UNIVERSITY OF OSLO
Department of Informatics

A study of different matching heuristics

Cand. Scient. thesis

Jan Kasper Martinsen

May 1, 2005

Preface

This thesis is written at the University of Oslo, Department of Informatics. I would like to thank my supervisors Noureddine Bouhmala and Xing Cai, for valuable input, guidance and their patience. I would also like to thank fellow students, friends and family.

Contents

Abstract

A well-known combinatorial optimization problem is the graph partitioning problem. Since solving it optimally requires very much time, we have to settle for approximated methods. One such method is the multilevel k-way partitioning scheme. The overall idea is the following: The size of the graph is reduced during a process known as coarsening, where smaller and smaller graphs are made. Then a k-way partition is found from the much smaller graph and the graph and the partition is projected back to the original size.

In this master thesis, we study a central part of the multilevel k-way partitioning scheme, coarsening. In order make smaller and smaller graphs we use a technique known as matching. We introduce a new matching heuristic, global greedy heavy edge matching and perform a large number of experiments comparing it to other matching heuristics.

Our results include the discovery of two new partition vectors for the graph partitioning archive [29], and that using global greedy heavy edge matching generally produces better results than already existing matching heuristics.

1 Preleminaries

1.1 Graph theory

Graph theory is the mathematical study of structures called graphs. The term graph is a bit confusing in relation to mathematics, because it refers to both graph of a function (i.e. a plot) and a kind of networks structure

Informally, a graph consist of a number of points, and a number of lines between these points. We refer to the points as nodes [1], and the lines as edges.

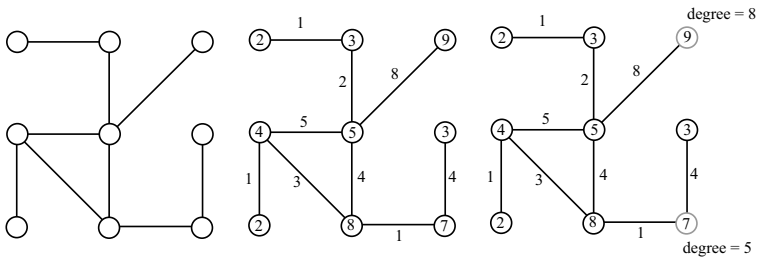


Figure 1: The leftmost image is a graph where the circles represent the nodes, and the lines represent the edges. The image in the middle is a weighted graph, where the number inside the node is the node-weight, while the number close to the edge is the edge weight. In the rightmost image we have marked the degree of two of the nodes.

Formally a graph G consist of two finite sets: a set $N(G)$ of nodes and a set $E(G)$ of edges, where each edge is associated with a set consisting of two nodes known as endpoints.

There is also an extension of G, known as a **weighted graph**, where a weight (usually a integer) is associated with each node, edge or both.

Graph can also be grouped into types, for instance a graph is said to be **connected**, if there is a path from any nodes to all the other nodes in the graph.

We also define a function $w(node)$ and $w(edge)$ to return the weight of edges and nodes.

We define a nodes **degree** to be the sum of the weight of the edges that connects a node to its incident nodes. We also define the set of nodes incident to a node as the set of neighbours of a node.

[1] A common name for these points is also *vertices*, however we will avoid to use this name, since this term is also used in geometrical objects.

1.2 Graph partitioning problem (GPP)

The GPP is defined as follows; Given a unweighted graph $G = (N, E)$, partition N into k subsets, $N_1, N_2, \ldots, N_k$ such that $N_i \cap N_j = \emptyset$ for $i \neq j$, $|N_i| = |N|/k$ and $\bigcup_i N_i = N$ and the number of edges of whose incident nodes belong to different subsets is minimized.

A partition of G, is typically represented by a partition vector P of length $|N|$, such that for each node $n \in N$, $P[n]$ is a number (integer) in the range between 1 and k, indicating the partition which n belongs to.

Given a partition vector P of a graph G, the number of edges whose incident nodes belongs to different partitions is known as the **edge-cut**.

The **load imbalance** is defined as the relationship between the ideal partition weight, the sum of node-weight divided by k and the heaviest partition P_{max}. More formally we define load imbalance as;

$$\text{load imbalance} = \lfloor 1 - \frac{|N|/k}{|P_{max}|} \times 100 \rfloor \tag{1}$$

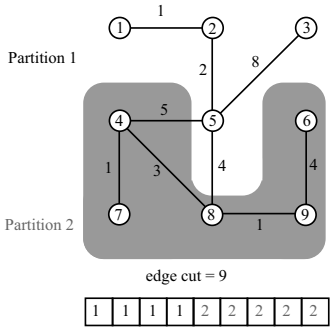


Figure 2: Example of a partitioned graph with edge-cut = 9. The nodes are numbered from 1 to 9. Below the graph is the partition vector, where each node belongs to either partition 1 or 2.

The GPP belongs to a problem class of combinatorial optimisation problems known as NP-hard [10]. NP-hard basically means that there doesn't seems to be any algorithms with polynomial run-time for solving this problem optimally, rather it must be done by searching through all the possible combinations (brute-force) do find the optimal solution.

1.3 Applications of GPP

The GPP has applications in many real life problems. Such applications include scientific computing, task scheduling and VLSI design. Some examples are domain decomposition for minimum communication mapping in parallel execution of sparse linear system solvers, mapping of spatially related data

items in large geographical information systems on disk to minimize disk I/O requests, and mapping of task graphs to parallel processors.

1.4 Adjacency matrix

One way to represent the graph is through a so called adjacency matrix. An adjacency matrix is a special kind of square matrix, where the rows and columns are labeled by graph nodes.

Lets AM be such a matrix, if the matrix element at $AM_{n_1,n_2} \neq 0$ then there is an edge (adjacency) between the nodes n_1 and n_2. An example of such a matrix can be seen in Figure 3

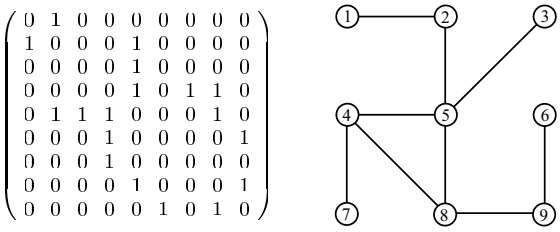


Figure 3: The matrix to the left is the adjacency matrix of the graph to the right.A value of 1 at position $AM_{i,j}$ indicates there is an edge width node i and j as endpoints.

One way to store adjacency matrices, are with the compressed row storage format (CRS) [16]. The format is organized the following way;

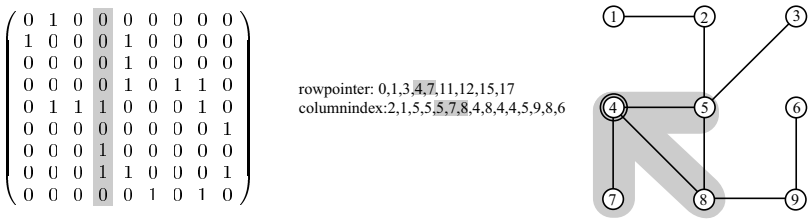


Figure 4: The relationship between an adjacency matrix (leftmost) and the graph (rightmost) with the CSR format (middle).

Figure 4 illustrates the relationship between the adjacency matrix, the graph and the CRS format. If we select node 4 from the rightmost graph, we see that its adjacency information is found in column 4, where each of the 9 rows determine which nodes of the graph node 4 is adjacent to. Looking at the CRS format, the same information is found by accessing the *columnindex* from $rowpointer[4+1]$ to $rowpointer[4]$ (i.e. the values

$5, 7, 8$). Looking at the rightmost graph we see that this are the nodes that are adjacent to the node 4.

1.5 Organization of thesis

This thesis is divided into five chapters; a general introduction, a review of already existing graph partitioning algorithms, a special form of graph partitioning algorithms, known as as multilevel k-way partitioning,which includes, our new heuristics, a series of numerical experiments where we examine the properties of the new algorithms, and last the conclusion of our work and ideas for future research.

2 Previous work

2.1 Introduction

Creating a program that solves the GPP optimally isn't very hard. One can simply search through all the possible partitions to find the optimal one. However, there is a catch. The number of possible partitions grows exponentially with the size of the graph, so this might take very much time.

There have been developed GPP methods that find suboptimal partitions in a reasonable amount of time, but where the edge-cut and load imbalance can be quite far from the optimal partition.

In this chapter, we present three classes of algorithms for solving GPP; Algorithms that creates partitions from scratch, algorithms based on improvement of existing partitions and so-called multilevel algorithms.

2.2 Algorithm for solving GPP from scratch

Two simple methods and building blocks of the multilevel GPP method, which we treat extensively, are greedy partitioning and the recursive bisection scheme.

2.2.1 Greedy GPP algorithm

One simple and fast method for solving GPP is to use the greedy algorithm proposed by [8].

Initially all nodes are set to *available*. The algorithm starts by selecting an *available* node from the graph with the minimal degree. This node is placed in a list. A node is then removed from the list, marked as *unavailable* and placed in a given partition. Next, the incident *available* nodes of this node are added to the list and then we repeat the procedure, removing a node from the list, marking it as *unavailable*, placing it into the given partition, and adding its *available* nodes to the list, until the appropriate number of nodes are in the given partition.

