3 Description of Strategy and Algorithm

In this section, we discuss about our proposed algorithms and strategies for graph partitioning, convexity detection and longest depth of partition set $\boldsymbol{\pi}$ of G(V, E).

3.1 Strategy

There are two subproblems in the graph partitioning and covering problem. First deals graph partitioning while second deals with convexity detection and longest depth of partition set $\boldsymbol{\pi}$. We are using the following strategies for graph covering, convexity detection and for longest depth of partition set $\boldsymbol{\pi}$ of G(V, E).

1. Create a random initial k-way partition set π = (P_1, • • • , P_k) of the initial graph G(V, E) while maintaining the balancing constraints of each partition P_i i.e. L(P_i) ≤ w(P_i) ≤ U(P_i) , 1 ≤ i ≤ k.
2. Start with an initial partition set π = (P_1, • • • , P_k). Calculate the initial cost of each node and assume the final cut-size of graph G(V, E) is equal to the initial cut-size.
3. At each iteration during a pass, consider all possible moves of each unlock node from its home partition to any of the other (k – 1) partitions and choose the best such move according to the maximum move gain while maintaining the balancing constraints of size of partition. Update the node cost and move gain of all affected nodes. Perform passes until no improvement in cutsize size is obtained.
4. For each i, i=1 to k, calculate every path p between input node $v_i \in V$ and output node $v_o \in V$ of partition P_i . Check if any node of path p belongs to P_t where $P_i \neq P_t$. If such node is found then add the partition P_i to non-convex partition set $P_{nc} \in \pi$ else convex partition set $P_c \in \pi$. Choose the longest depth L_i among every path p of partition P_i.
5. Choose the longest depth L_{nc} of non-convex partition set P_{nc} among the longest depth L_i of each partition $P_i \in P_{nc}$, 1 ≤ i ≤ k. Choose the longest depth L_c of convex partition set P_c among the longest depth L_i of each partition $P_i \in P_c$, 1 ≤ i ≤ k.
6. Increment the value of iteration itr by 1.
7. Repeat the step 1, step 2, step 3, step 4, step 5 and step 6 until itr becomes maximum or $L_{nc} \leq L_c$.

3.2 A k-way Graph Partitioning

In k-way graph partitioning method, we start with a random initial partition set π = (P_1, • • • , P_k) and calculate the initial cost of each node of G(V, E). At each step during a pass we consider all possible moves of each node from its source part to any of the other parts in the partition and choose the best such move i.e. the one with the maximum move gain. The selected node is then moved to the target part and is locked for rest of the pass. The node gains of all the affected neighbors are updated accordingly. The next node is chosen in the same way from all the remaining unlocked nodes and is moved to its target part. The node move process is repeated until all the nodes are locked or there are no legal moves available due to the balance constraint. Now maximum gain sum S_b of the selected b nodes is calculated and if S_b is found positive then we make the first b nodes move permanent and improve the cut-size by decreasing it by S_b. Maximum gain sum S_b defines the effect on cut-size after moving first b selected nodes permanently. S_b is equal to the sum of move gains of first b selected nodes. All the nodes moved after the bth node move is reversed to the original part so that the actually moved nodes are the sequence of moving first node from part a_1 to part b_1, second node from a_2 to b_2 ,..., the b_{th} node from a_b to b_b , where a_1,a_2 ....a_b,b_1,b_2,....b_b∈ π = (P_1, • • • , P_k), k is the number of parts. The whole process is called a pass. Several passes are performed until S_b is no longer positive. Then we say that the local optimum with respect to the initial random solution is obtained. The move that do not violate the balance criterion are called legal moves, otherwise illegal. In this method, only legal moves are considered for a move. In k-way partitioning algorithms, each node has (k-1) possible move directions at each step in a pass where each move direction corresponds to a move from its source part to each of the other parts of partition set π. There are k(k-1) possible move directions in total as only the best move is considered in each move direction. An array of pointers, is called bucket array [4], is used for storing move gain for all possible moves. Bucket array is used to keep track of move gain of nodes with respect to their move direction. Since a bucket array stores move gain of all possible moves which fall in interval [-G_{max} , G_{max}].Thus size of bucket array is defined as

$$\textbf{bucketsize} = 2 \times G_{max} + 1 \qquad \textbf{(8)}$$

3.2.1 Initial_Partition Algorithm

We use Initial_Partition algorithm to produce a random initial k-way partition set π=(P_1, • • • , P_k of graph G, where k is the number of parts of partition set π. This algorithm maintains the balancing constraints of each part P_i , 1 ≤ i ≤ k, i.e. L(P_i) ≤ w(P_i) ≤ U (P_i), where L(P_i) and U(P_i) are lower and upper bound of size w(P_i) of part P_i. We take upper bound U(P_i) = n′, where n′ =|V′| and lower bound L(P_i) = 0, because each partition should be covered by the resource graph R(V′, E′).The value of k is calculated according to the resource graph R(V′,E′) and application graph G(V,E) and given by

