[15418] Project Final Report: Parallel VLSI Partition Algorithms Based on Message Passing Model

James Wu (jhensyuw) and Yueqi Song (yueqis)

1 Summary

We implemented different parallel algorithms for graph partitioning [4] using MPI to analyze their performance in terms of solution quality, load balancing, speed enhancements, and communication overhead on PSC machines, over various different graph settings[1].

2 Background

2.1 Problem Formulation

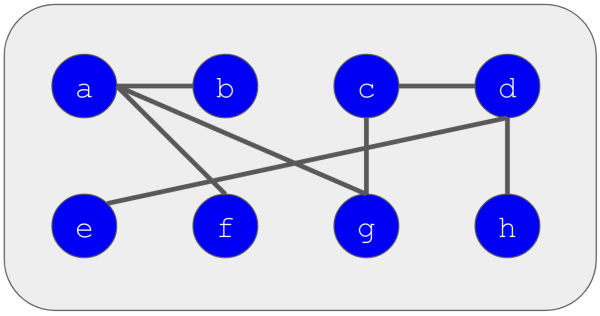

Figure 1: An example of an undirected graph structure

The problem we are solving is VLSI partitioning, an essential phase in the creation of increasingly complex integrated circuits (ICs). This process involves segmenting a larger circuit into various smaller components or modules. Such division makes the task of optimizing the circuit considerably more feasible. For example, once a circuit has been divided into parts, these individual segments can be specifically optimized to boost the overall performance of the chip, in alignment with specified design prerequisites.

This problem is a NP-hard problem, which means that the optimal solution to this problem could not be found in polynomial time; and as the problem size grows, the time needed to find the optimal solution grows exponentially. In addition, this is also a NP-complete problem, which means given a potential solution candidate, this solution could be verified in polynomial time.

Current research in this field has led to the development of numerous methodologies aimed at achieving more efficient partitioning results. These methodologies take into account a variety of cost metrics such as minimizing the length of the connections (wire length), achieving a balance in the load across different parts of the circuit, and reducing the overall power consumption.

This strategic partitioning not only streamlines the optimization process but also significantly impacts the functionality and performance of the final product. Each module can be individually tested and modified, which enhances the design's adaptability and scalability, crucial for meeting evolving technological demands.

To further illustrate the concept of VLSI partitioning, we could formulate it as a graph partitioning problem. Imagine an undirected graph `G(V, E)`, where `V` represents vertices of the graph and `E` represents edges, where both vertices and edges have weights. Graphs do not necessarily need to be fully interconnected or acyclic. Figure 1 shows an example to such a graph, with the weight of each edge and each vertex uniformly set to 1.

For parameters, we have a parameter `p`, which indicates that the vertices of graph G should be divided into `p` partitions. The objective is to distribute the weights of the vertices as evenly as possible across each partition, while simultaneously minimizing the total weight of the edges that cross between different partitions. We define the total weight of edges crossing different partitions as "cut size". We also need to satisfy the balanced condition, where one partition can contains at most `c%` of all cells for some given parameter `c%`, since without this constraint, we can simply place all cells in one

[1] All the code and implementations could be found in the Project Web Page: https://github.com/yueqis/Parallel-VLSI-Partition.

partition and get zero cut size.

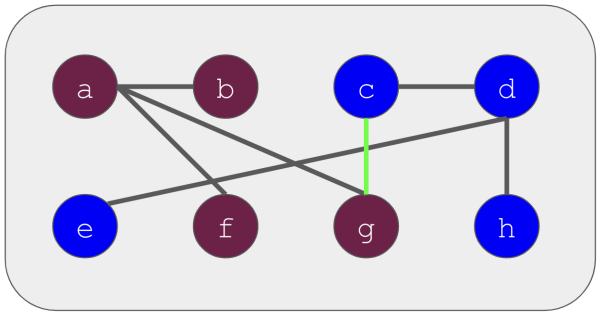

Figure 2: Optimal solution for partitioning the graph shown in Figure 1 to two partitions, resulting in a cut size of 1 and equivalent number of vertices in each partition. The green edge represent the only edge crossing the two partitions. The purple vertices represent the first partition, while the blue vertices represent the second partition.

As an example, consider the scenario described in Figure 1, where there are 8 vertices and each edge and each vertex has a unit weight. The goal is to split these vertices into 2 partitions. The optimal configuration, shown in Figure 2, results in only one edge crossing between the two partitions. In contrast, a less optimal partitioning solution, might be randomly assigning the first four vertices `a, b, c, d` to partition A while assigning the rest vertices `e, f, g, h` to partition B, which will result in a cut size of 5. This example demonstrates how strategic partitioning can significantly reduce interconnect costs and improve the overall efficiency of the system.

2.2 Inputs, Outputs, and Data Structures

Our algorithm takes lists of cells (vertices) and lists of edges as inputs. The outputs of our algorithm are the final partitions and the cut size between these partitions. The data structures utilized in our algorithm include:

- `edges` (input), which stores information about the cells involved for each edge and the weight of each edge.
- `cells` (input), which contains details about each cell, including the name of the cell, the weight of the cell, and the neighbors of the cell.
- `partitions` (output), which contains information about which partition is each cell assigned to.

We parallelize our algorithm with MPI [2], where each MPI thread operates within a partitioner module that encompasses `edges`, `cells`, `partitions`, and the parameters.