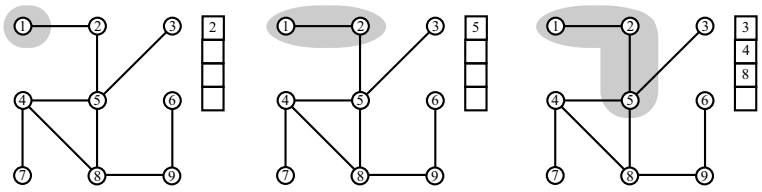


Figure 5: Three stages of the greedy partitioning algorithm.

There have also been done some work on creating variants of the greedy partitioning [3], for instance by being more selective when adding incident nodes to the list.

2.2.2 Recursive bisection scheme

Much work have been invested into a version of the GPP where the graph is partitioned into 2, known as bisection. Bisection is well studied [2], since we can by a scheme known as recursive bisection extend the bisection so it can create k partitions.

The idea is the following; We start off with a graph G which we want to divide into k partitions. G is first bisected into two parts, G_{left} and G_{right}. We repeat the process above on G_{left} and G_{right} until the graph has been divided into k partitions.

It is obvious that simply dividing the graph repeatedly into equal parts will not work for all ks. For instance, we have problems if k is an odd number. The solution presented is a variant of the one suggested in [5]. The idea is to allow G_{left} and G_{right} to have different size as in equation 2 and 3.

$$|G_{left}| = \lfloor |N| \times \tfrac{2}{k} \rfloor \tag{2}$$

$$|G_{right}| = |N| - |G_{left}| \tag{3}$$

Figure 6 is an example of recursive bisection, where k is an odd number. The graph has 133 nodes, which we want to divide into 5 partitions. We start by dividing 2/5 of the 133 nodes into G_{left} and 3/5 of 133 nodes into G_{right}. Next step is to divide G_{left} and G_{right}, this time k is not 5, but the numerator of the previous fraction, such that the right and left size of G_{left} will be 1/2, while the left and right size of G_{right} will be 1/3 and 2/3. At this point we have to finished left and right of G_{left} and the left side of G_{right}, since there is no point to continue dividing by 1. The right part of G_{right} is splitted into two new partitions, resulting in a total of 5 subsets, completing our partitioning task.

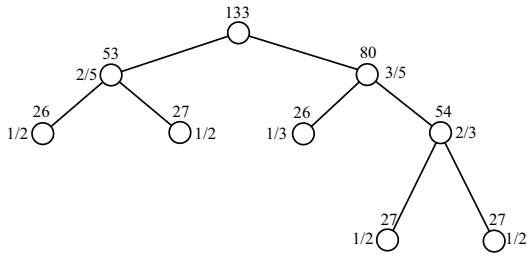


Figure 6: The tree from the recursive bisection scheme, where the number of nodes are the number above the circle and the size of the subset relative to the node above, is next to the circle

There is some obvious properties of recursive bisection, for instance given

that the graph is unweighted, then the difference between the heaviest and lightest partition will be at most 1 node.

Recursive bisection has been used quite extensively for GPP. Some methods based on the recursive bisection method are; Recursive coordinate bisection [32], scattered partitioning [24] and recursive graph bisection [25] It has also been shown that recursive bisection can produce partitions that can be quite far from an optimally partitions [26] with respect to edge-cut and load imbalance

2.3 Iteratively improving an initial partition

Both greedy partitioning and recursive bisection might return partitions that are far away from optimal solutions. Since we are not certain that a partition is the optimal solution, we say that it is in a **local optima**. The optimal partition is known as the **global optima**, and in case the partition is the optimal solution, we also say that global optima also is in a local optima. Different partitions, have different local optimas and the idea of the iteratively improving algorithms is to move the partitions from one local optima to a lower local optima.

Below we briefly presents algorithms that improves an existing partition such as a simple improvement algorithm and Kerninghan-Lin algorithm.

2.3.1 Simple improvement algorithm (SI)

One simple way to go from one local optima to a lower local optima is the following; Go through all the nodes by random. If two incident nodes are in different partitions and swapping them leads to decrease in edge-cut, then a swap is made.

2.3.2 Kerninghan-Lin (KL)

KL [21] takes the SI algorithm a step future. SI only swap nodes, if there is an immediately decrease in the edge-cut. However, allowing swapping of nodes, that increase edge-cut, might lead us to a move that decreases the edge-cut later on, or to use the terminology above, it might increase our ability to climb from one local optima to a lower local optima. This is the idea behind KL.

For simplicity we present the algorithm as a bisection. We have two lists that contains the nodes from partition. We sort the lists based on the decrease in edge-cut swapping this node will have on the edge-cut, such that nodes that decreases the edge-cut the most is stored on top of the list.

The algorithm starts by selecting the node that decreases edge-cut most, swap it, update the list and the edge-cut and the swap is recorded. The algorithm continues until the lists are empty. Then we go through the list of edge-cuts and select the edge-cut that is smallest, the swaps are made until

this edge-cut is reached and the procedure above is repeated once more. The algorithm terminates when we are unable to decrease the edge-cut any future. KL has been modified into several variants, for instance improvement of runtime [9] and to handle k way partitioning [12].

2.3.3 Other iterative methods

We also note that there are several other iterative methods for improving existing partitions, popular methods include genetic algorithms [15], simulated annealing [7], ant colonization [6] and tabu search [11].

2.4 Multilevel paradigm

Our work is based around so-called multilevel algorithms. The multilevel scheme is based on the following, the smaller an instance of a problem is, the faster it is to solve it.

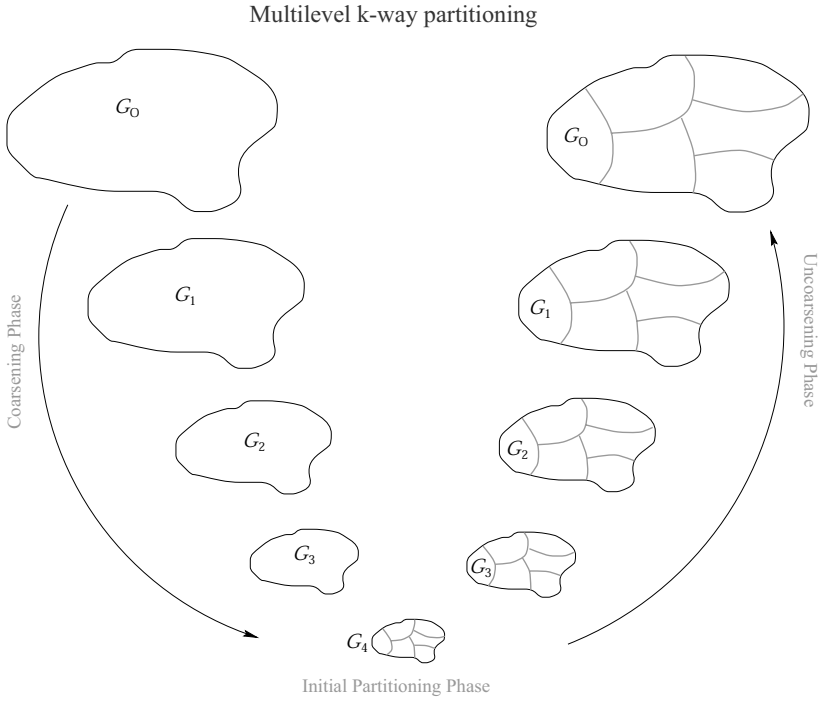


Figure 7: The three phases of the multilevel algorithm (figure from [19])

Multilevel GPP consists of three phases, coarsening, initial partitioning and uncoarsening. First a smaller and smaller graph is generated by combining nodes and edges. Then a partition of the significant smaller graph is found. Finally the partition is projected and refined until a partition of the original size is found.

2.4.1 Coarsening

During coarsening, a sequence of smaller graphs $G_i = (N_i, E_i)$ is constructed from the original graph $G_0 = (N_0, E_0)$, such that $|N_i| < |N_{i-1}|$. This continues until the number of nodes is equal or smaller than a certain threshold t.

In order to reduce the size of the graph we combine nodes and edges. This is done with a technique from graph theory known as **matching**. Formally a matching is defined the following way;

A matching of a graph is a subset M of the edges, where all of the endpoints (i.e. the nodes) are distinct. An example of a matching can be seen in Figure 8.

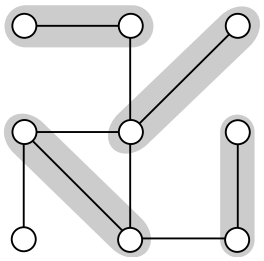


Figure 8: An example of a matching of a graph, where the grey fields indicate that the edges are a part of a matching.

A **matching heuristic** is the order in which we add edges to M. For instance the size of the matching is determined by how and in which order the edges are selected. as seen in Figure 9.

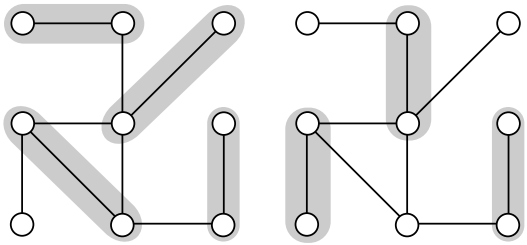


Figure 9: Example of two different types of matchings. We see that which order the edges are selected does matter for the size of the matching.

