 4 shows that our algorithm performs better on the hidden cases than on the public cases, indicating that our algorithm has strong generalization capability.

6 Conclusion and Outlook

6.1 Innovations

We optimized the greedy partitioning process using multiple scoring functions, improving the ability to approach the global optimum. We also dynamically adjusted the algorithm's steps based on the circuit scale, balancing computation time and result quality. Additionally, we used smart pointers to manage memory, automatically releasing resources when memory is close to overflow, thereby improving memory utilization.

6.2 Future Optimization Directions

Since the runtime of our program is far below the 1-hour time limit for each case, there is ample room to introduce other algorithms. We consider the following optimization approaches:

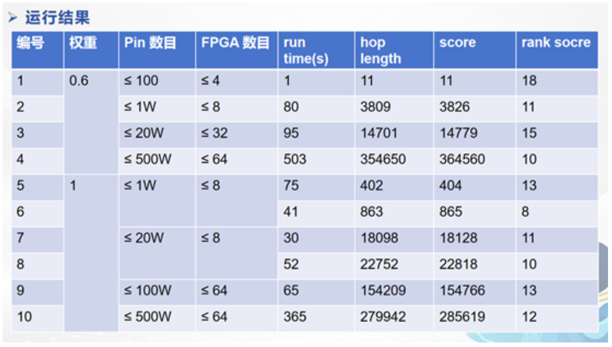
- Explore other acceleration methods: Consider implementing algorithm optimizations or introducing new techniques to improve computational speed.
- Improve the logic replication algorithm: Optimize the logic replication strategy by pre-allocating space for replication operations.
- Introduce genetic algorithms: Use historical results to optimize the search process, avoid getting stuck in local optima, and aim for the global optimum.

6.3 Reflections

In terms of data structure selection, we used the standard linked list from the C++ standard template library. However, we could improve performance by introducing a custom max-heap data structure. By using shift operations to maintain the heap structure in $O(\log n)$ time complexity, we can avoid maintaining multiple inaccurate and redundant small heaps during each operation. This approach would allow us to operate on the heap globally, enhancing performance and reducing unnecessary computational overhead.

6.4 Github Link

The github link of our project is https://github.com/EDAGroupProject/LoReHyPar/.



➤ 运行结果

编号	权重	Pin 数目	FPGA 数目	run time(s)	hop length	score	rank socre
1	0.6	≤ 100	≤ 4	1	11	11	18
2		≤ 1W	≤ 8	80	3809	3826	11
3		≤ 20W	≤ 32	95	14701	14779	15
4		≤ 500W	≤ 64	503	354650	364560	10
5	1	≤ 1W	≤ 8	75	402	404	13
6				41	863	865	8
7		≤ 20W	≤ 8	30	18098	18128	11
8				52	22752	22818	10
9		≤ 100W	≤ 64	65	154209	154766	13
10		≤ 500W	≤ 64	365	279942	285619	12

Figure 1: Runing Results

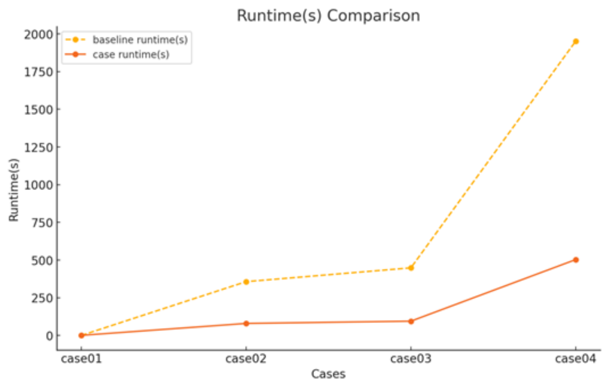


Figure 2: Runtime Comparison

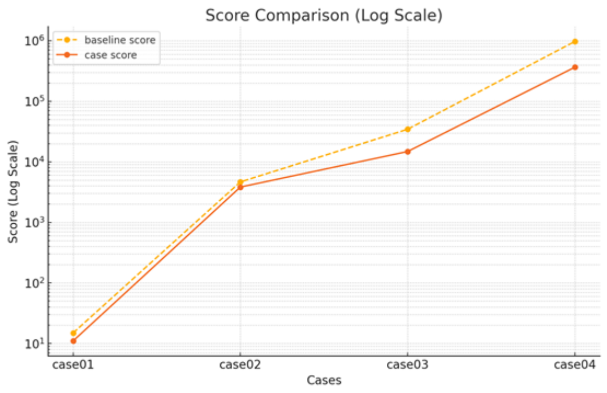


Figure 3: Score Comparison

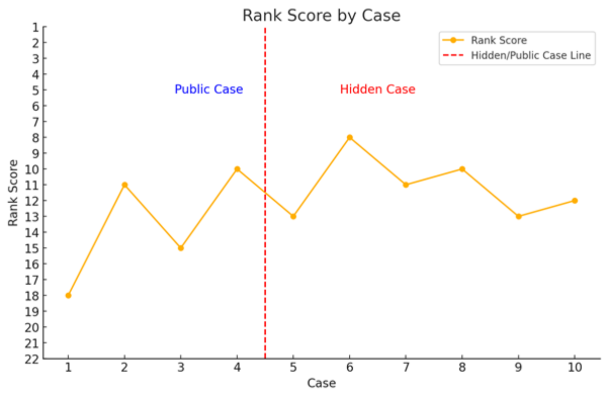


Figure 4: Ranking Comparison