A New Effective And Efficient Multi-Level Partitioning Algorithm

Youssef Saab
Computer Engineering and Computer Science Department
University of Missouri-Columbia, USA

Abstract

This paper describes a new multi-level partitioning algorithm (PART) that combines a blend of iterative improvement and clustering, biasing of node gains, and local uphill climbs. PART is competitive with recent state-of-the-art partitioning algorithms. PART was able to find new lower cuts for a number of benchmark circuits.

1. Introduction

A partition of a hypergraph $G(V, E)$ is an unordered pair (V_1, V_2) of subsets of V such that $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. The size $S(A)$ of a subset of nodes $A \subseteq V$ is the sum of the sizes of its constituent nodes. A partition (V_1, V_2) is a bisection (balanced partition) if $\mid S(V_1) - S(V_2) \mid \leq \alpha S(V)$, where α is some pre-specified constant. A net e is said to be cut by a partition (V_1, V_2) if it links nodes in V_1 and in V_2, i.e., $e \cap V_1 \neq \emptyset$ and $e \cap V_2 \neq \emptyset$. The cut (cost) of a partition (V_1, V_2) is the subset (number) of nets cut and is denoted by $cut(V_1, V_2)$ ($cost(V_1, V_2)$). The network bisection problem (NB) seeks a bisection of minimum cost,

2. Multi-level partitioning

Contraction is an operation in which several nodes are clustered (contracted) together to form a single node. Contraction is used in the coarsening phase of multi-level algorithms [7, 2]. PART differs from other multi-level algorithms in its coarsening phase which produces a sequence $(G_0, P_0) \rightarrow (G_1, P_1) \rightarrow \ldots \rightarrow (G_{l-1}, P_{l-1}) \rightarrow (G_l, P_l)$, where $G = G_0$ is the input graph, and $P_i = (V_{i1}, V_{i2})$ is a partition of G_i for $0 \leq i \leq l$. For $1 \leq i \leq l$, the pair (G_i, P_i) is obtained from (G_{i-1}, P_{i-1}) as follows. An iterative improvement algorithm is applied to refine partition P_{i-1} which along the way computes disjoint subsets of nodes for contraction such that nodes to be contracted belong to the same side of the refined partition P_{i-1}. Therefore, after contraction of G_{i-1} into G_i, partition P_{i-1} projects to a partition P_i of G_i. To get the process started, the first partition P_0 of G_0 is generated randomly.

The un-coarsening phase in PART proceeds just like other multi-level algorithms. For $l > i \geq 0$, partition P_{i+1} is projected to a partition of G_i which is then refined to replace P_i by an iterative partitioning algorithm. However, unlike other multi-level algorithms, some refinement steps are skipped during the un-coarsening phase as we will explain later.

The advantages of the coarsening scheme used by PART are:

- The best partition is refined during coarsening, so that partition P_l of the coarsest graph projects to a good partition of $G = G_0$.
- Coarsening is a by-product of iterative improvement at almost no additional computation cost.
- Coarsening and un-coarsening can be repeatedly applied until no further improvements can be made.

During the un-coarsening phase many refinement steps are skipped altogether. A refinement step is skipped if it is unable to produce any improvement. The coarsening phase guarantees that $cost(P_{i+1}) \leq cost(P_i)$. Therefore during the un-coarsening phase, if $cost(P_{i+1}) = cost(P_i)$ then refinement is skipped. Otherwise, P_{i+1} projects to a partition of G_i that will be refined and then will replace P_i. We note that most of the refinement steps are skipped during the un-coarsening phase of PART. The reason for skipping refinement during the un-coarsening phase is simple. During the coarsening phase, P_{i+1} was obtained from P_i using an

This work was supported by a 1999 Research Board grant, University of Missouri.

iterative improvement algorithm. If during the uncoarsening phase we see that $cost(P_{i+1}) = cost(P_i)$, then the iterative improvement algorithm was unable to improve P_i during coarsening. This means that the iterative algorithm will not improve this partition now, and therefore the refinement step is skipped.