We define that if $|M| = |N|/2$, then the matching is a so called **perfect matching**. Not all graphs have a perfect matching, for instance not all graphs have an even number of nodes and the topology of the graph might prevent us from making a perfect matching. For instance in Figure 8, the number of nodes are 9, and therefore a perfect matching cannot exist.

A matching such that it is not possible to add more edges is a **maximal matching**. Note that different matching heuristics, as seen in Figure 9, yields different maximal matchings.

Matching in the multilevel setting is used to create smaller graphs. We combine the two endpoints of a matched edge into a new node, known as a **multinode**. The matched edge is removed from the graph (or 'hidden' inside the multinode). The weight of the new multinode is the sum of the weight of the two endpoints. This means that if the node-weight was uniformly set to 1 to begin with, it could after the first coarsening level vary between 1 and 2.

The set of neighbours of the multinode will be a combination of the incident nodes of the two endpoints that forms the multinode. If both the endpoints are incident to the same node, then the edge from the multinode will be the sum of edgeweight of both these edges. Such an example can be seen in Figure 10, where node 4 and 8 are the endpoints in a matched edge. Both node 4 and 8 are incident to node 5. Therefore the edge-weight of the edge between the multinode formed by 4 and 8 and the multinode formed by 5 and 3 will be the sum of the edges between node 4 and 5 and 8 and 5.

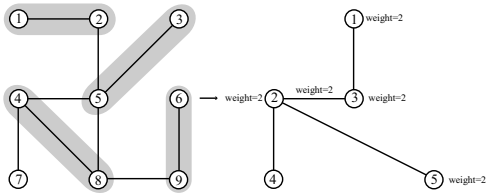


Figure 10: Example of matching, and how nodes and edges are combined

Unlike nodes, edges can be combined several times during a coarsening level. We found out that given that the edge-weight is initially uniform, then the weight of an edge after one coarsening level could be from 1 to 4(

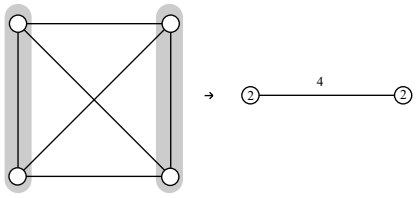


Figure 11: Example of how the edge-weight can be between 1 and 4

In Figure 11), the four endpoints create two multinodes and since they are incident to each other, the edge-weight between the two multinodes will be 4. This seems to be the maximal increase in edge-weight, given that the

edge-weight is uniform[2]

2.4.2 Initial partitioning

Once we complete the coarsening phase, we find an initial partition. This can be done a number of ways, for instance there is the possibility to coarse until the number of nodes is equal to the number of partitions. However, we discovered that it is quite expensive to coarse until the number of nodes are this small, since the size of M decreases as the number of coarsening levels increases. We also found out that the weight of the nodes was very inhomogeneous, so initially the load imbalance was quite high.

We use a variant of recursive bisection, known as multilevel recursive bisection (which will be introduced at the end of this chapter) where a variant of greedy partitioning [19] is used for the bisection.

.

2.4.3 Uncoarsening

The phase in the multilevel algorithm where we project the partition vector successively from $G_i, G_{i-1} \cdots G_0$ is known as uncoarsening.

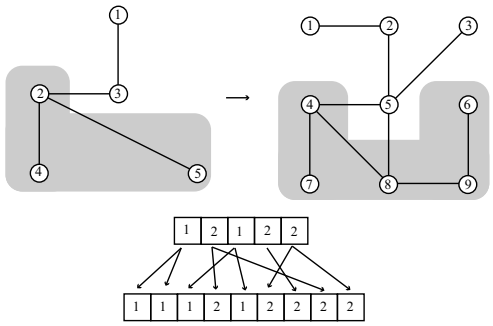


Figure 12: How the partition is remapped during the uncoarsening.

In Figure 12 we see the uncoarsening process. The multinode is separated into two nodes and these nodes are placed in the same partition as the multinode.

Since we added node and edge-weight when combining nodes into multinodes, we know that when we project the partition back to its original size, the edge-cut and load imbalance can be no worse than the initial partition.

But as we expand the graph from $G_i, G_{i+1} \ldots G_0$, we can for each uncoarsening level try to decrease the edge-cut and load imbalance of the graph. This process is known as refinement, which is related to the iterative improvement of the partition discussed in the previous section.

[2]If there exist a larger possible weight, we would be happy to hear about it.

We will present one refinement schemes, greedy refinement (GR) but before we present GR, we introduce some key concepts and definitions. A graph G_i has a partition vector P_i. For each node $n \in N_i$ we define the neighbourhood of n as the set of the partitions adjacent to n. If n is only adjacent to one partition, then we say that n is an interior node, else we say that it is a boundary node.

We formalizes this through external degree (ED) and internal degree (ID).

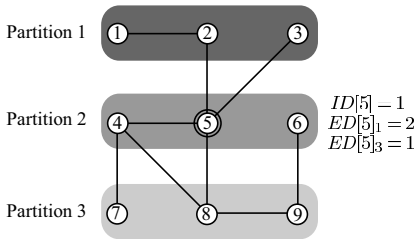


Figure 13: External and internal degree of node 5.

In Figure 13 we see the ED and ID of node 5. The node has two incident nodes that are in partition 1, one incident node that is in partition 3, and one that is in partition 2. From this we introduce the concept of *gain*.

For instance, moving node 5 from partition 2 to partition 1 has a *gain* equal to $gain[5]_1 = ED[5]_1 - ID[5] = 2 - 1 = 1$

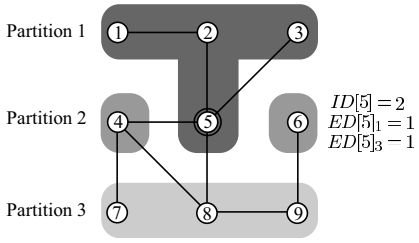


Figure 14: Moving node 5 to partition 1 to decrease the edge-cut

In Figure 14 we see the result of moving node 5 to a neighbouring partition. The *gain* is 1, which means that we have reduced the edge-cut by one, which can be verified by comparing Figure 13 and Figure 14.

However we note that the movement of node 5 caused partition 1 to contain one more node than the other partitions. We cannot allow movement that decreases the edge-cut, while increasing the load imbalance. If we kept on moving nodes that gave a positive gain, we would end up with a edge-cut

that was 0, however, we would also end up with one partition containing all the nodes, and two empty partitions.

This is the motivations for the **balancing condition**, which constrains the movement of nodes such that the load imbalance doesn't increase. Let PW be a list of length k, which contains the weight of each partition. Set $W^{min} = 0.9|N|/k$ and $W^{max} = 1.3|N|/k$[3].

A node can be moved from partition a to partition b only if;

$$PW[b] + w(n) \leq W^{max} \tag{4}$$
$$PW[a] - w(n) \geq W^{min} \tag{5}$$

The greedy refinement (GR) algorithm is a variant of the SI algorithm presented in section 2.3.1. Then the algorithm works the following way; We start by selecting randomly one of the boundary nodes. This node is moved to the partition such that the gain is maximized, while not violating the balancing condition. Once a node is moved, we are not allowed to move it again during the same iteration. If moving a node doesn't increase the edge-cut, but improves the load imbalance then it is also moved.

1. $ED[node]_b > ID[node]$ and $ED[node]_b$ is maximum among all the partitions that the node is adjacent to.

2. $ED[node]_b = ID[node]$ and that $PW[a] - PW[b] > w(node)$.

More formally a node is moved if either one of the conditions above are fulfilled. After one node is moved, the ED and ID for its incident nodes are updated.

The motivation for using this heuristic instead of a more advanced improvement algorithm such as KL is the following; KL is able to climb out local minimas because of its list of moves. That allows us to move entire clusters of nodes across the partition boundary. However, in the multilevel context we are not moving nodes, but entire clusters of nodes (multinodes). That means that the 'lookahead' in algorithms like KL is less important, because we are moving nodes that are likely to be moved anyway.

A refinement method that increases GR capabilities of climbing out of local optimas, is global Kernighan-Lin (GKLR) [18]. The idea is the same as for GR, only that we add the moves to a list as in KL, such that we are allowed to make moves with negative gain.

[3]We multiply with 1.3 to allow some imbalance, since by allowing some imbalance improves our ability to climb out of local optimas. We multiply with 0.9 to be sure than none of the partitions contains too few nodes.

2.4.4 Multilevel recursive bisection

Algorithms such as Kernighan-Lin, genetic graph partition, simulated annealing, tabu search and recursive bisection all have multilevel extensions [1, 22]. Below we briefly present the multilevel variant of recursive bisection (MLRB) and then we devote an entire chapter to the multilevel GPP algorithm which our work is based around, the multilevel k-way graph partitioning (MLKP).

The name multilevel recursive bisection could be misleading[4]. One could be lead to think that the graph is coarsen, then recursive bisection is used to find a partition of the smaller graph, then it uncoarsend to the original size. However, this is in fact very similar to MLKP which we will consider in the next chapter.

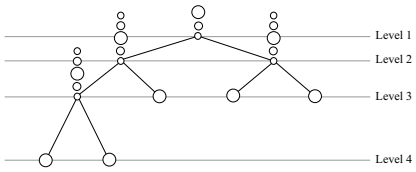


Figure 15: An illustration of the multilevel recursive bisection algorithm.