The operations performed on these data structures include the following:

- `move`, which alters a cell's partition and updates relevant data structures, including the count of cells in each partition and the distribution of edges among the partitions.
- `is_balanced`, which checks whether the current partitioning meets the specified balance constraints.
- `coarse`, which utilizes maximal matching algorithms to coarse the graph into a new graph with a smaller set of cells (vertices) and edges. This will be discussed in more details in Section 5.

Note that we only list the basic operations and data structures, as more specific ones will be presented in our approach.

2.3 Why Parallelism?

In our design, there are multiple parts that are computationally expensive and could benefit from parallelization. Parallelism offers the following benefits for VLSI partitioning [2].

Enhanced Scalability for Large Designs As IC designs become larger and more complex, traditional sequential partitioning methods struggle to keep up. Parallelism addresses this by distributing the circuit design across multiple processors, allowing for effective analysis and management of larger designs.

Large Solution Space Exploration The part that request the most extensive work in VLSI partitioning is searching the solution space, as the solution space of this task is exponentially large. VLSI partitioning benefits from parallelism by simultaneously exploring multiple regions of the solution space. Using multiple threads synchronously enables the program to explore multiple potential solutions at the same time, thus make the process of looking for the best solution much faster. Given that the solution space grows exponentially as the problem size increases (i.e., as the number of cells and number of edges increase), using a parallel approach could significantly enhance the chances of finding high-quality solutions quickly.

Increased Efficiency in Computations Parallelism enables faster computation of cost metrics and optimization processes in partitioning algorithms by utilizing the power of multiple cores. This reduces the overall runtime and improves the efficiency of VLSI partitioning tasks.

2.4 Challenges

There are a few potential challenges of developing a parallel algorithm for VLSI partitioning using MPI. In Section 3, Section 4, and Section 5, we will discuss in more details on how our solutions mitigate the challenges.

Dependency Each cell may be connected with multiple edges, and thus the solution to this task might be dependent on the interconnection of cells. This dependency makes it difficult for a parallel algorithm to work. Because placing one cell in the one partition may reduce cut sizes for some sets of edges, while increasing cut sizes for others. Between each thread, this dependency needs to be taken into consideration to reach the optimal solution.

Locality In our implementations, each cell is assigned an integer that represents the cell's relative position in the vectors that contain all cells. However, due to the dependency between edge and cells, locality will not be very good, as finding the optimal partition of a cell depend on its neighboring cells. Nevertheless, in the process of optimizing the solution, locality will be improved as our algorithm looks for the optimal solution. This is because as the program continuously minimizes its cut size, the number of edges crossing partitions will decrease, and therefore the number of memory access to cells that belong to different partitions will decrease, thus improving locality.

Load Balancing Due to the dependency between edges and cells, there does not exist a very stratightforward way to distribute equivalent amount of work to each thread. When tasks are unevenly distributed among processors, some may end up idle while others are overloaded at the same time, causing a bottleneck in the system. This may lead to less perfect speedup for the algorithm.

Synchronization/Communication Costs How to efficiently transmit necessary data in the format of sending and receiving messages between each thread could potentially affect the performance of the parallel algorithm. We need to effectively synchronize the threads by minimizing the number of transmissions and the length of transmissions, while following the constraints of dependencies of the VLSI partitioning task. In addition, if synchronization cost is too large, the effect of parallelism might be dominated by the large communication cost. This problem might be especially challenging when the number of threads is large such that the speedup provided by parallelism is offset by the high communication cost.

3 Simulated Annealing

Simulated Annealing (SA) is a common approach for addressing VLSI challenges due to its capability to manage the complex and broad solution spaces often encountered in VLSI designs. As a probabilistic technique, SA is adept at exploring a range of problem constraints, enabling it to tackle multiple conflicting objectives effectively, such as reducing interconnect costs and achieving balanced partition sizes [6]. Recognizing these strengths, we have chosen to develop a parallel version of SA.

3.1 Sequential Simulated Annealing

Our algorithm begins with a random initial partition that have evenly distributed number of cells in each partition. It also has a high initial temperature that gradually decreases over time. At each step, it randomly selects a cell and checks whether moving it to another partition can decrease the cut size. This step involves checking the information stored in `netArray` and `cellArray`, comparing the partition each cell belongs to, and evaluating how many cells are affected with this change. If this new solution does reduce the cost, we switch it to the target partition by calling `move` and update all relevant data structure, such as `netArray` and `cellArray`. However, the algorithm also accepts worse solutions with a probability that depends on the difference in the cost functions and the current temperature. This probability decreases as the temperature drops, reducing the likelihood of accepting worse solutions as the algorithm progresses. Such process is repeated until either the temperature reaches the minimum or `balcondition` returns a negative value upon balanced constraint is violated. In cases where the balance constraint is violated, we engage a `reshuffle` operation, which moves cells with smaller gains from larger to smaller partitions, even at the cost of increasing the current cut size. This

strategy helps the algorithm avoid becoming stuck in local minima early on, and directs it toward refining around a global minimum as the temperature lowers.