Number of parts(k) =n/n' +1, where n=|V| and n'=|V'| **(9)**

We add 1 to n/n′ because if n mod n′=0, then initial random partitioning creates k parts and each part contains n/n′ nodes which is U(P_i) of part P_i. Now if we try to move one node from one partition to another partition to minimize the cut-edges then it is not possible because balance criterion of partition $\boldsymbol{\pi}$ is not be satisfied. However, addition of 1 to n/n′ doesn't affect the partitioning when n mod n′ ≠ 0. The initial partition set π produced by this algorithm is used as the input in other algorithms for optimization of cut-size and longest depth L_{nc}.

3.3 Convexity and longest depth

Non-convexity of a partition P_i increases the inter-connecting edges between P_i and other partitions and it also increases the longest path or critical path of partition P_i. In VLSI circuit design, critical path of partition is a major problem because it increases the cost of circuit design. Minimization of cut-size decreases the number of non-convex parts but critical path remains an issue for circuit designing. After obtaining a local optimum value of cut-size, we try to minimize the longest depth L_{nc} of non-convex partition set P_{nc} using Longest_Depth, Init_Dfs and Dfs algorithms.

3.3.1 Longest_Depth Algorithm

We calculate the longest depth of each partition $P_i \in \pi$ using this algorithm. We use a flag to determine whether P_i is convex or non-convex partition. The value of flag is set when partition is non-convex and it is set by Dfs algorithm. At first, we calculate the local longest depth between every input node $v_i \in P_i$ and every output node $v_o \in P_i$ of partition $P_i \in \boldsymbol{\pi}$ using Init_Dfs and Dfs algorithm. From all local longest depth of P_i, we choose a local longest depth with maximum value, is called longest depth L_i of partition P_i. If the value of flag is set then P_i is added to non-convex partition set P_{nc} otherwise P_i is added to convex partition set P_c . This process is repeated for all

k partitions. Now, we choose the maximum longest depth L_{nc} among all partitions of P_{nc}, is called longest depth of non-convex partition set P_{nc} . Similarly, we choose a maximum longest depth L_c and a minimum longest depth L_{mc} among all partitions of P_c, are called longest depth and minimum longest depth of convex partition set P_c respectively. The number of all partitions for which flag is set, is called total number of non-convex partitions, is denoted by N_{ncp} and defined as

$$N_{ncp} = |P_{nc}| \quad (10)$$

3.3.2 Init_Dfs and Dfs Algorithm

Init_Dfs algorithm is used to initialize the data structures of Dfs algorithms. It creates two temporary arrays, first is, visited array to determine whether a node has been visited before or not and second is, path array to store all nodes of the path between an input node $v_i \in P_i$ and an output node $v_o \in P_i$ of partition P_i. Initially, all nodes are marked as unvisited. Dfs algorithm is like standard DFS algorithm except some differences. It checks all possible path between an input node $v_i \in P_i$and an output node $v_o \in P_i$. For each path p between v_i and v_o , all nodes in path array are examined that whether node belongs to partition P_i or not. If any node of any path between v_i and v_o doesn't belongs to P_i then flag is set. A path p with maximum length is selected, is called local longest depth and it is returned to Longest_Depth algorithm.

3.4 Convex Partitioning and Mapping Algorithm

This is our main algorithm which uses the functionalities provided by all the above algorithms. This algorithm does recursively partitioning to minimize the longest depth L_{nc}. We have assumed the maximum iterations for recursive partitioning is 40. We can't allow recursive partitioning more than maximum iterations because it increases the time complexity. However, it is practically observed that number of iterations itr lies in interval [1, 15] for most of the graphs. At the starting, a random seed is created and set. An initial k-way partition set π is obtained by using Initial_Partition algorithm. Then the cut-size of initial partition set is minimized by performing several passes on the initial partition set π and it continues until no improvement in cut-size size is obtained. When a local optimum partition set in term of cut-size is obtained, the longest depth L_{nc} of non-convex partition set P_{nc} and longest depth L_c of convex partition set P_c is computed by using Longest_Depth, Init_Dfs and Dfs algorithms and a comparison between L_{nc} and L_c is made. Either if L_{nc} is less than or equal to L_c or itr is greater than maximum iteration then the algorithm produces output and stops otherwise recursive partitioning continues.