The multilevel step is preformed on each level of recursive bisection three (Figure 15). Before the graph is bisected into G_{left} and G_{right}, we coarse the graph, find a bisection of the much smaller graph, and project it back to G_{left} and G_{right}. Then G_{left} and G_{right} are coarsed, both the graphs are bisected into a left and right side each, the solution is projected back to the left and right side of both G_{left} and G_{right} until we have k partitions.

[4] A better name could perhaps be recursive multilevel bisection

3 Multilevel k-way partitioning (MLKP)

In MLKP there is only one multilevel step.First the graph is coarsened, a partition of the much smaller graph is found and the partition is projected back to the original size.

Compared to MLRB, this approach is faster, since we are only doing one multilevel step, rather than one for each bisection. Another property that was 'discovered' was that this process also gives a more global view of the partition process. For instance the graph is coarsened from $G_0, G_1, \cdots G_i$. Then during the initial partitioning, one do not move nodes to different partitions, but rather clusters of nodes, and together with refinement during the uncoarsening one could argue that this gives a better view of the partitioning process than MLRB.

In this section we present six different matching heuristic, random matching, heavy edge matching, heavy edge minimization matching, gain vertex matching, local greedy heavy edge matching and global greedy heavy edge matching, where global greedy heavy edge matching is our contribution.

3.1 Coarsening phase

During each coarsening level, nodes and edges are combined successively. This is repeated until the number of nodes in the coarser graph is less than a **coarsening threshold**.

In this section we look at how this coarsening threshold is computed, why and how we set a **matching constraints** and introduce the concept of **matching ratio**.

3.1.1 Coarsening threshold

For each coarsening level, we start by creating a maximal matching, then a smaller graph is constructed from this matching. This process is repeated until we reach a certain threshold (Equation 6) or we are unable to reduce the size of graph after the matching with less than 10%.

$$\text{coarsening threshold} = \begin{cases} x = \frac{|N|}{40} \times log_2(k) & x > y \\ y = 20 \times k & x < y \end{cases} \tag{6}$$

From equation 6 we see that the threshold is determined by the number of partitions we wish to find, and that x is valid when k is small. We also see that the threshold increases as the k increases, which is natural, since we want the coarser graph to be larger if we are to partitioning it into more partitions.

We also note that we only check if the number of nodes reaches the threshold once for each coarsening level. This is obviously to save some execution time. If we where to coarse the graph until the number of nodes

where exactly equal to the threshold then we would need to test each time we matched an edge. This also means that we always end up with a graph at the coarser level that has less or equal number of nodes compared to the threshold.

In addition, we exit the coarsening loop, if the decrease in nodes from one coarsening level to another is less than 10%.

3.1.2 Matching constraint

To avoid problems with nodes with too high node-weight, for instance nodes with a weight higher than $|N|/k$ (the ideal weight of one partition) we introduce a matching constraint.

$$\text{matching constraint} \ = \ \frac{|N|}{\text{coarsening threshold}} \tag{7}$$

As seen in Equation 7 we only add an edge to a matching, if the sum of its endpoints are less than equation 7.

3.1.3 Matching ratio

The matching ratio describes the reduction in nodes between two coarsening level, formally it is defined as in [4].

$$\text{matching ratio} = \frac{|N_i|}{|N_{i+1}| \times 2} \tag{8}$$

Where the numerator is the number of nodes after i coarsening iterations, and the denominator number of nodes after $i + 1$ coarsening iterations. If for instance the matching ratio is equal to 1, then there have been a perfect matching.

3.2 Matching heuristics

Matching heuristics are rules on how we select edges to be included in a matching. The most common way to add an edge to a matching is the following; We select an unmatched node and then one of it unmatched incident nodes. These nodes are then endpoints in an edge. The motivation is obvious, since there are fewer edges than nodes.

3.2.1 Random Matching

First presented in relation to the multilevel GPP in [14], random matching (RM) is a simple, but fast way to create a maximal matching. First a node n_1 is selected by random, then we by random select one of its unmatched incident nodes n_2. Although named "random" matching, it is a quite powerful heuristic, since it uses the topology of the graph to create matchings. We

could easily imagined a random method which was less powerful in terms of reducing the size of the graph quickly, where two nodes where selected, and if they where adjacent, then a matching is produced.

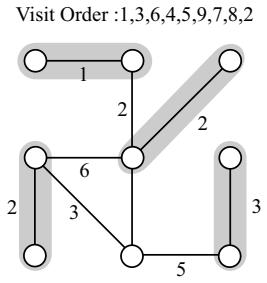


Figure 16: Random matching

RM have a complexity equal to $O(E)$.

3.2.2 Heavy Edge Matching

Originally introduced in [19] heavy edge matching (HEM) is a matching heuristic that aims to reduce the edge weights of the coarser graph. it is likely that if we select the edges with heavy weight, then the coarser graph will have smaller edge-weights, which ultimately could lead to a smaller edge-cut when partitioning the coarsest graph [17].

The heuristic works in the following way; A node n_1 is selected by random[5] Then the incident node n_2 of n_1 is chosen such that the edge-weight between n_1 and n_2 is maximized over all its unmatched incident nodes.

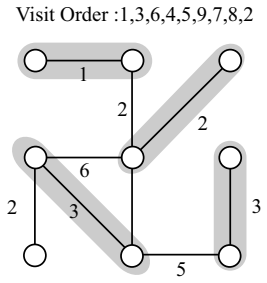


Figure 17: Heavy edge matching

First assume that the graph initially have an uniform edge-weight. For the coarsening level, RM and HEM will perform identically. We note that

[5]METIS graph partition package, have a heuristic where a node is not chosen by random, rather from the number of incident nodes it has.

for the rest of the coarsening levels it is likely that HEM will have a lower matching ratio then RM. To understand why, consider the following example, where the edge-weight is not uniform.

When two nodes n_1 and n_2 are matched, then the number of neighbours of $n_{(1,2)}$ is equal or larger than either n_1 or n_2. [6]. If we select the unmatched incident node that connects n_1 with a maximal edge-weight, then either n_1 or the selected incident node is multinode. This means that the combining the two nodes will result in a node with a large set of neighbours, and this obviously means a lower matching ratio.

We note that HEM has a complexity of $O(E)$, which is equal to RM. This does not mean that the HEM and RM is equal in terms of execution time. When a node is picked by random, then RM matches only the first unmatched incident node of the node it just picked, while HEM must search through all its incident nodes in order to determine which one have the heaviest edge weight.

3.2.3 Heavy Edge Matching Minimization

In some applications, nodes and edge-weights are initially uniform. During the first coarsening level, it is likely that when a node is selected, it will have several incident nodes that connects it with a maximal edge-weight. The problem however, how do we select among these?

Heavy edge matching minimization (HEMM) is our suggestion to solve this problem, by defining a rule for selecting among incident nodes with maximal edge weight.

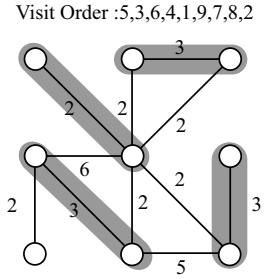


Figure 18: Heavy edge matching minimization

The heuristic is identical to HEM, except when a node have more than one unmatched incident node that is connected with a maximal edge-weight. Then we select one of these incident nodes that have the lowest number of neighbours.

[6]Except for a special case where the graph consists of two nodes, where they are connected by one edge

The motivation behind the heuristic is to increase the matching ratio for HEM. From [4] it known that using HEM decreases the matching ratio, compared to other heuristics. HEMM tries to increase the matching ratio, while persevering the properties of HEM.

The complexity of the method is $O(E)$. In practice, there is no difference between the execution time of HEM and HEMM, since the number of incident nodes are placed inside a variable.

3.2.4 Gain Vertex Matching

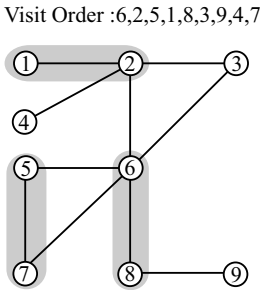


Figure 19: Gain vertex matching

Gain vertex matching (GVM) was proposed in relation to graph partition in [4]. Unlike HEM, GVM minimizes degree weight rather than the edge-weight.

The heuristic works in the following way, first a node n_1 is selected by random, then the unmatched incident node n_2 is selected such that the degree of the multinode $n_{(1,2)}$ is minimized. The complexity of GVM is $O(N \times E)$, but by precalculating the degree of each node it can be quite fast. An interesting digestion, HEMMIN performs identical to GVM during the first coarsening iteration.

3.2.5 Local Greedy Heavy Edge Matching

Inspired by [27], the idea behind local greedy heavy edge matching (LGHEM) is the following: In HEM we select a node n_1 by random, then the unmatched incident node n_2 is selected such that it is connected to n_1 with the maximal edge weight. The edge between n_1 and n_2 is the heaviest unmatched edge that is incident to n_1, but it might not be the heaviest edge of n_2 as seen in Figure 20.