3.2 Parallel Version 1: Synchronous Broadcast with Message-free Shuffling

Each thread has its address space for a `partitioner`, which includes a `netArray` and `cellArray`. During each iteration of simulated annealing, threads are allocated an equal portion of cells to work with. It only checks the information based on its `cellArray` and `netArray`, so `move` and `balcondition` are limited to that thread's changes to the data structure. At the end of the iteration, each thread tries to broadcast its updates to all other threads, which triggers the operation, `synccells`. Following the approach that avoids deadlock in our lectures, threads with odd indexes are set up to receive updates first, while those with even indexes transmit their data initially. Subsequently, the roles reverse: threads with odd indexes send updates, and those with even indexes receive them. The sent messages are the number of total moved cells and their IDs and corresponding partitions.

In addition, we also redistribute the cells each thread is responsible for every iteration.The reason is that we observe simulated annealing gets stuck to a local optimal solution when each thread always traverse some subsets of cells in a specific order. To implement this shuffling without the need for inter-thread message synchronization, we assign a fixed random seed to each thread. Each thread then shuffles its cells based on its process ID. This strategy ensures that each thread examines a different set of cells in every iteration while guaranteeing that all cells are evaluated by a thread. This approach helps enhance the diversity of the search space explored by the algorithm, potentially leading to better optimization outcomes.

This method ensures an organized exchange of updates between threads to maintain consistency across the system. However, the runtime is high as expected because synchronous send/receive can lead to idle CPU time. The sender might get stuck waiting for the system to handle the message, especially if the receiver is slow or not ready to receive the data. Therefore, we substitute our method with the following asynchronous broadcast.

3.3 Parallel Version 2: Asynchronous Broadcast Ignoring Outdated Updates

Instead of waiting for the communication, each thread can perform other tasks, such as updating its own data structure based on other threads' updates, while the communication system handles the message from other threads in the background. Each thread still tries to broadcast its updates to all other threads.

In addition to simply changing to our known asynchronous broadcast, we've introduced several modifications. Each thread is now equipped with its own timer. Upon completing the dispatch of updates to other threads, a thread checks for incoming updates. If updates are received, it updates its `cellArray` and `netArray`. If updates are received, the thread will wait until the timer expires and will proceed with the next iteration of simulated annealing. Note that some communications might remain incomplete as each thread moves forward. In subsequent iterations, threads will discard any outdated updates and will reset their timers. This is achieved by attaching a tag to each message, which identifies the specific iteration of simulated annealing the update corresponds to. We have also established a time limit across all threads, although this might result in some threads completing more simulated annealing iterations than others, depending on their individual processing speeds and the timing of their communications.

This strategy is to reduce communication costs by sacrificing the performance of solutions. Because the staleness (cells that are not updated because communication takes too much time) may accumulate in each thread. This can lead to suboptimal decision-making and inefficiencies in the convergence toward a global optimum. However, the final solutions only degrade slightly or even remain similar values compared with the previous parallel version. One reason is that we have introduced random shuffling in each SA iteration so that the probability of one cell which is always not updated because of delayed communication is pretty low. Another reason is that after many iterations, simulated annealing still can find a good solution if the abundant possibilities of partition permutations are traversed.

It is worth to mention the importance of the careful selection of timer duration of each thread's timer. Setting the timer too long results in excessive waiting times for messages from other threads,

leading to higher communication costs. By contrast, if the timer is set too short, the thread may consistently miss updates from others. To effectively address this issue, we dynamically adjust the timer duration based on the size of the problem, ensuring that the timing is proportional and optimized for the specific workload each thread handles. This approach helps balance the need for timely updates with the efficiency of thread execution problem size.

3.4 Parallel Version 3: Clustered-based Message Passing with Preprocessing

To minimize inherent communication costs in our system, we've implemented a strategy where each thread sends messages only within a defined cluster during each SA iteration, instead of broadcasting to all threads. A cluster is defined as a group of threads whose cells are interconnected, while cells belonging to different clusters have no connections.

Our algorithm employs a preprocessing step to group interconnected cells into the same cluster and assigns them to corresponding threads. Thus, during each SA iteration, communications are only within-clustered. However, to satisfy the balance constraint, we perform broadcasts across all clusters occasionally every some number of iterations.

The preprocessing algorithm is as follows. Each thread can only be assigned a fixed number of cells. For each cell, we perform a Breadth-First Search (BFS) based on its edges. The cell will be assigned to a thread whose current cells are directly connected to it. If there are not other connected cells, we consider all threads allocated to be a cluster.

The rationale behind this approach is to reduce unnecessary message exchanges between cells that are not connected because such communications do not contribute to minimizing the cutsize. Since a cell's dependencies are limited to other cells it connects with through an edge, we can effectively create several disjoint sets of cell clusters based on the edge connections. Updates sent within these clusters are sufficient for reaching an optimal solution without the interference of unrelated cell information. The primary exception requiring broadcast is due to the balance constraint, where each cluster must be aware of the cells in other clusters to prevent assigning to many cells in one partition.

Unfortunately, this cluster-based approach may fail when cells are densely connected. The preprocessing algorithm will lead to a single cluster containing all cells in this case. Furthermore, some clusters may contain many cells, while the others contain few cells. This implies the potential poor load-balancing issue between threads.