3. Iterative improvement and contrction

The core of PART is an iterative improvement pass that refines the current partition, and at the same time colors some of the nodes. The colors are used to contract the graph during the coarsening phase and are not used at all during the uncoarsening phase. The iterative improvment pass is a variant of the Fiduccia-Mattheyses [5] algorithm and is essentially a modified version of the improvement pass used in [8].

Given an initial bisection, i.e., a partition that satisfy the size balance criterion, sequences of nodes are moved from one side of the partition to the other side until every node has moved once. Several subsets of nodes in the same sequence may be colored with the same color signaling that these nodes should be clustered together. The rational for this clustering scheme is simple. Nodes that are heavily connected to each other tend to migrate together during iterative improvement. The best bisection seen during the iterative improvement pass is saved.

The order in which nodes are moved from one side of the partition to the other is determined by the gain of nodes. The gain of a node is the net decrease (may be negative) in the number of nets cuts if the node is moved to the other side of the partition. The node with highest gain is moved first. To facilitate retrieval of the node of highest gain, a bucket structure as described in [5] is used with the LIFO scheme as described in [6]. The first initial bisection is randomly generated.

Coloring of the nodes during the iterative pass is done as follows. Initially all nodes are not colored. A soon as a node v is moved, it is made contractable. Then v and all other contractable nodes that are connected to it via critical nets, are colored by a new color and their contractability is removed. If it turns out that only v gets the new color, then the color of v is removed and the contractability of v is restored. A net is critical for node v if it is removed from the cut after moving v to the other side of the partition. At the end of the iterative improvement pass, nodes of the same color are clustered together. Other nodes connected to v via non-critical nets are not selected for clustering with v to slow down the clustering process and for efficiency purposes. See [8] for more detailed explanation on this issue.

The function $MOVE(v, A, B)$ performs the update needed to move node v from side A to side B of the current partition. Basically, $MOVE(v, A, B)$ moves v from A to B, sets $free(v) = false$, updates gains of all affected nodes, and, while doing gain updates, it colors v and all its contractable neighbors via critical nets with a new color and set $contractable(x) = false$ for all nodes that got the new color. If only v gets the new color then it sets $color(v) = NoColor$ and $contractable(v) = true$.

we are now ready to present the complete the description of the iterative improvement pass with contraction:

1. For each node v, set $free(v) = true$, $color(v) = NoColor$, and $contractable(v) = false$.
2. Save the initial input bisection (V_1, V_2) as the best bisection so far (B_1, B_2).
3. Partition Side Selection: for $1 \leq i \leq 2$, let h_i be the highest gain of a free node in V_i. If V_i has no free nodes then set $h_i = -\infty$. If $h_1 = h_2$ then toss a balanced coin and set $j = 1$ or $j = 2$ depneding on the outcome of the toss. otherwise set j as the index of the side with the highest gain.
4. Forward move: If V_j has no free nodes then go to step 7. Otherwise let v be a free node of highest gain in V_j. Record cost of current partition in $OldCost$. Call $MOVE(v, V_j, V_{(3-j)})$. if $cost(V_1, V_2)$ >= $OldCost$ then repeat this step. Otherwise, for each contractable node x, set $contractable(x) = false$.
5. Restore size: Let j be the index of the largest side of the partition. As long as $S(V_j) - S(V_{(3-j)}) > \alpha S(V)$ and V_j has free nodes do: let v be a free node of highest gain in V_j. If $\mid S(V_j - v) - S(V_{(3-j)} \cup v) \mid < \mid S(V_j) - S(V_{(3-j)} \mid$ then call $MOVE(v, V_j, V_{(3-j)})$. Otherwise for each contractable node x, set $contractable(x) = false$.
6. Save: If (V_1, V_2) is a bisection then if $cost(V_1, V_2) < cost(B_1, B_2)$ or if $cost(V_1, V_2) = cost(B_1, B_2)$ and $\mid S(V_1) - S(V_2) \mid < \mid S(B_1) - S(B_2) \mid$, then replace (B_1, B_2) with (V_1, V_2). Go to step 3
7. Restore best bisection: replace (V_1, V_2) with (B_1, B_2).
8. Contraction: Color each remaining uncolored node with a new color. cluster together all node with the same color.