From this observation, it is possible to develop a method that search locally in the node neighbourhood finding two unmatched nodes which are incident to each other with the mutually same maximal edge weight. First we define n_t as a temporary node. Now we select by random, an unmatched

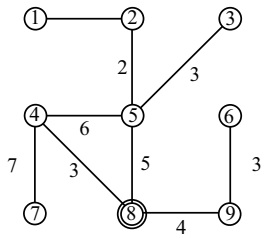


Figure 20: We see that the unmatched and incident to node 8 with the highest edge-weight is node 5. However, node 5 is incident to node 4 with an even higher edge-weight than the one between node 5 and node 8.

node n_1, and set $n_t = n_1$. As for HEM, we select the incident node n_2 that connects n_t with the maximal edge-weight. Let this edge-weight be equal to $w_{(n_t,n_2)}$. We do the same procedure for n_2, we find its incident node n_3 that connect n_2 with the heaviest edge-weight, and store this inside $w_{(n_2,n_3)}$. If $w_{(n_2,n_3)} = w_{(n_t,n_2)}$ then we add the edge with n_2 and n_3 as endpoints to a matching. If not, then $n_t = n_2$ and we continue the search until we find to edges with the mutually maximum edge weight.

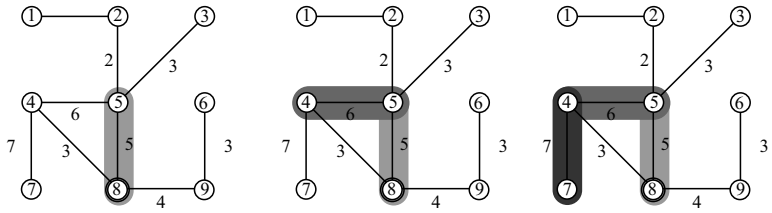


Figure 21: We start by picking node 8. We set n_t to node 8. Then we find node 8s incident unmatched node that connects node 8 to the maximal edge-weight, which is node 5. Is node 8 the incident unmatched node that connects node 5 to the maximal weight? No, the edge-weight between node 4 and node 5 is heavier than the edge-weight between node 8 and 5. Then we set n_t to node 5 and continue to search until we find two nodes that are incident to each other with the mutually the same maximal edge-weight. As seen from the figure this is node 4 and node 7

Will there always be two unmatched nodes with the mutually maximal edge-weight? Yes indeed! No matter what node that are selected, it have either a unmatched incident node that connects to the selected node with a maximal weight, or have no unmatched incident nodes at all. The incident node can either have an incident node connected to it, with the same maximal weight as the first node, or it can have one connected with a higher weight. Since there only are a finite number of nodes and edges, we will

sooner or later run into a node that have the same maximal weight, or a node that only have one edge (in this case, we know that this is the maximal weight).

LGHEM is more complicated than HEM, and have a much higher complexity, $O(E^2)$.

3.2.6 Global Greedy Heavy Edge Matching

My master thesis supervisor [4] suggested that LGHEM, is almost the same as searching thought the edges for the heaviest edge, than search through the nodes. We name this method Global Greedy Heavy Edge Matching (GGHEM) which is sort of an extension to LGHEM. In LGHEM, we select the nodes in a greedy manner from a random selected node. In GGHEM, we do not select randomly, we simply

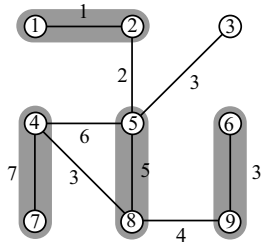


Figure 22: GGHEM matching heuristic

sort the edges descending based on their weight. Let E_s be a list containing all the edges sorted by their edge-weight. this edge to a matching.

The method works the following way, we select an edge from E_s, this edge can either be a matched or unmatched. If it is matched, we simply skip to the next edge in E_s. If it isn't a matched, the we first record the endpoints n_1 and n_2, and include it to the matchings. Now we need to update edge every edge that has one either n_1 or n_2 as a endpoint, and set them to unmatched. We continue until we reach the end of E_s. Since we need to sort edges, the complexity of this heuristic is $O((E) \times log_2(E) \times (E))$.

3.2.7 Implementation of GGHEM

The multilevel graph partitioning algorithm that we use is based around the nodes rather than edges. More specific, the datastructure is based on CRS. From Figure 23 it is clear that inserting GGHEM into the CRS datastructure where the edges are implicit given can be a bit of a challenge. Since the edges are unnumbered and stored twice, it isn't particular easy to sort efficiently.

To cope with this, we create a list where we store each edge, or more specific we store the two endpoints and the edge-weight. This list must be

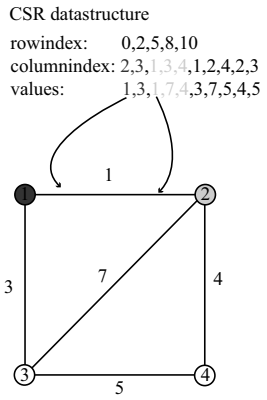


Figure 23: The example of the CSR structure where the edges are implicit given

created from the CRS datastructure for each coarsening level.

We define a list *columnused*, where initially all is set to *available*. We start by the two leftmost numbers in *rowindex* (i.e. 0 and 2). As seen in Figure 23 the size of the set of neighbours of node 1 is $2 - 0 = 2$. We go throught the incident nodes of node 1 in columnindex, if they are set to *available* in *columnused* we create a new edge as seen in Figure 24

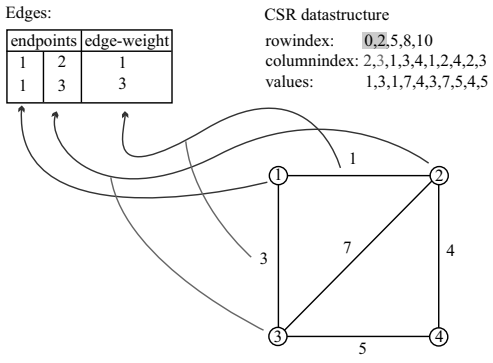


Figure 24: Creating edge based structure from the CRS structure

When the list of edges is completed, the list is sorted with the introsort algorithm [23], so edge edges with a maximal weight is in the top of the list, while edges with a minimum weight is at the bottom of the list. Since the algorithm are to begin with are node based, stores a matching as seen in Figure 25.

We are now using edge based datastructure, but the reconstruction of the graph is still based on matchingvector. That means that we need to

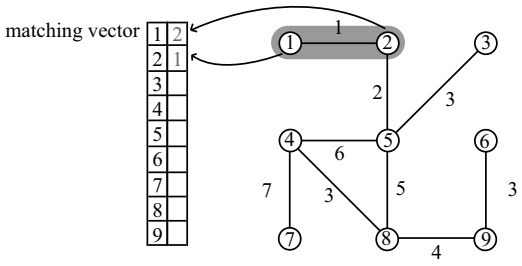


Figure 25: matching vector

update the matchingvector, when selecting from the edges list.

We start by selecting an edge from the top of the list. There can be three different cases, either one of endpoints in the edge is in matchingvector, one of the endpoints is in the matchingvector or both of the endpoints are in the matchingvector.

In the first case, we insert each of the endpoints in the matchingvector, $matchingvector[endpoint1] = endpoint2$ and $matchingvector[endpoint2] = endpoint1$.

The second case, we need to find out if the other unmatched endpoint are without unmatched incident nodes. If it has no unmatched incident nodes, then $matchingevector[unmatchedendpoint]$ = unmatched endpoint.

For the third case, we simply skip to the next edge. The algorithm terminates once we reach the end of the list, or we have processed all the edges.

3.3 Initial partitioning and refinement

For initial partitioning MLRB is used, while for refinement GR is used. We note that there also is used a variant of GR, to improve the load imbalance (with the cost of increasing edge-cut).

4 Experiments

For testing the matching heuristics, we have inserted LGHEM and GGHEM into the well known graph partitioning software Metis 4.0 [20]. For initial partitioning, we use MLRB from the same software, and for refinement we use greedy refinement heuristic.

In the following sections we examine what effect these modifications had on edge-cut, load imbalance and execution time.

Finally we compare our partitioning results with the graph partitioning archive [29], where we discover two partitions with the smallest known edge-cut by using LGHEM and GGHEM as the matching heuristic.

4.1 Measuring execution time

One way to measure time on a computer is to count them number of clock cycles since the program started. For instance, assume that we want to measure how much time a certain method call takes. Then we first measure the amount of clock-cycles used before the method is called, and the amount of clock-cycles used after the method is called. We subtract the two measurements (i.e. stop - start), and divide them with frequency of the microprocessor to get the time in milliseconds. We found that it wasn't always as accurate as we wanted (a milliseconds represent an eternity for a modern computer), so we did the following; We measured it 8 times, and presented the average time.

4.2 Datasets

For performing the experiment, we have selected a serie of 34 graphs (Table 1) from the graph partitioning archive [29].

4.3 Comparing HEM with LGHEM

In this section we compare what consequences changing HEM with LGHEM will have on the final edge-cut and the load imbalance. Since LGHEM is more complex than HEM, we also measure what effect the change had on the execution time.