3.5 Limitations

Despite developing various versions of parallel algorithm that minimize communication costs, we have not achieved significant speed improvements with simulated annealing as the number of threads increases. The main challenge is to achieve good load balancing. It is difficult to evenly distribute tasks among threads based on our problem's formulation. Our current strategy of assigning an equal number of cells to each thread appears sound for load balancing. However, the load can be not well-balanced because some cells are connected to a significantly higher number of edges than others. The interconnection of cells between edge connections makes the task distribution more complicated, leading to uneven workloads among threads and causing some threads to be idle. This difference in task allocation results in poor speedup. Therefore, we decide to develop another multilevel approach in Section 5 that alleviates this problem.

4 Kernighan–Lin Partitioning

The Kernighan–Lin algorithm is a well-known method in graph partitioning and network modeling because of its ability to effectively handle complex and large solution spaces often encountered in these areas. This iterative approach is adept at effectively finding an optimized partitioning, while maintaining equivalent number of cells for each partition. Additionally, the Kernighan–Lin algorithm is a heuristic algorithm, trading in some optimality for time complexity. In other words, finding the most optimal solution to the VLSI problem requires exponential time, while the Kernighan–Lin algorithm provide a good locally optimal solution in $O(n^3)$ where n represents the number of cells in the graph. Researchers have worked to enhance this algorithm by incorporating parallel computing techniques to further enhance its efficiency and scalability, leveraging its proven capabilities; or have advanced upon this approach, further improving performance of such algorithms [1, 3, 5].

4.1 Sequential Kernighan–Lin Partitioning

Our implementation of the Kernighan–Lin algorithm first assigns equivalent number of cell to each of the partitions by random.

Then, at each step, our algorithm computes the *advantage* or the *gain* of switching two cells from each of the partition. For example, switching cell a with cell b in Figure 2 will result in no gain and no loss, since cell a and cell b are in the same partition; while switching cell c with cell g will result in a negative gain of 2, since cell c would be switched to the purple partition and cell g would be switched to the blue partition. A positive gain indicates that the number of edges crossing different partitions would decrease after switching, while a negative gain indicates that the number of edges crossing different partitions would increase after switching. After computing the gains of all possible pairs of nodes that might be switched, the Kernighan–Lin algorithm switches two nodes with the largest gain, if such a gain is positive.

Ultimately, the Kernighan–Lin algorithm iteratively computes the gains and switch nodes, until there is no such switching possible. In other words, the Kernighan–Lin algorithm terminates when all gains are negative.

Given that at each iterative step, two cells are switched from each other's partition, the number of cells in each partition will be fixed, thus the constraint that the partitioning should be balanced with similar number of cells in each partition is always met.

For implementation, we use cpp to implement the algorithms. For the parallel algorithms described below, we use cpp to write code, use MPI to perform parallelism and message passing, and test on PSC machines[2].

4.2 Parallel Version 1: Isolated Batch Assignment with Synchronous Broadcast

Each thread is assigned an equivalent number of cells that the thread is in charge of switching the cells assigned to it. A thread will look at the gain of *switching each of its assigned cell with the other cells it is assigned to*, and record the switches where the gain is positive. Once that a thread perform a switch, it will broadcast the result of this switch to all other threads. In other words, it will broadcast the indexes and the new partition of the two cells it switched. After sending the message, each thread will also receive the switch from other threads, and then update its partition vectors based on the switches from other threads.

However, we found a critical issue with this parallel version. We observed that firstly, the cut size is much worse than the sequential version; secondly and surprisingly, we observed that the average[3] speedup of using 2 threads is more than *100* times compared to the sequential version. This is because that in the sequential version, switches could happen between any pairs of cells; while in this parallel version, for each thread, switches could only happen on pairs of cells assigned to the same thread. Given this issue, in this parallel version, the cut size is very not optimal, and the size of the search space and problem size is decreased to be much smaller than that of the sequential version. Therefore, we move on developing another way of parallelizing the algorithm.

4.3 Parallel Version 2: Non-Isolated Batch Assignment with Synchronous Broadcast

To optimize the problem described in Section 4.2 and minimize the cut size of the solution, we performed another way of parallelizing the algorithm. Here, each thread is assigned an equivalent number of cells that the thread is in charge of switching the cells assigned to it. A thread will look at the gain of *switching each of its assigned cell with the any other cells in the graph*, and record the switches where the gain is positive. Once that a thread perform a switch, it will broadcast the result of this switch to all other threads. In other words, it will broadcast the indexes and the new partition of the two cells it switched. After sending the message, each thread will also receive the switch from other threads, and then update its partition vectors based on the switches from other threads.

Using this method of parallelizing the algorithm, the cut size is much better and comparable with the sequential version. This is because the search space and problem size is equivalent to the sequential version now.

However, we found another critical issue with this parallel version. We observed that the speedup drops dramatically as the number of thread increases. We recorded the time spent on each parts of the parallel algorithm, and found that sending and receiving messages takes the most time. This observation lead us to wonder: is synchronization between threads necessary during the process of searching the solution space? In other words, would the cut size get worse significantly if threads

[2]For implementatation, we studied the pseudocode from https://patterns.eecs.berkeley.edu/?pageid=5712Kernighan-Lin-Algorithm.

[3]Averaging over all test samples we performed.

don't synchronize with each other during the process of searching the solution space? These questions motivated us to experiment with the third parallel version.

