MULTI-LEVEL GRAPH PARTITIONING

By

PAWAN KUMAR AURORA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2007

To my loving family

ACKNOWLEDGMENTS

I am deeply indebted to my supervisor Prof. Dr. Timothy Davis whose stimulating suggestions and encouragement helped me. His expertise in writing robust software was useful and a great learning experience for me. I have never written a better software. I would like to express my thanks to Dr. William Hager and Dr. Arunava Banerjee for agreeing to be on my committee.

My special thanks go to my parents for supporting my decision to pursue higher studies and providing all the moral support. My brother Jitender deserves a special mention here for being my best friend. Last but not the least, I thank my fiance Sonal for all her love and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

MULTI-LEVEL GRAPH PARTITIONING

By

Pawan Kumar Aurora

August 2007

Chair: Timothy Davis
Major: Computer Engineering

Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, VLSI design, and task scheduling. The multi-level graph partitioning algorithm reduces the size of the graph gradually by collapsing vertices and edges over various levels, partitions the smallest graph and then uncoarsens it to construct a partition for the original graph. Also, at each step of uncoarsening the partition is refined as the degree of freedom increases. In this thesis we have implemented the multi-level graph partitioning algorithm and used the Fiduccia Mattheyses algorithm for refining the partition at each level of uncoarsening. Along with the few published heuristics we have tried one of our own for handling dense nodes during the coarsening phase. We present our results and compare them to those of the Metis software that is the current state of the art package for graph partitioning.

CHAPTER 1
INTRODUCTION

Given an un-weighted graph G with V vertices and E edges and given a number k, the Graph Partitioning problem is to divide the V vertices into k parts such that the number of edges connecting vertices in different parts is minimized given the condition that each part contains roughly the same number of vertices. If the graph is weighted, i.e. the vertices and edges have weights associated with them; the problem requires the sum of the weights of the edges connecting vertices in different parts to be minimized given the condition that the sum of the weights of the vertices in each part is roughly the same. The problem can be reduced into that of bisection where the graph is split into two parts and then each part is further bisected using the same procedure recursively. The problem addressed in this thesis is that of bisecting the given graph according to a given ratio. Also, the input graph is assumed to be un-weighted. However, this assumption is just at the implementation level and does not in any way change the underlying algorithms.

It has been shown that the Graph Partitioning problem is NP-hard [1] and so heuristic based methods have been employed to get sub-optimal solutions. The goal for each heuristic method is to get the smallest possible cut in reasonable time. We discuss some popular methods in the next chapter.

Graph Partitioning is an important problem since it finds extensive applications in many areas, including scientific computing, VLSI design and task scheduling. One important application is the reordering of sparse matrices prior to factorization. It has been shown that the reordering of rows and columns of a sparse matrix can reduce the amount of fill that is caused during factorization and thus result in a significant reduction in the floating point operations required during factorization. Although the ideal thing is to find a node separator rather than an edge separator, an edge separator can be converted into a node separator using minimum cover methods.

The multi-level graph partitioning algorithm reduces the size of the graph gradually by collapsing vertices and edges over various levels, partitions the smallest graph and then uncoarsens it to construct a partition for the original graph. Also, at each step of uncoarsening the partition is refined as the degree of freedom increases. In this thesis we have implemented the multi-level graph partitioning algorithm and used the Fiduccia Mattheyses algorithm for refining the partition at each level of un-coarsening.

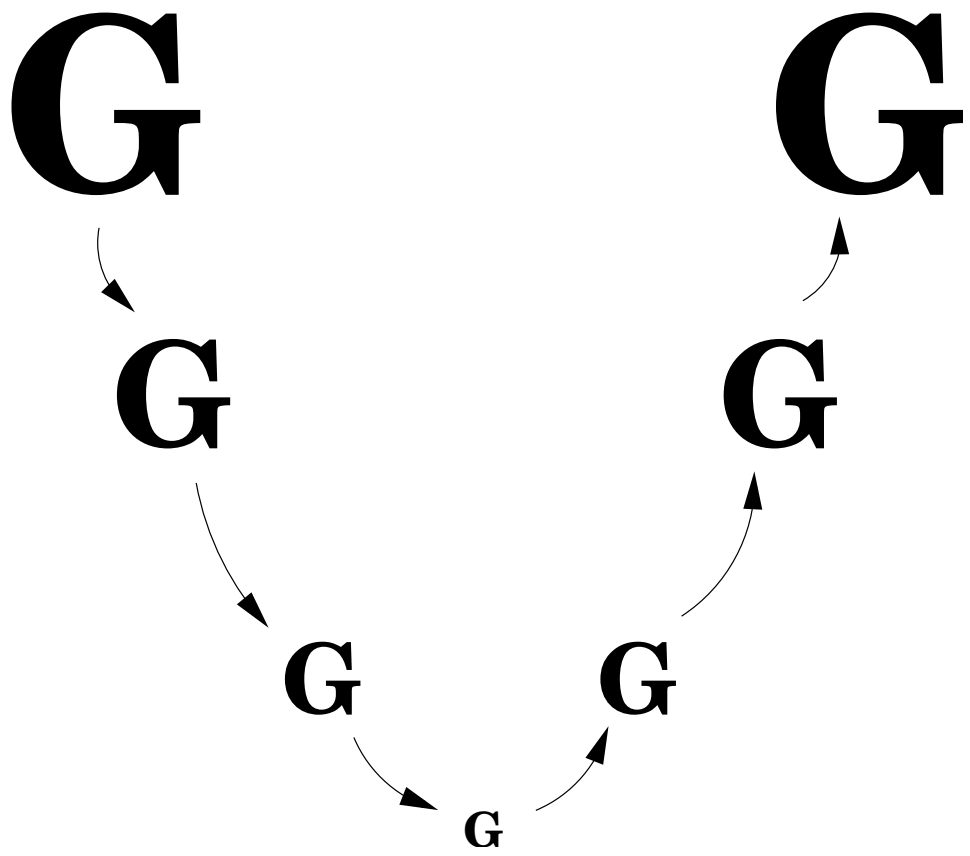


Figure 1-1. Multi-level Coarsening and Uncoarsening. The smallest graph is cut and the partition gets projected and refined as it moves up to the original biggest graph.

In the multi-level approach the coarsening phase is important. If a graph is folded in such a way that the properties of the graph are preserved i.e. the coarse graph is a smaller replica of the fine graph, a good cut of the coarse graph translates into a good cut of the fine graph [2]. Using the general coarsening heuristics described in chapter 3, it is possible that the properties of the graph are not preserved when handling dense nodes and may

result in a very unbalanced coarse graph. We present a heuristic for handling dense nodes that preserves the properties of the graph during coarsening and gives a more balanced coarse graph in fewer coarsening steps.

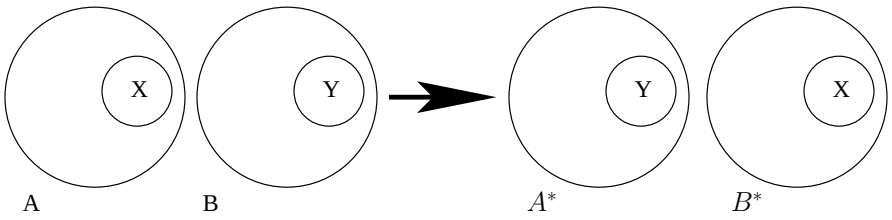
CHAPTER 2
REVIEW OF SOME GRAPH PARTITIONING METHODS

2.1 Kernighan and Lin

The KL algorithm [3] starts with an arbitrary partition and then tries to improve it by finding and exchanging a set of nodes in one partition with a same size set in the other partition such that the net effect is a reduction in the cut size. The set of nodes is found incrementally starting with one node in each partition and adding one node in each step. The algorithm starts by calculating the difference of external and internal costs for each node and then selects that pair of nodes a and b, one in each partition, for which the gain is maximum. Gain is defined as

$$G = D_a + D_b - c_{ab},\ where\ D_a = E_a - I_a\ and\ D_b = E_b - I_b.$$

Nodes a and b are then kept aside and the D values are recalculated for the remaining nodes assuming that nodes a and b have been swapped. The process is repeated and a new pair of nodes is selected. This is repeated until all the nodes get set aside. Finally a subset of k nodes is selected from each partition such that the net gain is the maximum positive value. These nodes are exchanged and the new graph is again improved using the same algorithm. The process stops when the maximum gain by exchanging any subset of nodes is not more than 0.



$$A^* = A - X + Y$$
$$B^* = B - Y + X$$

Figure 2-1. Kernighan and Lin: subsets X and Y are swapped

2.2 Fiduccia and Mattheyses

The FM algorithm [4] is basically an efficient implementation of the KL algorithm using special data structures that bring down the complexity from n^2 to linear in terms

of the size of the graph. Also, unlike KL, the FM algorithm moves one node at a time instead of exchanging a set of nodes between the partitions. However, it uses the concept of node gain and moves nodes even if the gain is negative in order to climb out of local minima. This algorithm works by inserting nodes into various buckets according to their gains and uses a doubly-linked list for nodes within the same gain bucket. This makes the deletion and insertion of nodes into buckets a constant time operation. More details about this algorithm are presented in the next chapter where we discuss our implementation of this algorithm.