4.3.1 Comparing the final edge-cut when using the HEM matching heuristic and LGHEM matching heuristic

In Figure 26 we see the edge-cuts of the 16-way, 32-way and 64-way partitions of the graphs in Table 1. The edge-cuts are presented relative to the edge-cuts we would get if we used HEM to produce the coarser graph. For instance if one of the histogrambars in Figure 26 are below the baseline, then using LGHEM creates a lower edge-cut than using HEM.

graph name	No. of Nodes	No. of Edges
add20	2395	7462
data	2851	15093
3elt	4720	13722
uk	4824	6837
add32	4960	9462
bcsstk33	8738	291583
whitaker3	9800	28989
crack	10240	30380
wing_nodal	10937	75488
fe_4elt2	11143	32818
vibrobox	12328	165250
bcsstk29	13992	302748
4elt	15606	45878
fe_sphere	16386	49152
cti	16840	48232
memplus	17758	54196
cs4	22499	43858
bcsstk30	28924	1007284
bcsstk31	35588	572914
fe_pwt	36519	144794
bcsstk32	44609	985046
fe_body	45087	163734
t60k	60005	89440
wing	62032	121544
brack2	62631	366559
finan512	74752	261120
fe_tooth	78136	452591
fe_rotor	99617	662431
598a	110971	741934
fe_ocean	143437	409593
144	144649	1074393
wave	156317	1059331
m14b	214765	1679018
auto	448695	3314611

Table 1: Graphs used in the experiments

We found that using LGHEM creates lower edge-cut for 61% of the cases. We have also found that if LGHEM is used, and the edge-cut is lower, then it is on average 6% lower. If using HEM creates a lower edge-cut, then it's on average 2% lower.

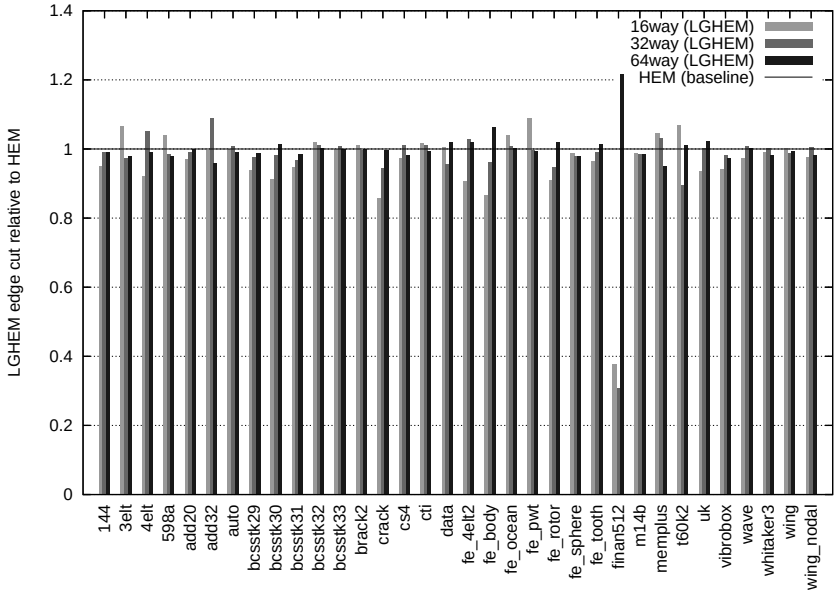


Figure 26: The edge-cuts of the 16-way,32-way and 64-way partitions of the graphs in Table 1

From Figure 26 we see that the 16-way and 32-way partitions of *finan512* both has a more than 70% lower edge-cut. We also see that the 64-way partition of the same graph has a 20% higher edge-cut.

We have examined this case closer, by creating 2-way,4-way,8-way,16-way, 32-way,64-way,128-way and 256-way partitions of *finan512*. Since it turns out that only the 16-way and 32-way partitions have a 70% lower edge-cut(i.e. there isn't a property of the graph that makes LGHEM superior to HEM), we examine these two cases closer. We also examine the 64-way partition, where the edge-cut was 20% higher.

We have two ideas to what causes the low edge-cut: Either we are unable to refine the coarser graph produced by HEM, or there are some properties of the coarser graph such that we are unable to make an initial partition with a low edge-cut.

We could imagine that the GR heuristic was unable to move any node that would decrease the edge-cut without violating the balance condition.

In Figure 28 we see the edge-cut of the initial partition. We see that for *finan512* the difference between the edge-cut when the coarser graph is produced by LGHEM and HEM, is larger than in Figure 26.

We have studied the 16-way and 32-way partitions of *finan512* closer. In Table 2 we see that the coarser graph produced by HEM gets more refinend than the one produced by LGHEM. Then it is clear that the reason for the big difference between the edge-cuts when using LGHEM and HEM isn't

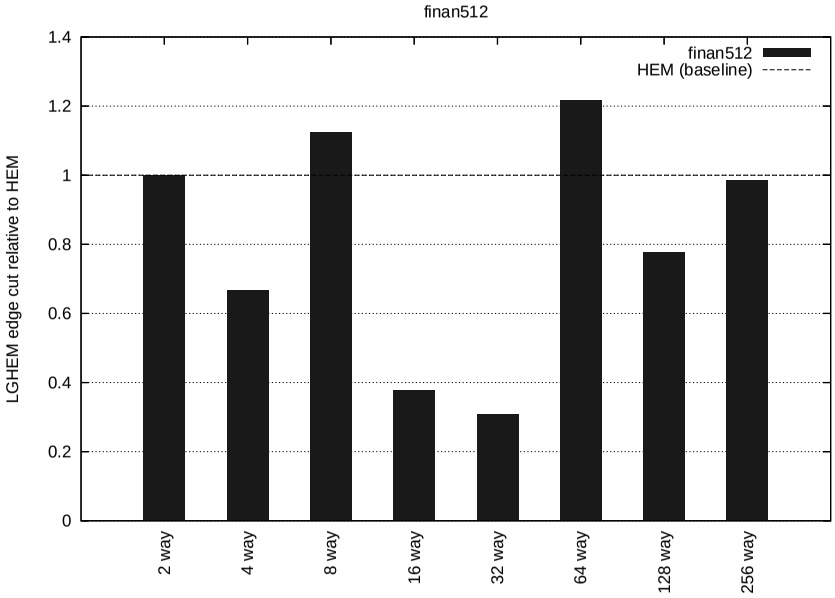


Figure 27: The edge-cut of the 2,4,8,16,32,64,128 and 256 way partitions where LGHEM is used as the matching heuristic, relative to the edge-cut when HEM is used as the matching heuristic.

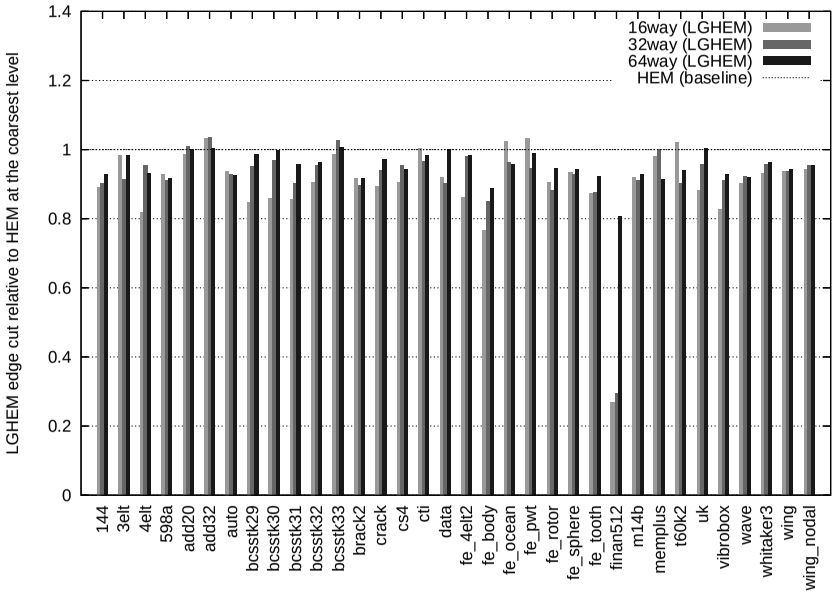


Figure 28: The initial edge-cut for 16,32 and 64 partitions, relative to the initial edge-cut when HEM is used as the matching heuristic.

graph name	matching heuristic	k	iEC	EC	edge-cut decrease
finan512	HEM	16	20832	3432	83%
finan512	LGHEM	16	5599	1296	76%
finan512	HEM	32	33788	8401	76%
finan512	LGHEM	32	9969	2592	74%

Table 2: 16-way partitions for *finan512*. iEC denotes the initial edge-cut, and EC is the edge-cut after refinement. We also measure the decrease in edge-cut between the initial partition and the final partition

caused by lack of refinement.

We found that one could explain the difference between LGHEM and HEM by looking at the coarser graph for the 16-way and 32-way partitions of *finan512*. In 3 we look at the number of edges ($|E|$) and the sum of weight of the edges ($\sum w(e) \in E_i$).

graph name	matching heuristic	k	$\|E_i\|$	$\sum w(e) \in E_i$
finan512	HEM	16	20594	197474
finan512	HEM	32	21404	200402
finan512	LGHEM	16	7786	147068
finan512	LGHEM	32	10734	162000

Table 3: 16-way partitions for *finan512*

It turns out that if LGHEM is used, then the coarser graphs have (on average) 50% less edges and that if we sum the edge-weights, it is 26% less than if the coarser graphs was produced by HEM. This does explain why the edge-cuts of *finan512* is smaller for initial partitions. Since there coarser graphs have smaller edge-weights and and less edges(Table 3). It is also likely that it is easier to find a partition with a lower edge-cut. This explains why the edge-cut is smaller for the final partition, since we see in Table 2 that the partitions got the same amount of refinement.

But it does not explain why it is 70% smaller. The coarser graph produced by HEM has more edges and nodes than LGHEM. That means that it is not straight forward to compare the graphs. To try to compare how uniform the edge-weights are distributed for the coarser graphs, we used a coefficient of variation [31]. That is, we divided the standard deviation of the edge-weight by the average edge-weight and found out that this value is somewhat higher for the coarser graph produced by HEM than LGHEM. That means, that the edge-weight is less uniformly distributed for the coarser graph produced by HEM.