The above pass takes an input graph G_i and a initial bisection $P_i = (V_{i1}, V_{i2})$ of G_i. It then improves P_i and contract G_i. It returns the contracted graph

$G_{(i+1)}$ and an initial bisection $P_{(i+1)}$ of $G_{(i+1)}$ which is the projection of the best bisection P_i of G_i. We encapsulate this iterative improvement pass in the function $REFINE-AND-CONTRACT(G_i, P_i)$. This function is the main function used during the coarsening phase. The same function with the contraction operation removed is used in the uncoarsening phase. We call this function $REFINE(G_i, P_i)$. During the uncoarsening phase the function $MOVE(v, A, B)$ can be modified to ignore coloring of nodes although this does not significantly affect the running time.

A single multi-level pass consists of a coarsening phase followed by an uncoarsening phase. We will encapsulate this pass in the function $IMPROVE(G, P)$ which takes as input a hypergraph G and a partition $P = (V_1, V_2)$ of G, and returns an improved parttion of G. One approach to optimize an initial bisection is to call function $IMPROVE$ repeatedly until no more improvements can be made. We experimentally found that this approach can find good bisections. However, we were able to obtain better results by applying node gain biasing and local bisection perturbations inspired by the work in [3].

4. Node biasing

Paper [3] points out that during a pass of a Fiduccia-Mattheyses [5], a net e can be in one of three states:

- Free: all nodes of net e are free.
- Loose: only one side of the partition contains locked nodes of net e. The side that contains locked nodes of net e is called the anchor of e. The other side is called the tail of e.
- Locked: Both sides of the partition contain locked nodes of net e.

Furthermore, the state of a net changes from free to loose to locked in that order. Clearly locked nets cannot be removed from the cut for the remainder of the iterative pass. Therefore, it makes sense to increase the chances of removal of loose nets from the cut before they become locked. This is achieved by positively biasing the gains of nodes of a loose net that are in the tail of the loose net. The author of [3] use a formula for the additional bias that favors short nets, dense connectivity at the anchor, and sparse connectivity at the tail. They apply the bias every time a node of a loose net moves, and they do not remove the bias once the net is locked. Our approach to biasing is simpler. A bias of 1 is added to all nodes in the tail of a net e only immediately after the state of e changes from free to loose. Also, we remove the additional bias as soon as the loose net becomes locked.

We have experimently observed that biasing is not always good. After all some of the loose nets may actually belong to the optimal cut. Therefore, biasing them to be removed from the cut may hinder the algorithm from finding the optimal solution. Another case where biasing is not good is when the partition being improved is close to being optimal. In this case, biasing prevent the distinction between a move that actually improves the cut (positive gain without bias) from another that does not (equal positive gain but biased). Our experiments suggest that biasing is most beneficial initially when the partition is far from optimal, and that it should be turned off once the partition is close to optimality. Furthermore sometimes biasing altogether should be avoided. This lead us to the notion of selective biasing.

In selective biasing, only a selected subset of nets can be biased. if we pose the question: which nets are harder to remove from the cut? the intuitive answer is longer nets are harder to remove from the cut than short nets. One way to increase the chances of removal of long nets from the cut is to simply apply biasing to loose nets longer than some threshold length. We found that biasing long nets was beneficial for some circuits, but it was quite bad for others. For the other circuits, doing the opposite, i.e., biasing short nets, was very effective.

The conclusions that can be drawn from our preliminary experiments is that depending of the structure and connectivity of the input circuit one of the following may be beneficial:

- bias all loose nets ($mode = 1$).
- avoid biasing altogether ($mode = 2$).
- bias loose nets longer than some threshold length ($mode = 3$).
- bias loose nets shorter than some threshold length ($mode = 4$).