2.3 Quadratic Programming

Hager et al [5] have shown that the Graph Partitioning problem can be formulated as the following continuous quadratic programming problem:

$Minimize\ f(x) := (1-x)^T(A+D)x$

$Subject\ to\ 0 \leq x \leq 1,\ l \leq 1^T x \leq u,$

where 1 is the vector whose entries are all 1. When x is a 0/1 vector, the cost function $f(x)$ is the sum of those a_{ij} for which $x_i = 0$ and $x_j = 1$. Hence, when x is a 0/1 vector, $f(x)$ is the sum of the weights of edges connecting the sets V_1 and V_2 defined by

$V_1 = i : x_i = 1\ and\ V_2 = i : x_i = 0.$

They have shown that for an appropriate choice of the diagonal matrix D, the min-cut is obtained by solving the above minimization problem.

CHAPTER 3
METHODS

3.1 Graph Compression

The sparse matrices resulting from some finite-element problems have many small groups of nodes that share the same adjacency structure. We compress such graphs into smaller graphs by coalescing the nodes with identical adjacency structures. As a result of compression, the graph partitioning algorithm must process a smaller graph; this, depending on the degree of compression achieved, can reduce the partitioning time.

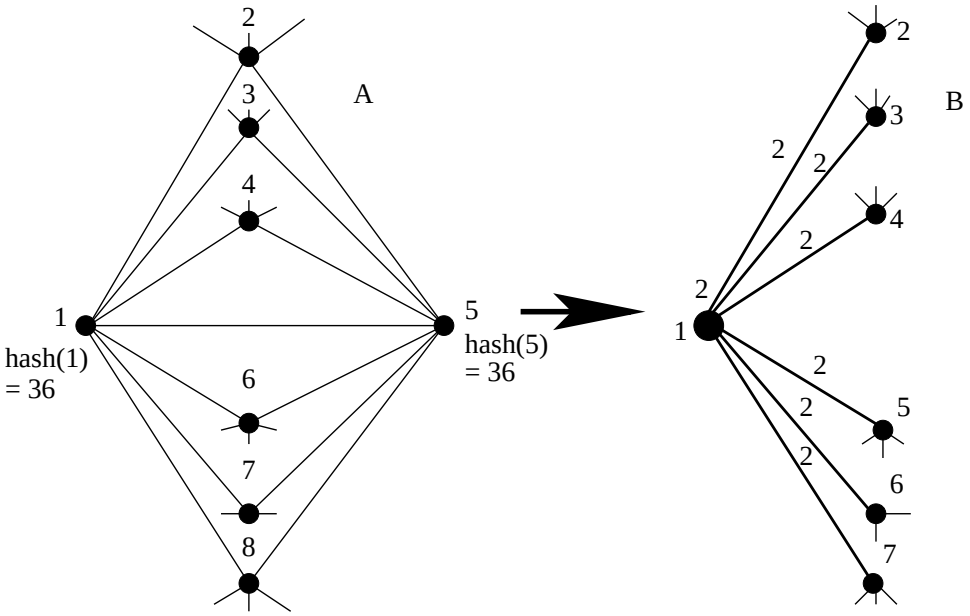


Figure 3-1. Graph compression. A) Input graph having 13 edges and 8 nodes. Nodes 1 and 5 have the same adjacency structure. B) Compressed graph with 6 edges and 7 nodes.

We used a hash value based technique for compression. For each node of the input graph, a hash value is calculated as follows: $hash(i) = i + \sum(Adj(i))$. For example for node 2 with adjacency structure 1, 3, 5, 10, $hash(2) = 2 + 1 + 3 + 5 + 10 = 21$. Then the hash value of each node is compared to that of its neighbors. If two nodes have the same hash value, then their adjacency structures are compared. Having the same hash value does not guarantee that the nodes have the same adjacency structure, although not having the same hash value does guarantee that the nodes do not have the same adjacency

structure. All the nodes having the same adjacency structure are then merged into one node and the new node has a weight equal to the sum of the weights of the merged nodes. All the edges from the merged nodes to the common neighbors are collapsed together and their weights added up. In case the input graph is un-weighted, each edge and node is assumed to have a weight of 1 and the output compressed graph is weighted. The time taken to compress the input graph is proportional to its size.

3.2 Handling Disconnected Graphs

A lot of sparse matrices arising in practice have disconnected components. That is the underlying graphs have more than one connected components. Such graphs during coarsening cannot be folded beyond a certain size since there are no edges left to be matched. We pre-process such graphs before passing them to the partitioning routine. For all input graphs, we determine if they have one or more connected components by doing a depth first search and marking all nodes that can be reached. If an input graph has more than one connected components, we pick a random node from each component and add zero weight edges joining these nodes. This results in a fully connected graph that can be coarsened down to two nodes if required. However, the coarsening heuristics tend to avoid folding these edges until there are no other edges left. This preserves the properties of the original graph during coarsening and the cut of the coarsest graph is closer to that of the input graph.

3.3 Multi-level Algorithm

The multi-level algorithm as implemented by us coarsens the input graph to the required size but does not uncoarsen and refine it back to the original graph in one step. Rather it uncoarsens it up to a certain intermediate level and then coarsens it back to the lowest level. This process of uncoarsening and refining up to an intermediate level and coarsening again to the lowest level is repeated a number of times. Each time the partition vector is saved and the partition that gives the best cut is used when uncoarsening and refining proceeds past the intermediate level to the top level graph. Since the coarsening

heuristics use randomization, by repeating it several times we can get significantly different results and use the best of them. Also, since this repetition is done only from an intermediate level when the graph is much smaller, it does not account for a substantial increase in the running cost.

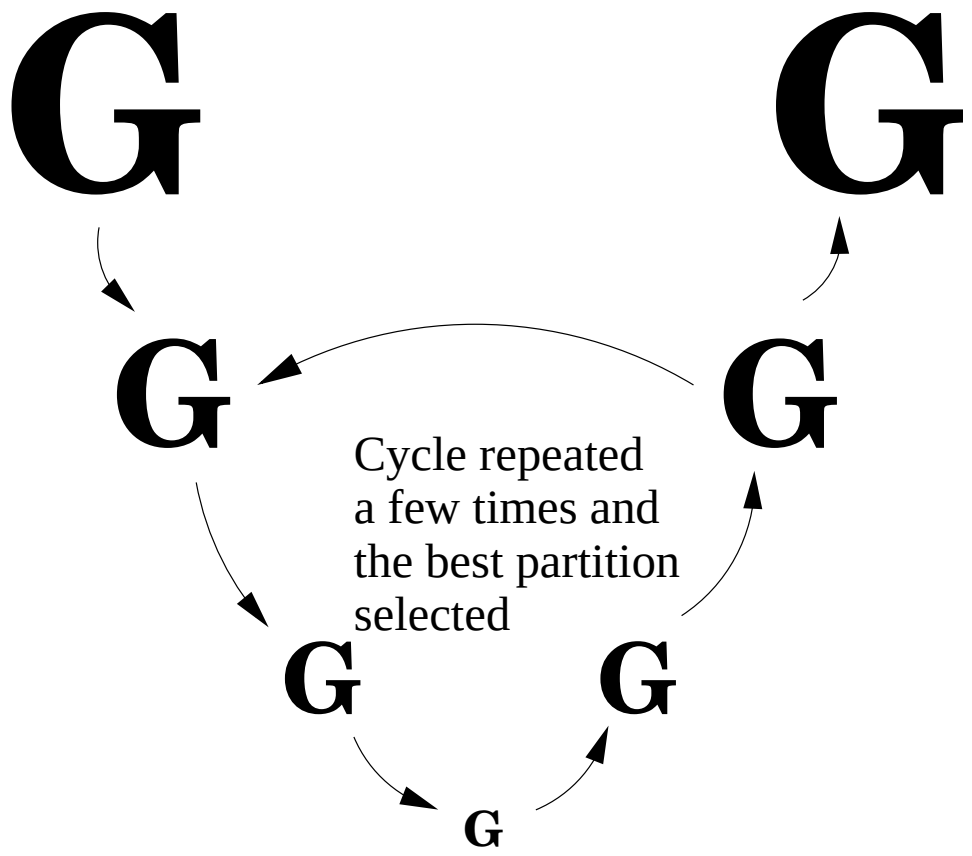


Figure 3-2. Multi-level with repeated coarsening and uncoarsening plus refining.

3.4 Coarsening

The input graph is coarsened over several levels to get the coarsest graph that is not larger than a user defined size in terms of the number of vertices. To go from a fine graph to a coarse graph at the next level, two steps are necessary. During the first, a maximal matching of vertices of the fine graph is found and during the second, these matched vertices are collapsed to get the coarse graph.

Based on a heuristic, an unmatched vertex is matched to one of its unmatched neighbors. Maximal matching is obtained when there are no edges that are incident on

two unmatched vertices. We use a total of four different heuristics to get the matchings. These heuristics are employed at different times or sometimes a combination of two different heuristics is used to find the matchings for the same graph.

3.4.1 Random Matching

A random permutation of the vertices of the graph is generated. The vertices are then visited in this order and for every unmatched vertex another random permutation of its adjacent vertices is generated. The neighbors are then visited in this random order and the first unmatched vertex is selected to be matched.

3.4.2 Heavy Edge Matching

As with Random matching, first a random permutation of all vertices is generated. The vertices are then visited in this order and for every unmatched vertex all its unmatched neighbors are visited and the one that connects with the heaviest edge is selected. The idea is to match the heavy edges so that they get collapsed and the resulting coarse graph has only light edges that can be cut.