4.4 Parallel Version 3: Non-Isolated Batch Assignment with Master Slave Message Passing

To optimize the problem described in Section 4.3, improving speedup of the algorithm while maintaining a relatively good cut size, we performed the third parallelization of the algorithm. Same with the initial methodologies in Section 4.3, each thread is assigned an equivalent number of cells that the thread is in charge of switching the cells assigned to it. A thread will look at the gain of *switching each of its assigned cell with the any other cells in the graph*, and record the switches where the gain is positive. Different from the message passing approach in Section 4.4, after a thread performs a switch, it will *not* broadcast the result of this switch to other threads, and it will also *not* receive switch updates from other threads. Instead, this parallelization approach tolerate some staleness, and synchronization happens at the end of the local optimization process of each thread. In other words, after each thread finishes switching, it will reach the locally optimal status, which means no other cells assigned to the thread could be switched to further locally optimize the partition. Then, each thread will send its partitioning to the master thread (which in most case is the thread with pid 0), to synchronize the partition from all threads. To gather the results from all threads, for each cell we check whether it has the same partitioning among all threads. If it has the same partitioning among all threads, then put the cell in that partition, if it does not, then put the cell in the partition that will result in the best cut size.

Using this approach, computation speedup is improved for large number of threads compared to parallel version 2, since communication cost is reduced significantly. Additionally, since local optimal is achieved every time, this approach still achieves comparable cut size with the sequential version.

Comparing to SA, this approach achieves much better computation speedup, since the communication cost of this approach is not very high. Additionally, this approach has better load balancing than the SA approach. This is because each thread is in charge of equivalent amount of cells, while for each cell assigned to a thread, the thread needs to look at all other cells for potential switch. Although the master thread is in charge of integrating the result from other threads, the time spent on integration is only a relatively small fraction of computation time. Therefore, the workload is balanced across threads. We will discuss the results of this approach in more details in Section 6.

4.5 Limitations

As discussed previously, graph partitioning is NP-hard, which means finding the most optimal solution takes exponential time. Given this, the Kernighan–Lin algorithm is a heuristic algorithm, trading in optimality of the solution for time complexity. In other words, finding the most optimal solution to the VLSI problem requires exponential time, while the Kernighan–Lin algorithm solve the task in polynomial time, providing a good quality locally optimal solution.

Additionally, since this algorithm runs in $O(n^3)$ time complexity, it's sequential version runs with higher computation time compared to SA. However, if taken into account the much better cut size, speedup, and computation time given higher number of threads, this approach seems to be a nice solution.

Moreover, graphs exhibit a high degree of dependencies, as each cell may have multiple connections through edges. Therefore, solving this task may heavily rely on the interconnections among cells. In our implementation, each cell is assigned an integer that represents its position relative to all cells stored in vectors. However, due to the dependencies between edges and cells, locality is not optimal, as finding the best partition for a cell depends on its neighboring cells. Nevertheless, as our algorithm optimizes the solution, locality improves. This improvement occurs because the program continuously minimizes its cut size, reducing the number of edges crossing partitions. Consequently, the program accesses fewer cells belonging to different partitions, enhancing locality over time.

But, can we optimize dependencies more? We tried reassigning the position of the cells in our `cells` vector, such that we try to put a neighbor beside a cell. Although this approach did not help in improving speedup, it motivated us to come up with our next approach to further improve dependencies.

5 Multilevel Approach

To further improve dependencies of our algorithm, we learn from the paper on Multilevel Graph Partitioning [3]. This approach incorporates the approach in Section 4. We will be mainly focusing on testing this approach in the results and analysis in Section 6.

In this approach, we follow three steps:

- Coarse the graph
- Perform the Kernighan–Lin Algorithm on the coarsed graph
- Uncoarse the graph

5.1 Coarse

We perform maximal matching in coarsening the graph. Each edge in the matching is coarsed into a cell in the new graph, such that the number of cells and edges are reduced. If two cells that are both connected to another cell are combined, then we merge their edges as well. In this case, the average number of edges each cell is connected with is reduced, since edges from different cells might be combined after coarsening. Thus, the dependency is improved, as the number of edges and number of cells are both reduced, leading to less complex dependencies. In the most optimal scenario, the number of cell is reduced by 2 times.

5.2 Kernighan–Lin on Coarsed Graph

For the second step of periodically performing the Kernighan–Lin Algorithm on the coarsed graph to further refine the partition, we follow the same parallel algorithm described in Section 4.4. As discussed in Section 4.4, workload is balanced using our parallelization; and as discussed in Section 5.1, dependencies are improved. Therefore, this approach offers good load balancing and less complex dependencies. Additionally, since the graph is much smaller in this approach, the total computation time is significantly reduced.

5.3 Uncoarse

To uncoarse the graph, we simply take the coarse partitioning from Section 5.2, and iterate over all cells, is the cell belongs to a partition in the coarse setting, then also assign this cell to the same partition.

5.4 Limitations

The limitations of trading optimality for time complexity in the KL algorithm mostly apply to this algorithm.

Since the graph is coarsed, the solution quality of the partitioning result is not as good as in Section 4. This problem is especially obvious for smaller graphs. However, using maximal matching in the coarse phase maximally mitigate this disadvantage.

6 Result

6.1 Experimental Setup

The technologies we use are mainly MPI and C++. We utilize Python for some tasks, such as converting input formats and generating test cases. For debugging, we employed the GHC machine, while the PSC machine was used to gather experimental results. We incorporated some starter code from a previous assignment that includes basic parsing and a data structure for storing circuit information*
.