Then it is probably easier for the MLRB to create a partition with a low edge-cut when the coarser graph is produced by LGHEM since it has

fewer edges and the weight of these edges are more uniformly distributed. It could be that if the edge-weights are non-uniform, this could be utilized to create a smaller edge-cut, for instance by making the nodes that has edges with a low edge-weight as boundary nodes. However, this would require our bisection heuristic to be more advanced than greedy partitioning.

We have seen that the 16-way and 32-way partitions of *finan512* have a 70% lower edge-cut when LGHEM produces the coarser graph. But why did the 64-way partitioning create a 20% higher LGHEM for the same graph? Below we investigate the 64-way partitioning of *finan512* and try to explain how it differs from the 16-way and 32-way cases.

The initial edge-cut is smaller for LGHEM than for HEM. However after refining, this changes, and the edge-cuts get 18% lower for HEM.

This means that the reduction comes from the refinement. When HEM is used then the edge-cut is decreased by more than 73% while when LGHEM is used then the edge-cut is decreases by more than 59%. Even though the edge-cut initially is 19% lower (see Figure 28) for the coarser graph produced by LGHEM , the decrease of edge-cut during refinement is so much higher when the coarser graph is produced by HEM, so we end up with an edge-cut that is 18% lower.

We found that the reason why we were able to refine this much more was because the coarsest graph HEM produced had on average 39% more incident nodes. At the same time the average edge- weight was only 2% higher and the sum of edges where 10% higher for the coarsest graph compared to LGHEM.

It is reasonable that the edge-cut initially got higher when HEM produced the coarser graph. however since the number of incident nodes are much higher it gives us great flexibility during refinement. It is quite likely that this is the reason why the 64-way partition of *finan512* created a lower edge-cut for the 64-way partition when HEM was used.

From this we can answer the question why the edge-cut was so much lower for the 16-way and 32-way partitions. When examining the number of edges in the coarser graph, we see from Table 4 that the number of edges in the coarser graph produced by HEM is much higher for 16-way and 32-way partitions (over 50%) by only 10% higher for the 64-way partition.

graph name	matching heuristic	16-way	32-way	64-way
finan512	HEM	20594	21404	22442
finan512	LGHEM	7786	10734	20264

Table 4: The number of edges for the coarsest graph where LGHEM and HEM have been used as a matching heuristic.

We know from the previous chapter, that the number of nodes in the coarser graph are determined by the coarsening threshold. We have found

that the number of nodes is over 50% higher for the coarser graph for the 32-way partition than for the 64-way partition. Recall from Equation 6 in Chapter 3 that this is impossible. This means that during two coarsening levels, the size of the graph was reduced by less than 10%. This can be seen by the number of coarsening levels in Table 5

graph name	matching heuristic	16-way	32-way	64-way
finan512	HEM	8	8	15
finan512	LGHEM	15	13	14

Table 5: The number coarsening levels were LGHEM and HEM have been used as a matching heuristic.

Why we were unable to reduce the graph with 10% for the 16-way and 32-way partitions, when we could do that for 64-way partitions? We found that the answer lies in the matching constraint (see Equation 7). For the 16-way and 32-way partitions we are allowed to make multinodes with higher weight than for the 64-way partition. Combining nodes with high weight created nodes with a larger number of neighbours, which again decreased the matching ratio.

The result is that we are unable to reduce the size of the graph is the following; we end up with a graph where the sum of edges is higher than if use the LGHEM matching heuristic. This again results in a higher initial edge-cut. For the 64-way partition where HEM was used, the edge-cut was initially higher, but it had a larger number of incident nodes and one more coarsening level so it was able to refine more efficiently then LGHEM.

However, since we were unable to use more than 8 coarsening levels, the number of incident nodes really wasn't really much larger than if we used LGHEM. We noticed that we were able to refine the graph where HEM was used some more than for LGHEM, however even though it was able to refine the graph more efficiently, it only had 8 uncoarsening levels, which also meant less refinement, than if LGHEM was used.

To prove that this was the cause of the problem, we did the following. For *finan512* we allowed the reduction to be less than 10% for the 16-way partition. Then HEM doesn't escape the coarsening loop before it reaches the coarsening threshold. This took 37 coarsening levels, but we ended up with a result that had a lower edge-cut than if we used LGHEM (similarly to the 64-way partition).

4.3.2 Comparing the load imbalance when LGHEM and HEM is used as matching heuristics

From recursive bisection, we know that if the node-weight is uniform, then the difference between two partitions will be at most one node. Since we are

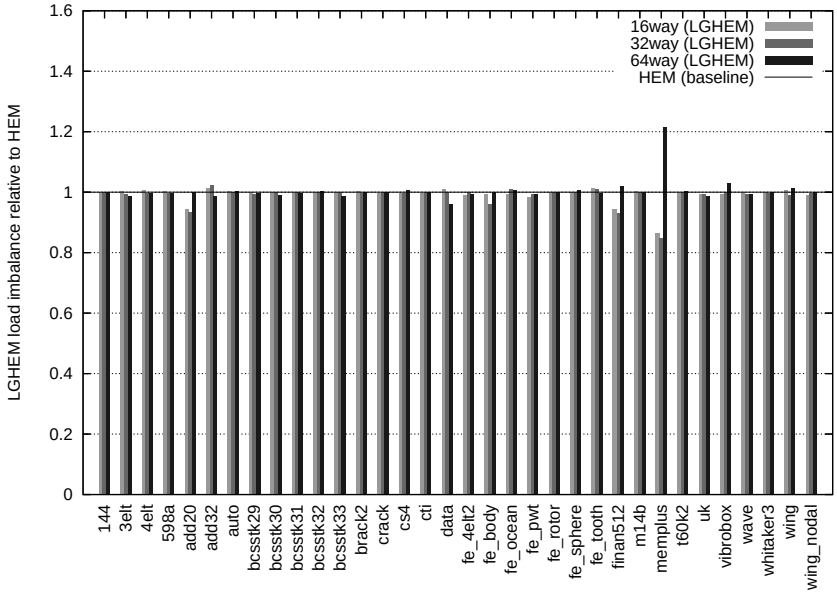


Figure 29: The 16-way 32-way and 64 way load imbalance relative to cases HEM

using matching constraint to make sure than node-weights will be (almost) uniform, we expect the graph to have a low imbalance regardless of what matching heuristic we use.

From Figure 29 we see that the load imbalance of the final graph is almost the same when using LGHEM or HEM. We measure the load imbalance relative to HEM as above, which means that if the bars are below the baseline then LGHEM creates a lower load imbalance than HEM.

We have measured the load imbalance, and found out that for 70% of the cases using LGHEM creates a lower load imbalance. If using LGHEM gives a smaller load imbalance than HEM, it is on average 2% smaller. If using HEM gives a smaller load imbalance than LGHEM it is on average 1% smaller. Since the difference between load imbalances are between 1% and 2%, we could argue that the difference in load imbalance is minimal. However, we could argue that partitions where LGHEM have been used as the matching heuristic tends to have slightly smaller load imbalance.

In Figure 29 the load imbalance for the 16-way partitioning of *memplus* is 18% lower when LGHEM is used. For the same dataset, but the 64-way partitioning, the load-imbalance is 23% lower when HEM is used.

We know that some initial load imbalance can be helpful during the refinement, in order to be more flexible in moving nodes to escape local optimas. For this experiment we allow a load imbalance up to 5%. However, if it is over 5% we must take steps to try refining the load imbalance with

the expense of an possible increase in edge-cut.

graph name	matching heuristic	iEC	maxweight of initial partition
memplus	LGHEM	16639	1186
memplus	HEM	16982	1126

Table 6: The initial edge-cut and the initial maximal weighted partition for the 16-way partition of *memplus* where LGHEM and HEM has been used as the matching heuristic.

From Table 6 we see that 16-way initial partitioning of *memplus*. Since both using LGHEM and HEM yields the same number of coarsening levels (8) and since the difference between the number of nodes between the two coarser graph is less than 5% we think that it is fair to compare the load imbalance of the two initial partitions. We see that initially LGHEM produced partitions that has a lower edge-cut and a higher load imbalance. We found out that for both the datasets, we needed to improve the balance on the expense of edge-cut, however for the partitioning that were produced by LGHEM we needed to do it for five times, instead of four as for HEM.

graph name	matching heuristic	EC	max partition
memplus	LGHEM	15845	1458
memplus	HEM	15149	1685

Table 7: The final edge-cut and the maximal weighted partition of *memplus* where LGHEM and HEM has been used as the matching heuristic.

From Table 7 we see the result. The partition where LGHEM was used has 14% lower load imbalance. We see also that the extra improvement of balance had a price, since the edge-cut increased for LGHEM. For the 64-way partition we see that partitions where HEM was used creates a load imbalance that is 18% lower than if LGHEM is used. Unlike the case above, we called the method the same number of times to try to balance it.

Unlike the case for 16-way partition, equal amount of iterations are spent to decrease the load imbalance with the expence of edge-cut. From Table we see that the initial load imbalance is lower for LGHEM than for HEM.

However, we see that the edge-cut for the final partition, LGHEM has now a much higher load imbalance than for HEM.