3.4.3 Heaviest Edge Matching

As the name suggests, the edges are sorted according to their weights and matching begins by selecting the heaviest edge. All the edges are visited in descending order and edges with unmatched end points are selected. This heuristic is used when the graph size has been reduced substantially so that not much work is done in sorting the edges.

3.4.4 Zero Edge Matching

This heuristic is used to match the zero weight edges that are added when handling dense nodes. It is similar to the heavy edge matching heuristic except that instead of matching the heaviest edge, we match an edge of weight zero. In order to differentiate these zero weight edges with the ones added to connect the disconnected components, we match those vertices that have a common neighbor. Also, we match the second zero weight edge instead of the first so that the last zero weight edge remaining after collapsing

all but two of the edges adjacent to a dense node is not selected for matching. The heuristic for handling dense nodes is discussed in detail in the next section.

Once we have a maximal matching of the vertices of the fine graph, we create a mapping vector that maps the vertices in the fine graph to those in the coarse graph. Then using the matching and the mapping vectors, the coarse graph is constructed. For every unmatched vertex, its adjacency structure is copied over to the coarse graph whereas for matched vertices, the new vertex in the coarse graph that is mapped to these vertices has a union of their adjacency structures minus the edge that connected them, as its adjacency structure. The edge weights are copied over except when the matched vertices have a common neighbor. In that case the edges to the common neighbor get merged into one and the new edge has a weight equal to the sum of the weights of the merged edges. Similarly, the new vertex gets the sum of the weights of the merged vertices. Any duplicate edges resulting from the process are merged together with their weights added.

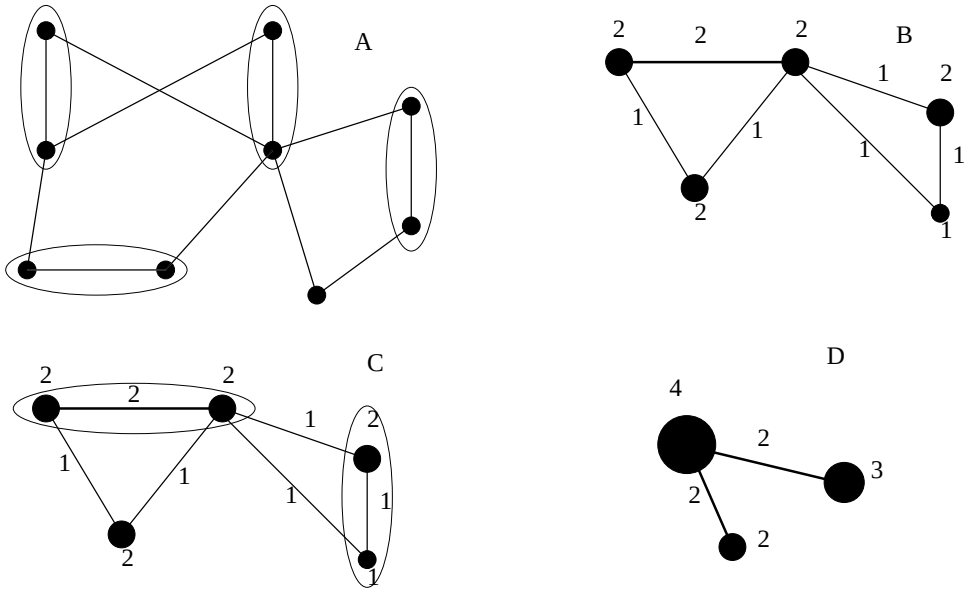


Figure 3-3. Graph coarsening. A) The original graph showing a maximal matching. B) Graph after one step of coarsening. C) A maximal matching of the coarse graph. D) Graph after two steps of coarsening.

3.5 Heuristic for Handling Dense Nodes

Graphs of some matrices arising from linear programming problems have a star-like structure (Figure 3-5.(A)). Also, some graphs may have a few dense nodes that resemble a star-like structure. The normal matching heuristics fail to preserve the properties of such graphs during coarsening and may produce a highly unbalanced coarse graph. Also, these heuristics can match only one of the star edges in one step and if the whole graph has only a few of these star-like dense nodes the coarsening process can become extremely slow (Figure 3-4). This implies a lot more levels of coarsening thus resulting in that many more graphs to be stored with many consecutive graphs being roughly the same size. Expectedly, this can use up all the memory resulting in a fatal error.

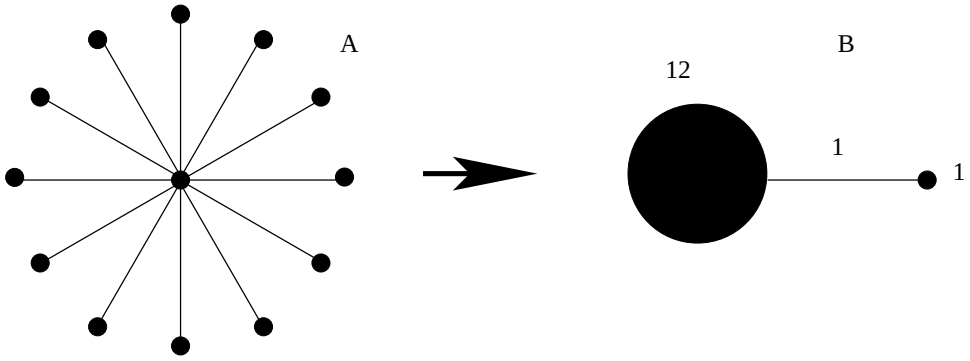


Figure 3-4. Coarsening of dense nodes. A) A dense node with degree 12. B) Node balance after 11 steps of coarsening using random or heavy edge matching.

Whenever we detect that the coarsening process is not reducing the size of the graph by a reasonable amount, we call this routine to handle the dense nodes. The process starts by calculating the degrees for all nodes and then sorting them according to degree. The nodes are then visited from the highest degree to the lowest degree. However, if we reach a node that has a degree less than the median degree or a degree less than three (default value that can be overwritten by the user) times the average degree or a degree less than one-fifth (default value that can be overwritten by the user) the maximum degree, the process is terminated and the routine returns the graph obtained so far. Here is how a dense node is handled. We add zero weight edges to connect the nodes adjacent to the dense node. However, these edges are initially added to a new intermediate graph of

the same size (Figure 3-5.(B)). This process is repeated for all the dense nodes that are handled and all the edges are added to the same intermediate graph. So at the end of the first step we have an intermediate graph that has only zero weight edges. In the next step we run the random matching heuristic on the intermediate graph. Since all the edges have the same weight, random matching heuristic is the obvious choice. Now we add this intermediate graph to the graph being handled for dense nodes. Since the added edges weigh zero, the internal degrees of the nodes are preserved in the original graph. Also, adding an edge over an existing edge makes no difference since the added sum remains unchanged. Either heavy edge matching or heaviest edge matching heuristic is now applied to the resulting graph and it is coarsened (Figure 3-5.(C)). The idea behind this heuristic is to pair-wise collapse the edges incident on a dense node so that coarsening proceeds faster and the resulting coarser graph preserves the structure and the properties of the original graph. Also, it results in a much more balanced coarsest graph and a good cut of the coarsest graph transforms into a good cut of the original top level graph. Some more processing is required in order to collapse the edges that still remain, since the first round only reduces the maximum degree by half. For this we repeatedly do zero edge matching (Figure 3-5.(C,D)) followed by heavy/heaviest edge matching until we reach a stage when the coarsening produces a graph with size not in desired proportion of the fine graph thus signaling the presence of dense nodes and a re-run of the handling routine is required. Zero edge matching, as the name suggests matches only the zero weight edges, thus enabling more pair-wise collapsing of the edges incident on a dense node. This matching heuristic matches only the second neighbor that is connected via a zero weight edge to a randomly chosen unmatched vertex. In Figure 3-5.(E) there is only one zero weight edge connecting the two nodes, hence it does not get matched and the matching shown is obtained using heavy/heaviest edge matching. Also, it distinguishes with the zero weight edges added to connect the disconnected components in the top level graph. By the nature of the edges added during dense node handling, these edges have endpoints

that share a common neighbor that is the dense node, whereas the edges added to connect disconnected components do not share this property, because if that is the case, then the disconnected components would not be disconnected. This logic is used to select the edges added for handling dense nodes while leaving out the ones that connect the disconnected components.

The idea behind the selection of second zero weight node, is to avoid the situation shown in Figure 3-5.(G). As shown in the figure, if the last zero weight edge is selected, the resulting coarse graph would not be correctly balanced. However, if one of the other edges is selected as shown in the figure, the resulting graph is more balanced (Figure 3-5.(F)).