Since it is difficult to determine which of the four items above is helpful, the obvious choice is to try all of them and pick the best result. Indeed, this is exactly what we did. we ran $IMPROVE(G, P)$ four times each time with a random initial partition and one of the four biasing approach above. Furthermore, we only applied the biasing during the coarsening phase of $IMPROVE$ and turned it off during the un-coarsening phase. To apply biasing, The following sample code need to be added to the end of function $MOVE(v, A, B)$:

```
for (each net e incident to node v) do
 if ( net e is selected for biasing ) then
   if ( v is the only locked node of e ) then
     /*free to loose transition*/
     for (each node w of e ∩ A) do
         gain(w) = gain(w) + 1;
   else if ( e has locked nodes in A and v is
         the first locked node of e in B ) then
     /*loose to locked transtition*/
     for (each node w of e ∩ B) do
         gain(w) = gain(w) − 1;
```

We call our approach global sampling since we sample four different initial random partition to obtain a bisection that will be subject to further refinement. We used a threshold of 5 for selective biasing. Obviously other thresholds can be used. We found that a threshold of 5 works reasonably well on all the circuits which we experimented with. This global sampling is summarized in the pseudo-code:

```
generate a random initial bisection P;
Q = P; /* save best bisection so far*/
for 1 ≤ i ≤ 4 do {
    call IMPROVE(G, P) using mode i;
    if (cost(P) < cost(Q)) then
      Q = P; /* save best bisection so far*/
    generate a random initial bisection P;
    }
return Q;
```

The bisection Q returned by the above code will be further optimized as we will explain next.

5. Local sampling

Like global sampling, local sampling also perturbs the current partition then it attempts to improve it. However unlike global sampling which starts each time with a brand new random initial bisection, local sampling attempts to perturb only the nodes of nets that belong to the cut. This local hill-climbing does not increase the cost of the perturbed partition significantly. Yet it allows the removal from the cut of certain nets that are harder to remove otherwise. Our local sampling is similar to the so called stable net removal in [3].

We encapsulate the local perturbation in the function $PERTURB(P)$. This function first sets the states of all nodes of cut nets free. It then scans the cut nets of bisection P in random order. For each scanned net e the following is done. If all the nodes of e on the larger side of the partition are free, then they are moved to the other side of the partition and they are locked. Otherwise If all the nodes of e on the smaller side of the partition are free, then they are moved to the other side of the partition and they are locked. Nothing is done if net e has locked nodes on both sides of the partition.

After scanning all the cut nets of the initial bisection P, the perturbed partition P may no longer be a bisection. Partition P is then restored to a bisection as follows. The nodes are scanned one at a time and the scanned node is moved to the other side of the partition if: (1) moving it improves the balance of the partition, or (2) if (1) is false and the node currently belongs to the larger side of the partition and a random coin toss yields a head. The nodes are repeatedly scanned in this fashion until the balance criterion of the bisection is restored. Although it may seem that this approach is expensive, we have observed that it does not take much time for the 45-55% balance criterion (10% deviation in size) that is typically used in the literature.

Local smapling can now be described by the following pseudo-code:

```
Q = best bisection so far;
repeat 4 times:
        P = PERTURB(Q);
        IMPROVE(G, P);
        if (cost(P) < cost(Q)) then
          Q = P; /* save best bisection so far*/
return Q;
```

We iterate 4 times in local sampling just like we did for global sampling. Off course, any other number of iterations can be used. Local sampling can only improve its input bisection. If no improvements can be made the initial bisection is returned intact.

The full algorithm PART can now be described as follows:

1. Q = bisection returned by global sampling.
2. Let P be the best partition obtained by applying local sampling to Q.
3. Repeatedely call $IMPROVE(G, P)$ until no further improvement can be made.

In algorithm PART, node biasing is applied only during the coarsening phases of global sampling as explained before.

6. Experimental results

In table 1 we compare the result of PART to previously reported results in the literature on the ISPD98 benchmarks [1]. All experiments were performed on

Table 1. Results on ISPD98 benchmarks.