6.2 Benchmark

We are going to evaluate our algorithms in two kinds of comparison metrics. One metric is input size, where there are small, medium, and large cases. This indicates the number of edges given number of cells. The other metric considers two distinct types of circuit layouts. This categorization is because that the degree of dependency between cells and edges varies significantly. The first type features a uniform distribution of cells, where cells are randomly generated with a given number of edges. The second type, a dense distribution of cells, is based on Problem 3 from the 2001 CAD Contest. This configuration is characterized by many mesh-like edges in which more than two cells are fully interconnected. Each cell may have very different number of interconnections, too.

Table 1 shows the details of these benchmark.

6.3 Computation Speedup based on Input Size

The definition of computation time is strictly the time to compute the result. We start timing when the main computation starts (after all the processes have been created), and finish when all of the results have been calculated. The baseline will be the single-threaded CPU code of the corresponding approach.

We present the computation speedup of 6 cases with different cell number and edge number based

	Cell Number	Edge Number	Type
small_3000	3000	3000	uniform
small_7000	7000	7000	uniform
medium_3000	3000	5000	uniform
medium_7000	7000	10000	uniform
large_3000	3000	10000	uniform
large_7000	7000	20000	uniform
sparse_3000	3000	10000	uniform
sparse_7000	7000	20000	uniform
dense_3000	3000	10000	dense
dense_7000	7000	20000	dense

Table 1: Experimental Benchmark

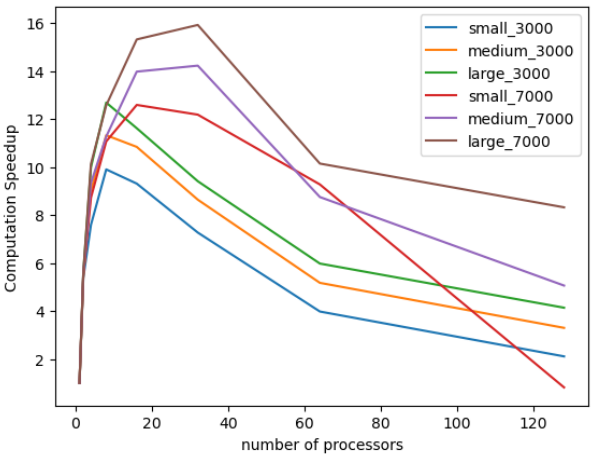

Figure 3: Computation Speedup of SA approach

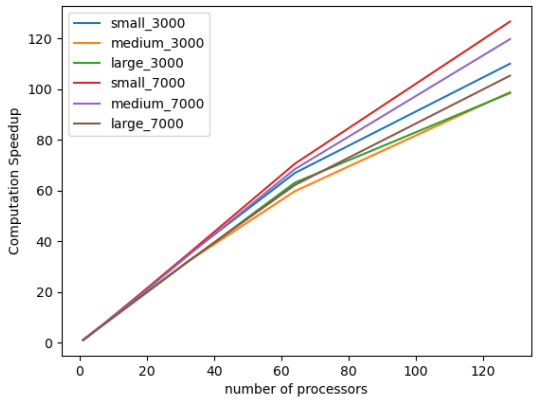

Figure 4: Computation Speedup of Multilevel approach

on both approach in Figure 3 and Figure 4.

The computation speedup of SA approach is presented in Figure 3. we can see that the speedup is nearly linear when the number of threads is less than 16. It increases slowly with 32 threads but begins to decline when the number of threads becomes excessive. The reasons that limit the speedup are as follows.

- **High Synchronization Costs**: In our parallel SA algorithm, threads exchange partition information via communication operations including sends, receives, and broadcasts. As the number of threads grows, both the complexity and volume of these communications tend to increase significantly. For example, all threads must synchronize, waiting for every other threads to update the information about the cells it moved. This synchronization can cause substantial delays in our simulated annealing approach.

- **Imbalanced Loads**: As mentioned in the previous section, it is challenging to distribute tasks among threads evenly. Our existing approach, which involves assigning an equal number of cells to each thread, initially seemed promising for balancing the load. However, the balance is often imperfect because certain cells are connected to many more edges than others. Additionally, it is harder to decompose the complex interconnections of edges and cells into finer portions as thread number becomes really large. This leads to unequal workloads across threads and leaves some threads idle for the SA approach.

In Figure 3, we can also observe that smaller test cases (3000 cells) have poor speedup compared with larger ones (7000 cells). The speedup limit is because that finer granularity of problem size often leads to lower efficiency when each thread is not doing enough computation for partitioning to justify the cost of communication and synchronization between circuit information updates. Furthermore, the computational workload is relatively light, meaning that the time spent on communication can be comparable to or even exceed the time spent on computation. This disproportionate communication overhead can severely limit the speedup that can be achieved.

In Figure 4, the speedup of the multilevel approach is very close to linear as the number of thread increases. The speedup ranges from 90 to 120 in the case of 128 threads. The reason why multilevel approach outperforms SA approach are as follows.

- **Reduced Communication Overhead**: By minimizing inter-thread communication during the local optimization phase, the multilevel approach decreases the time spent on partition updates. As a result, threads spend

more time computing the data required for KL algorithm and less time waiting on updates from other threads, which can be an obstacle of speedup. Furthermore, synchronization among threads only occurs at the end of the local optimization processes. This delayed synchronization prevents frequent interruptions in computation due to updates.