3.6 Cutting the Coarsest Graph

We use the Greedy graph growing partitioning algorithm (GGGP) to cut the coarsest graph. This algorithm is similar to the FM algorithm except that the gains are computed incrementally unlike the FM algorithm where the gains are precomputed. Also, in this implementation we need only one bucket array that represents the frontier between the growing region and the rest of the graph. The graph is partitioned by selecting a node at random and moving it to the growing region. The cut is represented by the edges crossing over from the growing region to the other side of the graph. Based on this the gains for the vertices adjacent to the moved node are computed and the nodes inserted into the bucket array. Next, the node with the maximum gain that causes the smallest increase in the edge-cut without violating the required balance is moved to the growing region. As is done in the FM algorithm, this node is now deleted from the bucket array and the gains of its neighbors that are already in the bucket are updated. However, unlike FM, gains for those neighbors that are not already in the bucket or in the growing region are computed and the vertices added to the bucket array. The above process is repeated until no move maintains the required balance. Since this method is sensitive to the initial node selection, the whole process is repeated with four randomly chosen initial vertices. The one that gives the best cut-size is retained and the corresponding partition is returned. Since the

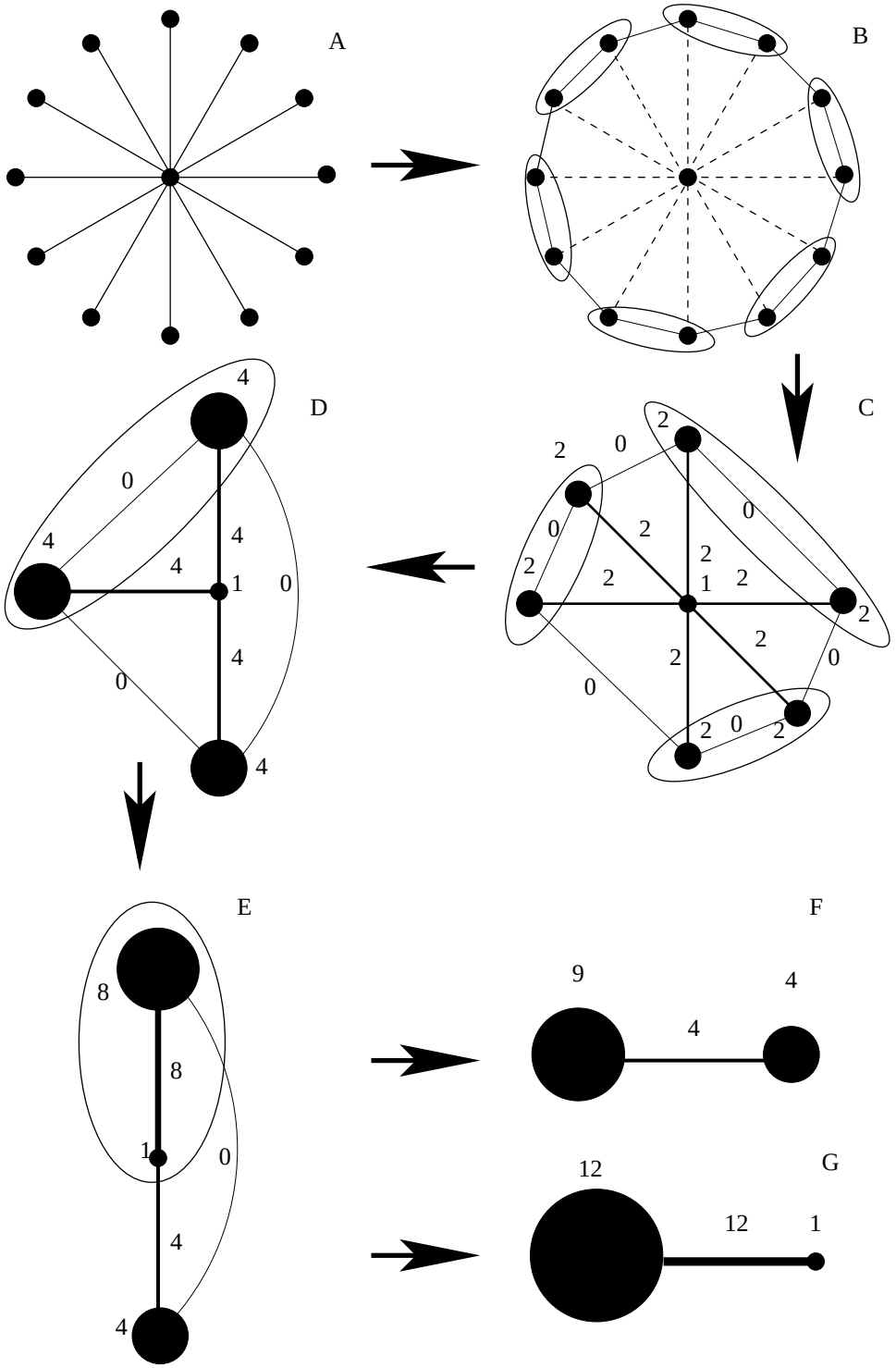


Figure 3-5. Dense nodes handling heuristic. A) A dense node with degree 12. B) The intermediate graph showing the dense node and the zero weight edges with a maximal matching. C) The original dense node after one step of coarsening also showing a maximal matching. D) After two steps of coarsening. E) After three steps. F) The final node balance after four steps of coarsening. G) When the zero weight edge in (E) is selected for matching.

coarsest graph is not more than a few hundred nodes in size, running this algorithm four times does not add much to the total partitioning cost.

3.7 Uncoarsening and Refining

During the uncoarsening and refining phase, the initial partition of the coarsest graph is projected onto the next level fine graph and is subsequently refined using the FM heuristic. This procedure is repeated until a partition is projected onto the top level graph and is refined to obtain the final partition and cut-size for the graph. The mapping vector is used to project the coarse graph partition onto the fine graph. During uncoarsening, based on the initial partition of the fine graph, the gains for the vertices are computed and the vertices inserted into the respective gain buckets. Two bucket arrays (Figure 3-6) are used, one for the left partition vertices and the other for the right partition vertices.

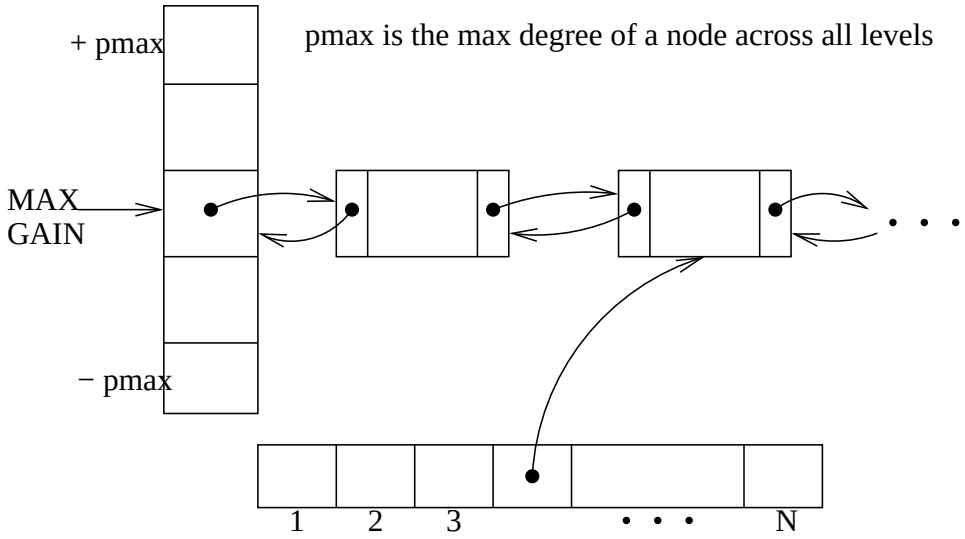


Figure 3-6. Bucket list structure

The gains are computed using the following equation:

$gain(i) = ed(i) - id(i)$,

where $ed(i)$ is the external degree and $id(i)$ is the internal degree for node i.

The internal degrees of the nodes are computed during the coarsening phase while constructing a coarse graph. At this point the internal degree of a vertex is just the sum of the edge weights of the edges incident on that vertex and the external degree is zero.

When projecting a partition onto the fine graph, for every vertex, its internal degree is subtracted by the weight of the edge connecting it to a neighbor on the other side of the partition and its external degree is increased by the same amount. At this point the coarse graph is deleted and the fine graph is passed to the FM algorithm routine for refining its initial partition. The bucket arrays are also passed to the FM algorithm to be used during refining.

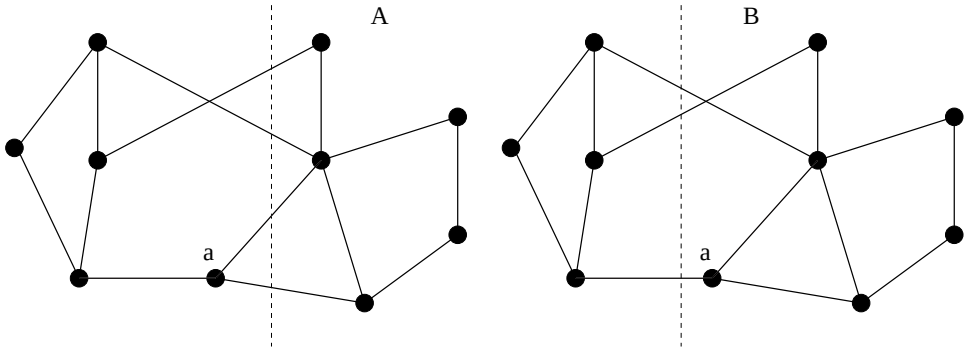


Figure 3-7. The FM algorithm. A) Graph showing a partition with three edges cut. Node 'a' has a gain value of +1. B) Graph after one step of FM. Node 'a' is moved to the other side resulting in a reduction of 1 in the cut-size.