Test	time(s)	cut	fr3	FM	CL	hM
ibm01	14.3	180	9	6.1	0.6	0.0
ibm02	31.4	262	9	1.5	1.1	0.0
ibm03	39.7	950	10	21.1	12.4	0.6
ibm04	55.9	522	2	15.5	7.9	3.8
ibm05	68.2	1672	6	12.1	28.3	2.6
ibm06	67.4	885	7	9.9	10.4	0.3
ibm07	108.6	824	6	25.8	12.7	3.5
ibm08	115.6	1140	10	12.7	10.6	0.2
ibm09	120.4	620	10	47.1	8.7	0.6
ibm10	212.0	1249	3	19.3	13.7	0.6
ibm11	200.7	956	2	52.6	11.2	0.4
ibm12	265.7	1872	5	20.5	27.5	2.5
ibm13	267.3	831	7	42.1	9.9	1.1
ibm14	499.1	1794	7	65.2	41.4	2.4
ibm15	981.7	2587	1	97.4	38.0	1.5
ibm16	953.6	1710	6	38.2	54.3	2.6
ibm17	158.9	2186	5	39.6	28.2	2.4
ibm18	876.3	1521	6	12.2	49.1	1.3

a DEC 8400 station. All the results are reported for 45-55% partitions (deviation up to 10% of exact bisection). For our algorithm PART, we report the average running time of a single run in seconds, the minimum cut found in 10 runs (cut), and the number of solutions in 10 runs that are within 3% (fr3) of the smallest cut. The results of three other algorithms are reported as percentages in cut over the cut produced by PART. The three other algorithms are FM [5], CLIP (CL) [4] and hMetis (hM) [7]. For FM and CLIP, we used the results reported in [1] which are the minimum cuts over hundred runs of each algorithm. The results of hMetis were erroneous in [1]. the corrected results which we used are available from vlsicad.cs.ucla.edu web site.

PART performed well on the ISPD98 benchmarks, and in all cases obtained the smallest cut. As can be readily seen from Table 1, PART significantly improves over the results of FM and CLIP. The improvement over the results of hMetis which is a very powerful partitioner are modest. For example on circuit ibm04, the result of hMetis is 3.8% larger than the result of PART.

Algorithm PART is robust. In many cases 5 out of 10 runs are within 3% of the minimum as column fr3 in table 1 show. This means that only few runs are needed to get a good solution.

The running time of PART is reasonable. On the largest circuit, ibm18, with 210613 nodes and 201920 nets, a single run of PART takes about 15 minutes.

7. Conclusion

We have presented an efficient, effective, and robust partitioning algorithm. Our algorithm effectively combines node gain biasing and local uphill climbs within the multi-level framework. For several circuits new lower cuts have been obtained than previously reported in the literature. In the future, we intend to investigate possible improvement to our algorithm which may include new ways of clustering and node biasing. We think that the running time can be improved without significant degradation in quality by limiting the number of levels during the coarsening phase (currently we do not use any preset limit), and by using the early-exit strategy in the un-coarsening phase as described in [7].

References

[1] C. Alpert. The ISPD98 circuit benchmarks suite. *Physical Design Workshop*, pages 80–85, 1998.

[2] C. J. Alpert, J.-H. Huang, and A. B. Kahng. Multilevel circuit partitioning. *Design Automation Conference*, pages 530–533, 1997.

[3] J. Cong, H. P. LI, S. K. Lim, T. Shibuya, and D. Xu. large scale circuit partitioning with loose/stable net removal and signal flow based clustering. *International Conference on Computer-Aided Design*, pages 441–446, 1997.

[4] S. Dutt and W. Deng. VLSI circuit partitioning by cluster-removal using iterative improvement techniques. *International Conference on Computer-Aided Design*, pages 194–200, 1996.

[5] C. Fiduccia and R. Mattheyses. A linear-time heuristics for improving network partitions. *Proceedings of the 19th Design Automation Conference*, pages 175–181, January 1982.

[6] L. W. Hagen, D. J. H. Huang, and A. B. Kahng. On implementation choices for iterative improvement partitioning algorithms. *IEEE Transactions on Computer-Aided Design*, 16(10):1199–1205, 1997.

[7] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Application in VLSI domain. *Design Automation Conference*, pages 526–529, 1997.

[8] Y. Saab. A fast and robust network bisection algorithm. *IEEE Transactions on Computers*, 44(7):903–913, July 1995.