- **Lower Dependency**: The strategy of coarsing the graph can reduce the dependency of cells and edges as mentioned in Section 5. This makes it easier as we distribute tasks to each thread. Because we can divide the circuits easily when the interconnections between are not that complicated. A good task assignment at the beginning leads to good load balancing, allowing the speedup to grow linearly as thread number increases.

6.4 Total Speedup based on Input Size

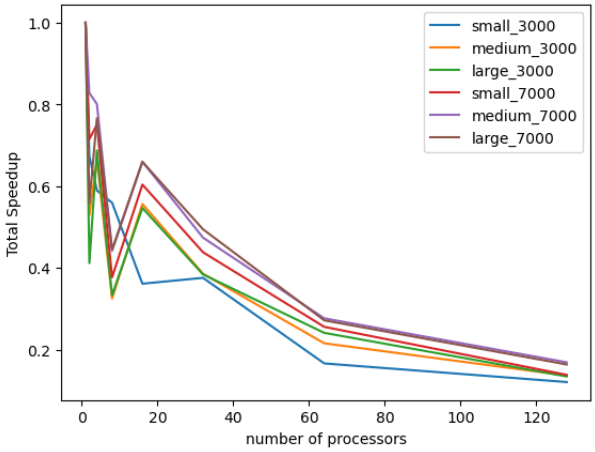

Figure 5: Total Speedup of SA approach

The total speedup here considers both initialization time (MPI threads creation, initial setup...) and computation time for our algorithm.

We present the total speedup of 6 cases with different cell number and edge number based on both approach in Figure 5 and Figure 6.

In Figure 5, we can observe that the total speedup of SA approach is decreasing as the thread number increases. The potential reason is that the problem size is too small. In the lecture, we learnt that sometimes when problems are too small, parallelism overheads may dominate parallelism benefits. The communication-to-computation ratio is too large for the machine. Since there is a significant overhead associated with communication, including initializing the MPI environment and setting up communications between threads. In smaller test cases, the computational workload is relatively light, and thus, the time spent on communication can be comparable to or even exceed the time spent on computation. We should test larger test cases for the SA approach to determine whether this conjecture is true.

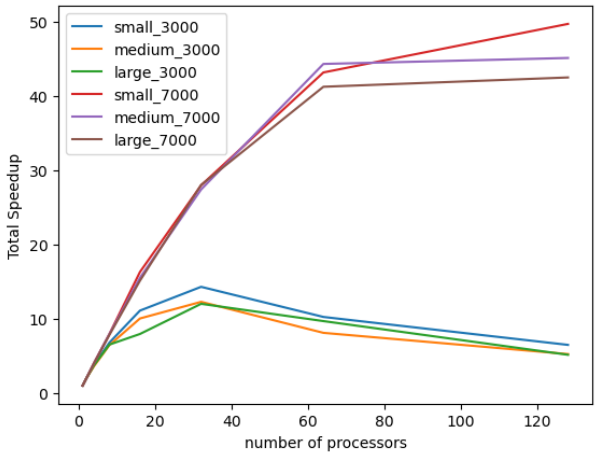

Figure 6: Total Speedup of Multilevel approach

In Figure 6, the total speedup of multilevel approach have two different trends according to the input size. The trend for smaller cases (3000 cells) is decreasing speedup, while the trend for larger cases (7000 cells) is limited increasing speedup. The reason for the decreasing speedup may be similar to that in our SA approach: the input size is a bit too small, causing the MPI environment costs to dominate. As for the larger cases, the total speedup may not be as good as the computation speedup, which is almost 120 speedup when there are 128 threads. But it still has pretty decent increasing speedup.

6.5 Computation Speedup based on Dense/Sparse graph

We present the computation speedup of 4 cases with different types of inputs in Figure 7 and Figure 8. These cases vary based on their distribution types: sparse distribution and dense distribution, where cells are heavily interconnected and exhibit high dependency.

The computation speedup of SA approach is presented in Figure 7. We can observe that cases with dense distribution have much worse speedup compared with those with uniform distribution. One primary reason why high dependency hinders speedup of our parallel SA approach is due to the large