Refining of the initial cut is done by moving nodes from one partition to the other such that it results in a decrease in the cut-size. A node to be moved is picked from the bucket array that has a higher maximum gain bucket. However, if this move fails to maintain the desired balance, the next node in the same gain bucket is tried. If none of the nodes in that gain bucket could be moved while maintaining the desired balance, the next lower gain bucket is tried. This could be from the other bucket array. If the initial partition itself is not balanced, the first pass of the refining algorithm is used as a balancing pass to get to the desired balance without caring for the cut-size. However, nodes are still moved starting from the highest gain bucket of the heavier side. A slack equal to the weight of the heaviest node (w) in the graph is provided such that each partition is within $r * total\ weight \pm w$, where r is the desired balance. This ensures that a balance can always be found especially in the case when r is 0.5. Once a balance has been obtained, it is always maintained for that graph since any node affecting the

balanced is not moved. However, later when this graph is projected onto the next fine graph the balance may not hold since the weight of the heaviest node in the fine graph can be less and hence the slack would be less. Subsequently, valid moves are made and the improvement in the cut-size is recorded. Any node that is moved is removed from its gain bucket and added to a free list that is used to re-initialize the buckets for the next pass. When a node is moved, the gains of its neighboring vertices are recomputed and the vertices moved to the new gain buckets. Nodes are moved even when they make the partition worse in an attempt to climb out of local minima. When the partition does not improve as compared to its previous best even after making around 50-100 moves, the pass is terminated and the moves are undone. If the current pass produced any improvement in the cut-size, another pass is initiated. All the nodes in the free list are added to their new gain buckets after the new gains are computed taking into account their new partition. More passes of the refinement algorithm are made until during a pass there is no improvement in the cut-size. Our experiments have shown that on an average 3-4 passes are required (Table 4-7).

CHAPTER 4
RESULTS

All the tests were run on Sun Solaris SPARC server using the MATLAB software version 7.3. Mex functions were written to interface MATLAB with the C libraries. Metis 4.0 [6] and hmetis 1.5 [7] were used for comparisons. We refer to our graph partitioner as GP when comparing with pmetis[1] and hmetis[2] .

Graph matrices were taken from the University of Florida Sparse Matrix Collection [8]. All the square matrices in the collection were sorted by the number of non-zeros and the first 1250 were selected for the various experiments. These matrices ranged in size from 15 to 1014951 in terms of the number of non-zeros and 5 to 325729 in terms of the number of rows/columns. Each un-symmetric matrix was made symmetric by adding its transpose to itself. The numerical values were ignored when doing the partitioning and the graphs were treated as undirected with no self-edges. Table 4-1 lists 10 graphs that were randomly chosen from the ones used for the experiments. These graphs are used to show a sample set of some of the results.

Various GP parameters and their values used across all experiments are listed in Table 4-2. For pmetis and hmetis, we used the default parameter values.

We used the performance profile plot (Figure 4-3) to compare the results. When comparing the cut sizes, for each graph the edge cut obtained using various methods/applications was divided by the smallest of these values and the result saved as a vector for each method/application. Finally these vectors were sorted and plotted against each other. The same process was used for comparing the run times.

[1] pmetis is the Metis interface used when not more than 8 partitions are required.

[2] hmetis is the hypergraph partitioner that can be used for normal graphs.

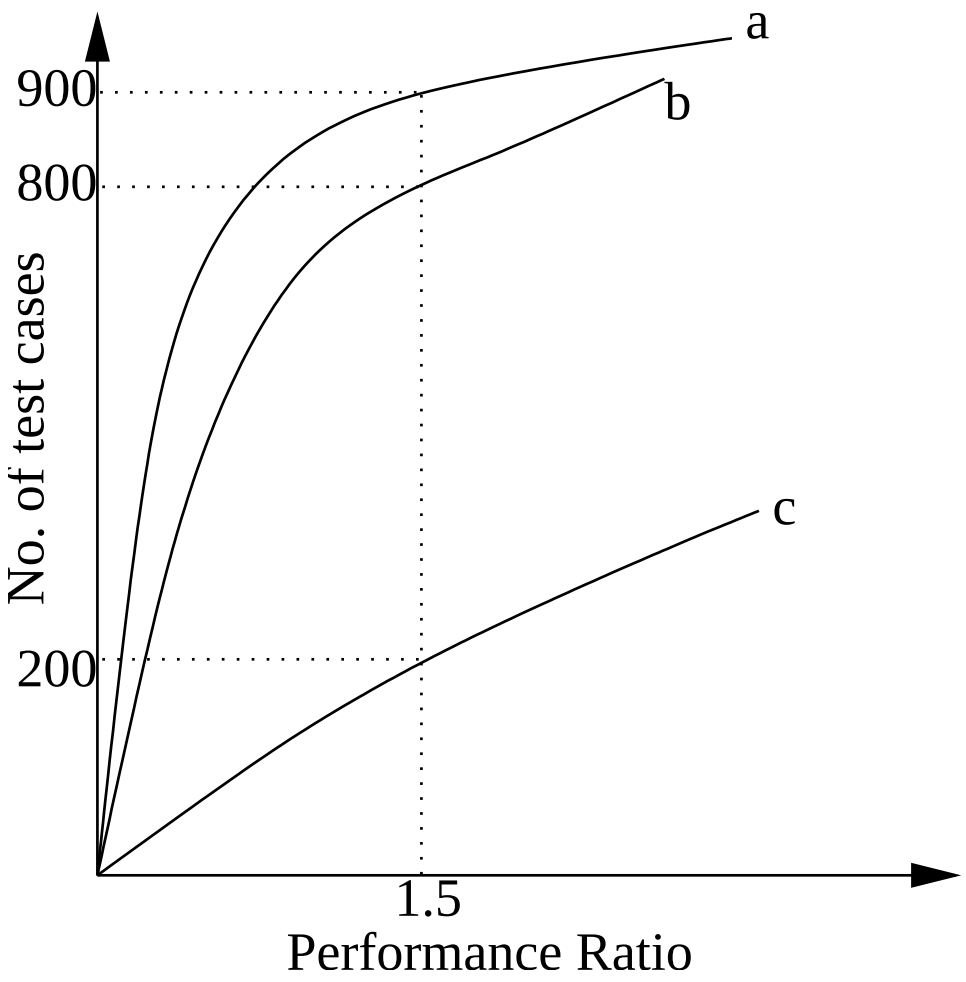


Figure 4-1. Performance Profile Definition: run times for methods a, b and c are being compared. For 900 of the test cases the time taken by method a ≤ 1.5 times the lowest run time obtained for the methods a, b and c, whereas for 800 of the test cases the time taken by method b ≤ 1.5 times the lowest run time obtained for the methods a, b and c and the number of test cases for which method c takes ≤ 1.5 times the lowest run time obtained for the three methods is only 200. Thus, here a is the best method, followed closely by b and c is the worst of the three methods that are being compared.

Table 4-1. Some of the graphs used

Name	No. of Vertices	No. of Non-zeros	Kind
GHS_indef/bloweybl	30003	109999	materials problem
HB/bcsstm26	1922	1922	structural problem
GHS_indef/aug2d	29008	76832	2D/3D problem
Bomhof/circuit_3	12127	48137	circuit simulation problem
Zitney/radfr1	1048	13299	chemical process simulation problem
Gset/G29	2000	39980	undirected weighted random graph
Sandia/fpga_dcop_34	1220	5892	subsequent circuit simulation problem
GHS_psdef/ford1	18728	101576	structural problem
GHS_indef/tuma1	22967	87760	2D/3D problem
Schenk_IBMNA/c-38	8127	77689	optimization problem

Table 4-2. Common parameters for GP

Parameter Description	Value
Ratio of number of nodes in fine graph to number of nodes in coarse graph when starting dense node detection	1.05
Level at which to resume repetitive Multi-level	$\lfloor total\ levels/2 \rfloor$
Number of consecutive moves without improvement during FM refining	100
The number of times the degree of a node is to the average degree to be classified as a dense node	3
Ratio of the degree of a node to the maximum degree when the handling of dense nodes stops	0.2
The ratio of the number of nodes in the current coarse graph to the number of nodes in the top level input graph until when heavy edge matching heuristic is used	0.01
Seed value for generating random numbers	6101975

Table 4-3. Results for pmetis, hmetis and GP(Repetitive Multi-level with 9 repetitions)

Matrix	pmetis		hmetis		GP	
	Cut Size	Run Time	Cut Size	Run Time	Cut Size	Run Time
GHS_indef/bloweybl	4520	0.2673	4502	6.5873	4579	0.6432
HB/bcsstm26	0	0.0097	0	0.0035	0	0.0307
GHS_indef/aug2d	116	0.0831	98	4.5201	107	0.1505
Bomhof/circuit_3	1314	0.0541	1840	6.9645	2443	0.8656
Zitney/radfr1	554	0.0082	501	1.3391	501	0.0669
Gset/G29	6850	0.0274	6691	5.4964	6813	0.3033
Sandia/fpga_dcop_34	26	0.0023	20	0.2421	18	0.0232
GHS_psdef/ford1	131	0.0411	124	5.0167	133	0.1092
GHS_indef/tuma1	167	0.0566	160	3.8550	199	0.1356
Schenk_IBMNA/c-38	975	0.0362	898	5.1726	1030	0.2915

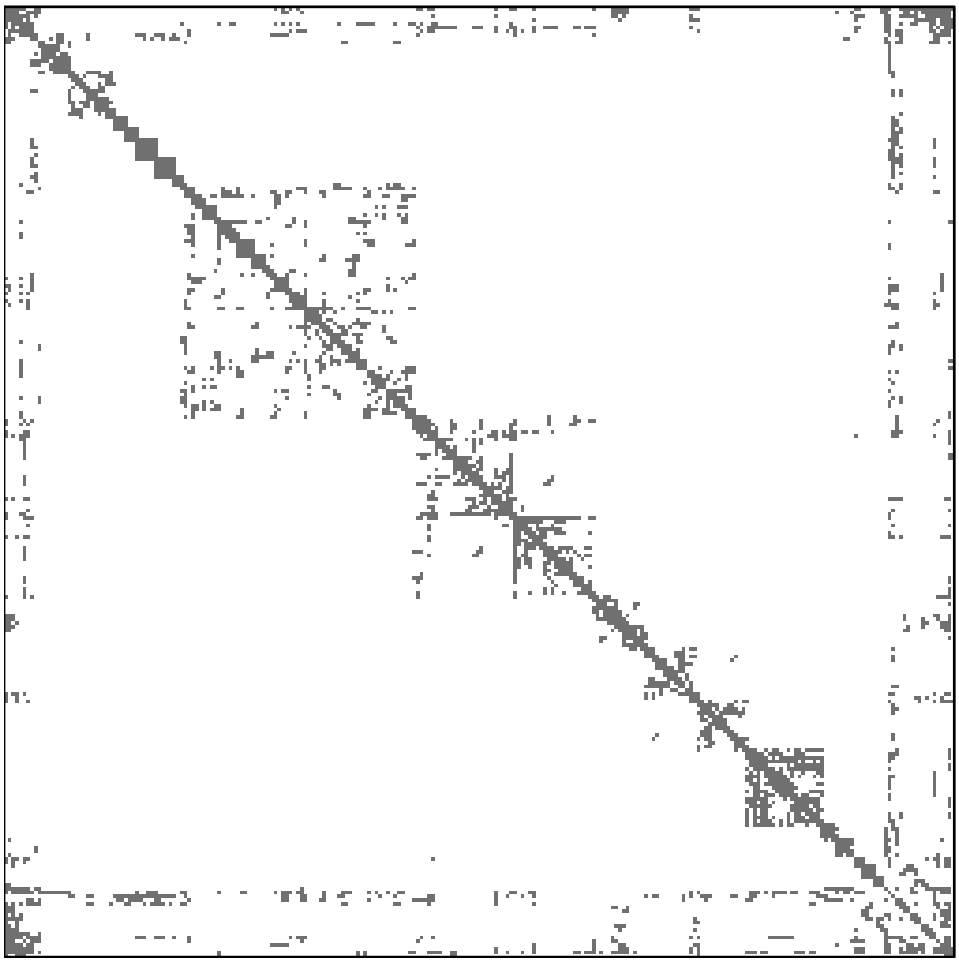


Figure 4-2. Input Matrix: GHS_psdef/ford1 displayed using cspy.m.

Table 4-4. Cut sizes for pmetis and GP(Repetitive Multi-level with 9 repetitions)

Matrix	pmetis	GP		
	r = 0.500	r = 0.499	r = 0.497	r = 0.495
GHS_indef/bloweybl	5024	5038	5014	4993
HB/bcsstm26	0	0	0	0
GHS_indef/aug2d	110	110	125	116
Bomhof/circuit_3	1377	2637	2453	2428
Zitney/radfr1	551	543	531	528
Gset/G29	6837	6865	6852	6849
Sandia/fpga_dcop_34	20	18	18	18
GHS_psdef/ford1	147	127	127	131
GHS_indef/tuma1	169	202	199	202
Schenk_IBMNA/c-38	1068	1024	1008	1141

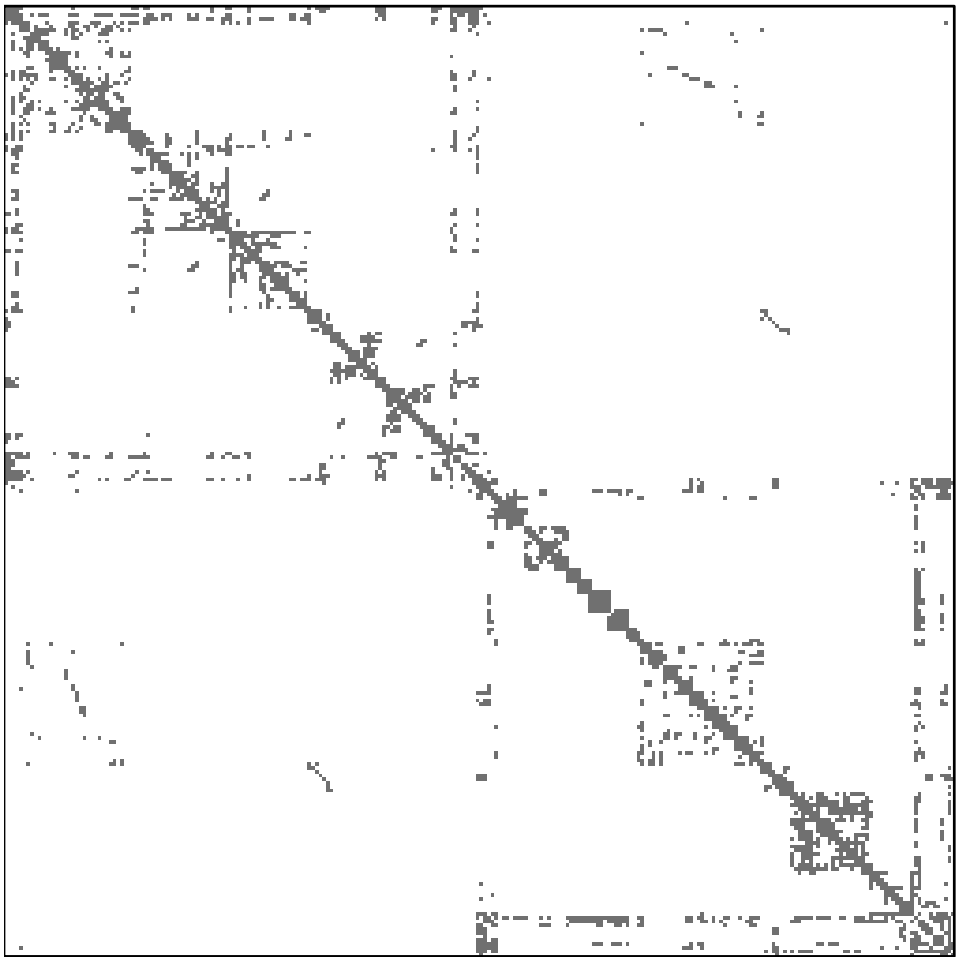


Figure 4-3. GHS_psdef/ford1 permuted using the partition vector and displayed using cspy.m; the entries in the second and fourth quadrant are the two partitions; the entries in the first/third quadrant are the edges separating the two partitions.

Table 4-5. Results for GP with simple and repetitive multi-level

Matrix	No Repetition		4 Repetitions		9 Repetitions	
	Cut Size	Run Time	Cut Size	Run Time	Cut Size	Run Time
GHS_indef/bloweybl	5091	0.2334	5108	0.3903	5050	0.6183
HB/bcsstm26	0	0.0218	0	0.0263	0	0.0303
GHS_indef/aug2d	136	0.1055	113	0.1258	113	0.1499
Bomhof/circuit_3	2747	0.2741	2575	0.5673	2575	0.9604
Zitney/radfr1	690	0.0181	690	0.0440	690	0.0737
Gset/G29	7294	0.0585	7294	0.1492	7294	0.2579
Sandia/fpga_dcop_34	62	0.0063	19	0.0157	19	0.0252
GHS_psdef/ford1	135	0.0699	135	0.0916	135	0.1206
GHS_indef/tuma1	208	0.0950	208	0.1224	208	0.1731
Schenk_IBMNA/c-38	1538	0.1398	1243	0.2144	1243	0.3400

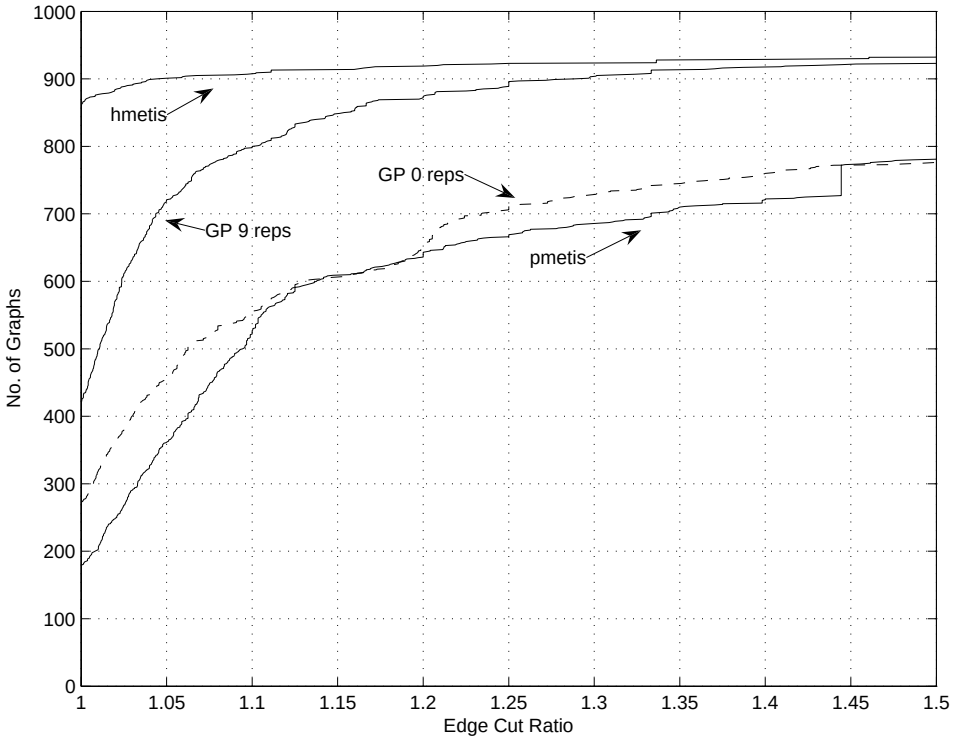


Figure 4-4. Performance Profile: comparing the edge cuts among hmetis, pmetis, GP(Repetitive Multi-level with 9 repetitions) and GP(Simple Multi-level with 0 repetitions) at r = 0.45. hmetis does better than the other three whereas GP does better than pmetis.