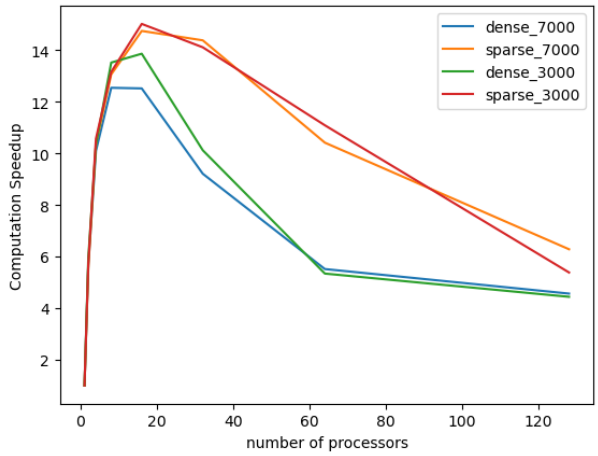

Figure 7: Computation Speedup of SA approach

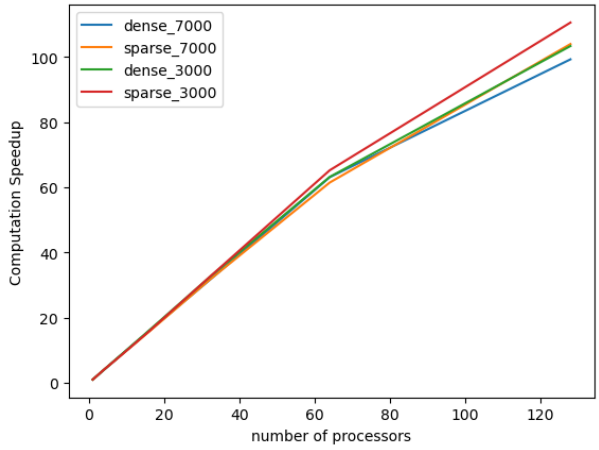

Figure 8: Computation Speedup of Multilevel approach

communication overhead required to maintain the integrity of electrical connections across different partitions. Each cell may be connected through multiple edges, making it necessary to exchange data frequently between MPI processes to synchronize changes. High level dependency increases message traffic, which can overwhelm the communication medium and increase the total computation time. Moreover, ensuring that all processes handling connected cells to guarantee quality of the solutions (cutsize) are updated in a timely manner introduces high synchronization costs, where threads must often wait for updates. An example is our second version of parallel SA, where timers are set for threads to wait.

Load balancing can be another factor. Cells in densely connected regions of the circuit require more computation costs compared to those with fewer connections, making it difficult to evenly distribute tasks among threads. This results in some threads becoming overloaded and others underutilized, leading to inefficiencies and idle times. Furthermore, the high dependency restricts the amount of parallelism that can be effectively exploited. Our operations (checking `cellArray`/`netArray`, move to change cell partition, `balcondition` to check the balance constratints, and updating data structure) need to be carried out in a specific sequence to preserve the functionality and integrity of the circuit, limiting the tasks that can be executed concurrently.

The computation speedup of SA approach is presented in Figure 8, which achieves pretty good speedup as the number of threads increases. The dense test cases may have a bit worse speedup performance than the sparse ones, but the difference is not large. This indicates that our multilevel approach is able to handle cases with high dependency between cells and edges.

6.6 Solution Performance (Cut Size Between Partitions)

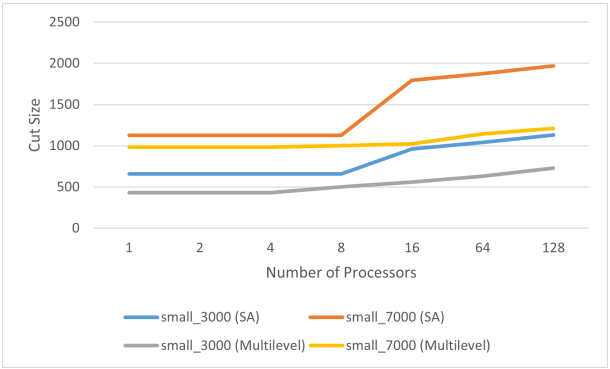

Figure 9: Cut Size of small cases (3000 cells and 7000 cells) between SA and Multilevel approach

We present the cut size solution of 2 cases with 3000 cells and 7000 cells in Figure 9. For both cases, the multilevel approach finds a better solution compared with the SA approach. In small 3000, grey line (Multilevel approach) has a smaller cut size compared with the blue line (SA approach). In small 7000, yellow line (Multilevel approach) has a smaller cut size compared with the blue line (SA approach).

In addition to demonstrating that the multilevel approach more effectively approximates the optimal solution, Figure 9 also shows that the quality of the solution only worsens slightly as the number of threads increases. It is worth to mention that the speedup achieved by our parallel algorithms does not sacrifice solution performance.

7 Work Distribution

- James (50%): devise the idea of simulated annealing approach and KL and multilevel, implement the parallel simulated annealing, devise the idea of multilevel approach, writing, figure plotting, benchmark design.
- Yueqi (50%): devise the idea of simulated annealing approach and KL and multilevel, figure plotting, implement the KL and multilevel approach, writing.

References

[1] Stephen Barnard and Horst Simon. "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems." In: Jan. 1993, pp. 711–718.

[2] J R Gilbert and E Zmijewski. "Parallel graph partitioning algorithm for a message-passing multiprocessor". In: *Int. J. Parallel Program.; (United States)* (1987).

[3] Bruce Hendrickson and Robert Leland. "A multilevel algorithm for partitioning graphs". In: *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. Supercomputing '95. San Diego, California, USA: Association for Computing Machinery, 1995, 28–es. ISBN: 0897918169. DOI: 10.1145/224170.224228. URL: https://doi.org/10.1145/224170.224228.

[4] *Partitioning of VLSI circuits and systems*. 1996, pp. 83–87. DOI: 10.1109/DAC.1996.545551.

[5] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997. eprint: https://doi.org/10.1137/S1064827595287997. URL: https://doi.org/10.1137/S1064827595287997.

[6] D. Kolar, J.D. Puksec, and I. Branica. "VLSI circuit partition using simulated annealing algorithm". In: *Proceedings of the 12th IEEE Mediterranean Electrotechnical Conference (IEEE Cat. No.04CH37521)*. Vol. 1. 2004, 205–208 Vol.1. DOI: 10.1109/MELCON.2004.1346809.