Table 4-6. Cut sizes for GP(Repetitive Multi-level with 9 repetitions) with and without dense node heuristic (DNH)

Matrix	No. of Vertices	No. of Non-zeros	Kind	Cut Size	
				DNH	Minus DNH
Bates/Chem97ZtZ	2541	7361	statistical/mathematical	2	9
Rajat/rajat22	39899	195429	circuit simulation	28	122
Rajat/rajat23	110355	555441	circuit simulation	186	533
Rajat/rajat19	1157	3699	circuit simulation	53	114
Bomhof/circuit_4	80209	307604	circuit simulation	2090	3576
Hamm/scircuit	170998	958936	circuit simulation	103	137
Grund/poli	4008	8188	economic problem	16	22
Schenk_IBMNA/c-43	11125	123659	optimization problem	3141	4334
Pajek/Wordnet3	82670	132964	directed weighted graph	4957	6927
Schenk_IBMNA/c-67b	57975	530583	subsequent optimization	1625	2323

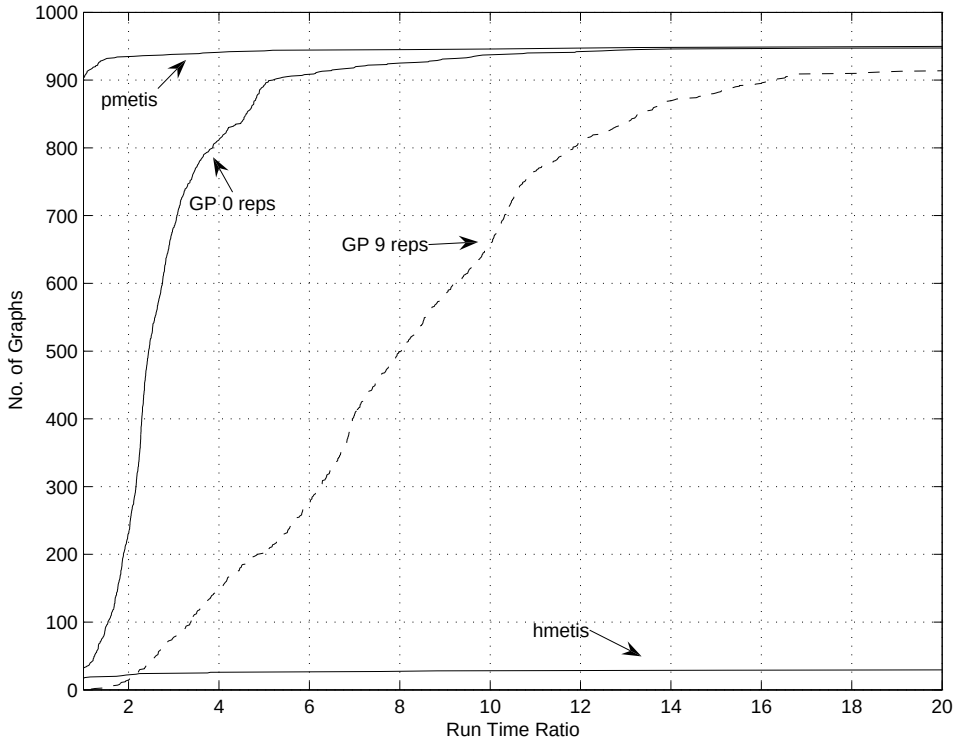


Figure 4-5. Performance Profile: comparing the run times among hmetis, pmetis, GP(Repetitive Multi-level with 9 repetitions) and GP(Simple Multi-level with 0 repetitions) at r = 0.45. pmetis does better than the other three whereas GP does much better than hmetis.

Table 4-7. Average number of passes of FM

Matrix	Passes
GHS_indef/bloweybl	2.9459
HB/bcsstm26	1.7778
GHS_indef/aug2d	2.4167
Bomhof/circuit_3	3.4333
Zitney/radfr1	2.5833
Gset/G29	4.5385
Sandia/fpga_dcop_34	2.5500
GHS_psdef/ford1	2.6667
GHS_indef/tuma1	2.8387
Schenk_IBMNA/c-38	2.1837

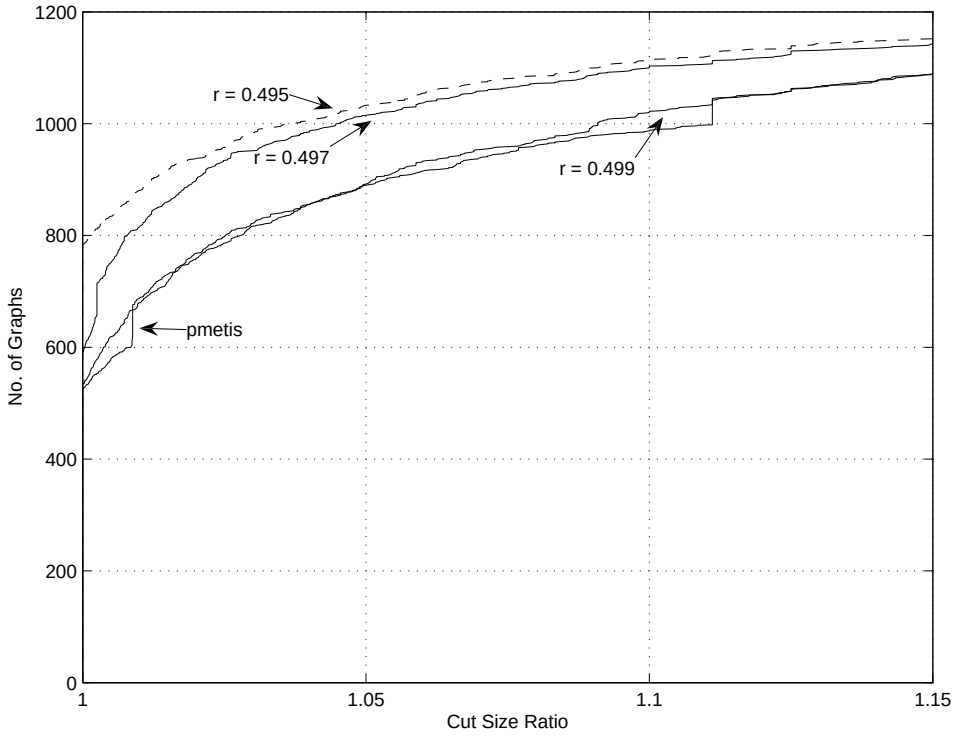


Figure 4-6. Performance Profile: comparing the edge cuts among Metis at r = 0.50 and GP(Repetitive Multi-level with 9 repetitions) at r = 0.499, 0.497, 0.495. GP almost matches Metis at r = 0.499 and does better in the the other two cases, the best being at 0.495.

4.1 Conclusion

We have a robust software that has been thoroughly tested for any memory leaks. Assertions were used at various points to assure the logical correctness of the software. Our graph partitioning software can be run with the simple multi-level and with repetitive multi-level options. These options give the user a flexibility to trade quality with time depending on what is more critical.

4.2 Future Work

4.2.1 Interfacing with QP-Part

As mentioned in chapter 2, QP-Part solves the graph partitioning problem by formulating it as a quadratic programming optimization problem. Hager et al have shown [9] that their method gives cuts superior to those of METIS, though their method is expensive when applied to the whole graph. We plan to interface QP-Part with the

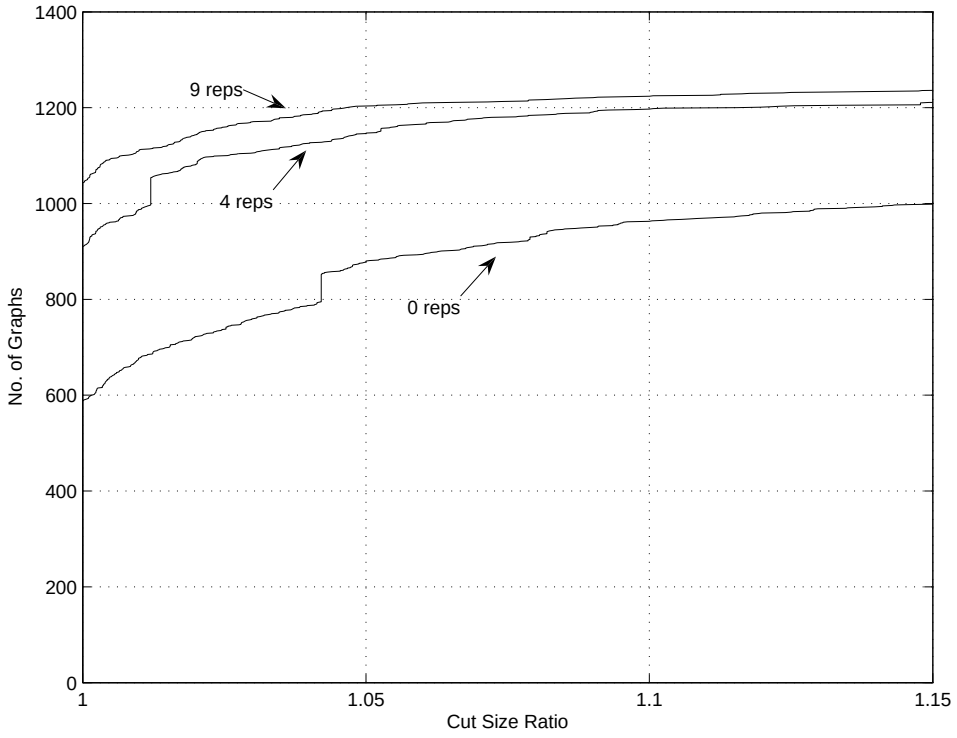


Figure 4-7. Performance Profile: comparing the edge cuts among simple Multi-level, Multi-level with four repetitions of coarsening and uncoarsening plus refining (middle) and Multi-level with nine repetitions of coarsening and uncoarsening plus refining (top).

multi-level approach such that the expensive QP-Part is used to refine the partition when the graph is small and the cheap FM algorithm is used when the graph is sufficiently large. This should give us better partitions at reasonable cost.

4.2.2 Implementing Boundary KL

In boundary KL, during uncoarsening only the nodes at the boundary of the partition are inserted into the gain buckets. Since the number of nodes at the boundary is a small fraction of the total number of nodes in the graph, this saves a lot of time. However, it is possible that this may reduce the quality of partition since some interior nodes can have higher gain values than some boundary nodes and such interior nodes do not get considered for move since they are not in either bucket list.

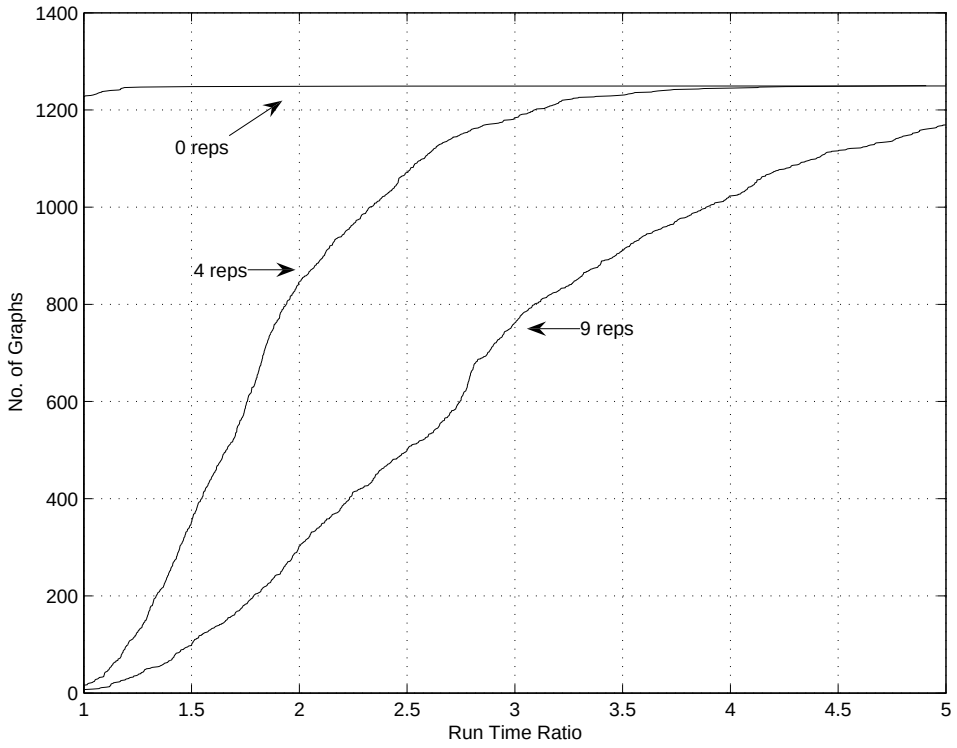


Figure 4-8. Performance Profile: comparing the run times among simple Multi-level (top), Multi-level with four repetitions of coarsening and uncoarsening plus refining (middle) and Multi-level with nine repetitions of coarsening and uncoarsening plus refining.

CHAPTER 5
PSEUDO CODE FOR GP

5.1 Top Level

```
Read (G_IN)
G_SYM+SRT = transpose (G_IN + transpose (G_IN))
G = G_SYM+SRT
G_C = Compress_Graph (G)
CC = Find_Connected_Components (G_C)
G = G_C
if size(CC) > 1
    i = 0
    for each CC do
        rep (i++) = Find_Random_Node (CC)
    end
    E = New_Graph (size(G_C))
    for j = 0 to i − 1 do
        if j < i − 1
            Add_Zero_Wt_Edge (E, rep (j), rep (j + 1))
            Add_Zero_Wt_Edge (E, rep (j + 1), rep (j))
        end
        if j > 0
            Add_Zero_Wt_Edge (E, rep (j − 1), rep (j))
            Add_Zero_Wt_Edge (E, rep (j), rep (j − 1))
        end
    end
    G = G_C + E
end
Read (DENSE_RATIO, MIN_SIZE, LEVEL, REPETITION_COUNT)
PREV (G) = NULL
Coarsen: matchings = Get_Matchings (G)
G_COARSE = Apply_Matchings (G, matchings)
PREV (G_COARSE) = G
NEXT (G) = G_COARSE
if size(G_COARSE) ≤ MIN_SIZE
    G = G_COARSE
    GOTO Cut
end
if size(G)/size(G_COARSE) < DENSE_RATIO
    G = Handle_Dense_Nodes (G_COARSE)
    GOTO Coarsen
end
Cut: partition = Cut_Graph (G)
if !G_MARK
```

```
    G_MARK = Get_Graph_From_List (G, LEVEL)
end
UnCoarsen: G_FINE = Uncoarsen (G, partition)
G_REFINE = Refine_FM(G_FINE)
G = G_REFINE
if G == G_MARK and REPETITION_COUNT > 0
    REPETITION_COUNT − −
    if partition(G) < Best_Partition
        Best_Partition = partition(G)
    end
    GOTO Coarsen
end
else if G == G_MARK
    partition(G) = Best_Partition
end
if PREV(G) ≠ NULL
    GOTO UnCoarsen
end
Return (G)
```

5.2 Handling Dense Nodes

```
Read (G_IN)
E_PREV = NULL
n = size(G_IN)
avg_deg = nnz(G_IN)/n
degrees = Compute_Node_Degrees (G_IN)
perm = Quick_Sort (degrees)
med_deg = median(degrees)
max_deg = degrees (0)
for i = 0 to n − 1 do
    if degrees (i) < 3 ∗ avg_deg or degrees (i) < 0.2 ∗ max_deg or degrees (i) < med_deg
        break
    end
    node = perm(i)
    E = New_Graph (n)
    for each adj(node) do
        Add_Zero_Wt_Edge (E, adj(node), next(adj(node)))
        Add_Zero_Wt_Edge (E, adj(node), prev(adj(node)))
    end
    E_NEW = E + E_PREV
    E_PREV = E_NEW
end
Get_Random_Matchings (E_PREV)
Return (G_IN + E_PREV)
```

REFERENCES

[1] M. Yannakakis, "Computing the minimum fill-in is NP-Complete," *SIAM J. Alg. Disc. Meth.*, vol. 2, pp. 77–79, 1981.

[2] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," Tech. Rep. TR-95-037, Computer Science Dept., Univ. of Minnesota, Minneapolis, MN, 1995.

[3] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Tech. J.*, vol. 49, pp. 291–307, 1970.

[4] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partition," in *Proc. 19th Design Automation Conf.*, Las Vegas, NV, 1982, pp. 175–181.

[5] W. W. Hager and Y. Krylyuk, "Graph partitioning and continuous quadratic programming," *SIAM J. Alg. Disc. Meth.*, vol. 12, pp. 500–523, 1999.

[6] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," Tech. Rep., Computer Science Dept., Univ. of Minnesota1, Minneapolis, MN, Sept. 1998.

[7] G. Karypis and V. Kumar, "hMetis: A hypergraph partitioning package," Tech. Rep., Computer Science Dept., Univ. of Minnesota, Minneapolis, MN, Nov. 1998.

[8] T. A. Davis, "University of Florida sparse matrix collection," www.cise.ufl.edu/research/sparse.

[9] W. W. Hager, S. C. Park, and T. A. Davis, "Block exchange in graph partitioning," in *Approximation in Complexity and Numerical Optimization: Continuous and Discrete Problems*, P. M. Pardalos, Ed., pp. 299–307. Kluwer Academic Publishers, 2000.

BIOGRAPHICAL SKETCH

Pawan Kumar Aurora was born in Kanpur, India, on October 6, 1975. He graduated from the Birla Institute of Technology, Mesra, Ranchi in 1998 with a degree in Computer Science. For seven years Pawan worked in the IT Industry initially for Tata Consultancy Services in Mumbai, India and later for IBM Global Services in Pune, India before relocating to Gainesville, Florida, USA to pursue graduate study in computer engineering. After completing his M.S., Pawan will pursue his PhD in computer science.