Normally we measure the load imbalance as the relation between the maximal weighted partition and the ideal weighted partition $|N|/k$ (Equation 1 in chapter 1). However, in this case this did not give us any clues to why the LGHEM heuristic had a much higher load imbalance than for the final partition. Neither did standard deviation or comparing the minium weighted with the maximum weighted partition.

graph name	matching heuristic	iEC	max initial partition
memplus	LGHEM	21091	300
memplus	HEM	23105	317

Table 8: The initial edge-cut and the maximal weighted partition for the 64-way partitioning of *memplus* where LGHEM and HEM has been used as the matching heuristic.

graph name	matching heuristic	EC	max partition
memplus	LGHEM	19849	485
memplus	HEM	20894	399

Table 9: The final edge-cut and the maximal weighted partition of *memplus* where LGHEM and HEM have been used as the matching heuristics.

In Figure 30 we have done the following; First we have gathered weights of the initial partitions, then we have sorted them from the smallest to the largest. Finally we plotted them as points. A lot of "straight" lines indicates that the partition is well balanced.

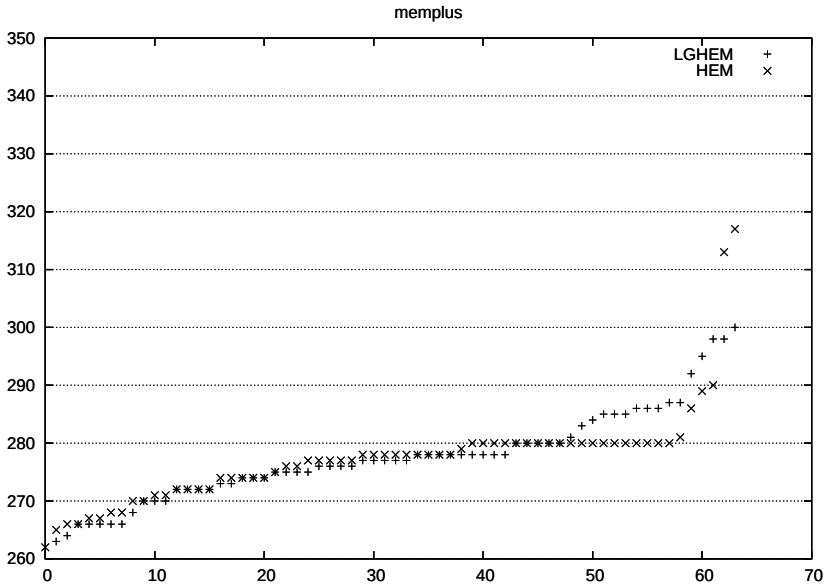


Figure 30: The weight of the partitions where LGEM and HEM have been used

From $x = 1 \cdots 40$ we see that the points almost overlap each other. However from $x = 48 \cdots 64$ we can see weakly that HEM is somewhat better

balanced than LGHEM. We found that this could explain why the load imbalance is 20% lower for HEM for the 64 partition.

4.3.3 Comparing the execution time of HEM and LGHEM

When selecting a node HEM must obviously search through all its unmatched incident nodes to find the one that is incident with the maximal edge-weight. LGHEM is an extension of HEM, and must do the same, but in addition, when the incident node with the heaviest edge is found, we must search through all of the incident nodes in order to determine if the nodes are mutually connected to the same maximal edge-weight. In addition, if the incident node have incident nodes than that connects it with a higher edge-weight, we must repeat the procedure again. This is repeated until two nodes are found, that connects with the mutually same edge-weight. We therefore expect that HEM searches through fewer edges than LGHEM during a coarsening level.

In Figure 31, we have measured the total time for the partitioning. The results are given relative to the results for HEM, which means that if the bar is below the baseline, then LGHEM is faster than HEM. We have selected graph that have a large number of nodes, to decrease the inaccuracy of the time measurement as much as possible.

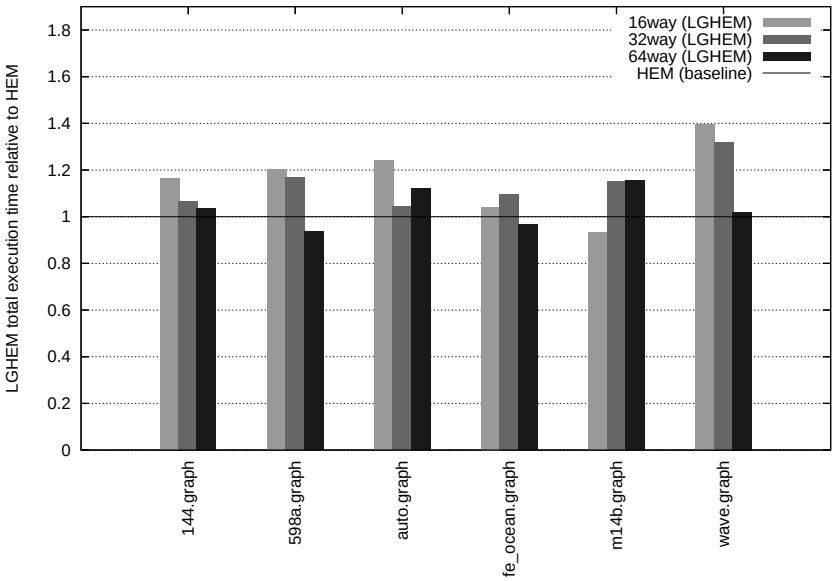


Figure 31: the total execution time for 16-way, 32-way and 64 way LGHEM partitions relative to HEM partitions for a selected subset of the datasets in Table 1 where the number of nodes ≥ 100000

From figure 31 we see that using LGHEM decreases the execution time for 20% of the cases. Since LGHEM is a more complex heuristic than HEM, it is not completely clear how LGHEM can be faster than HEM at all.

In Figure 32, 35 and 36 we have measured the execution time for the three phases (coarsening, initial partitioning and uncoarsening). In Figure 32 we see the execution time used for the coarsening phase.

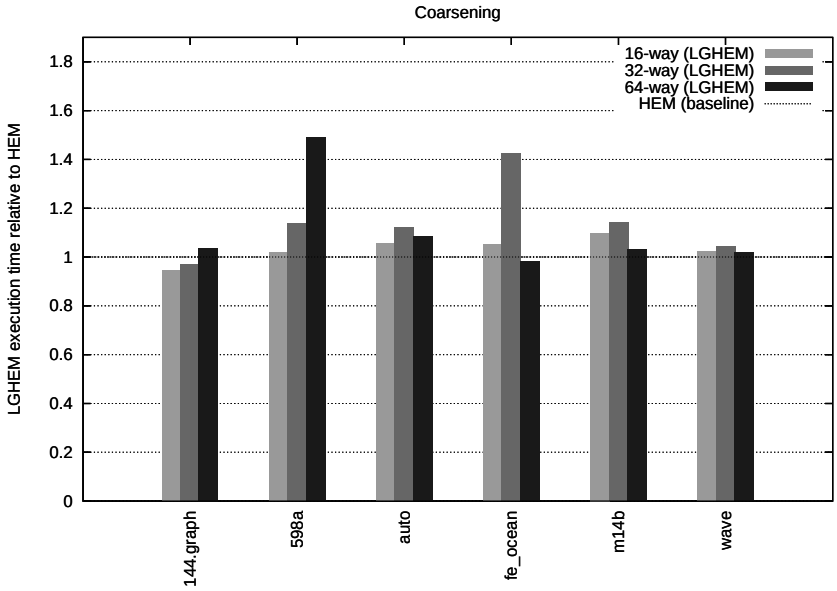


Figure 32: the execution time for the coarsening phase

We see that using LGHEM during the coarsening phase, makes the coarsening a bit slower than using HEM. However, LGHEM is a more complex method than HEM so we would expect the difference to be larger. When we consider a node and its incident nodes in LGHEM, we consider often twice as many as we would for HEM. However, on average the coarsening phase is only 15% faster when HEM is used as the matching heuristic.

We have found that LGHEM requires less coarsening levels to reach the coarsening threshold. This can be either because we are unable to reduce the graph by less than 10% between two coarsening level, or that matching ratio is higher. We have found that the number of nodes in the coarser graph is consisently lower when LGHEM is used, which indicates that it has a higher matching ratio.

Why does LGHEM have a higher matching ratio than HEM? From [4] we see that RM has a higher matching ratio than HEM. We know from [17] that HEM creates a coarser graph with lower edge-weight than RM. From edge-cut analysis in Section 4.3.1 we know that coarsest graph where LGHEM is used have a lower edge-weight than HEM. However, yet the

matching ratio is higher for LGHEM than for HEM. We would expect it to be the other way, since it created a coarser graph with a lower edge-weight, we would expect that this was parallel to RM and HEM, so HEM had a higher matching ratio than LGHEM.

However we have found that LGHEM on average matches edges where the endpoints have smaller set of neighbours than HEM. This is logical, because if we search through the graph for two nodes with mutually maximal edge-weight, it is likely that we will end up with an incident node with a lower set of neighbours(from the reasoning about LGHEM in the previous chapter). This again makes a multinode with a smaller set of neighbours, which again does not block future matchings. This gives the coarser graph produced by LGHEM a smaller and smaller degree and a smaller set of neighbours. This allow LGHEM to search through fewer edges in order to find two mutually maximal weighted edges.

In Figure 33 we have done the following for *wave*: Each time we consider an edge we increase a counter by 1. We divide this number by the number of edges in the graph.

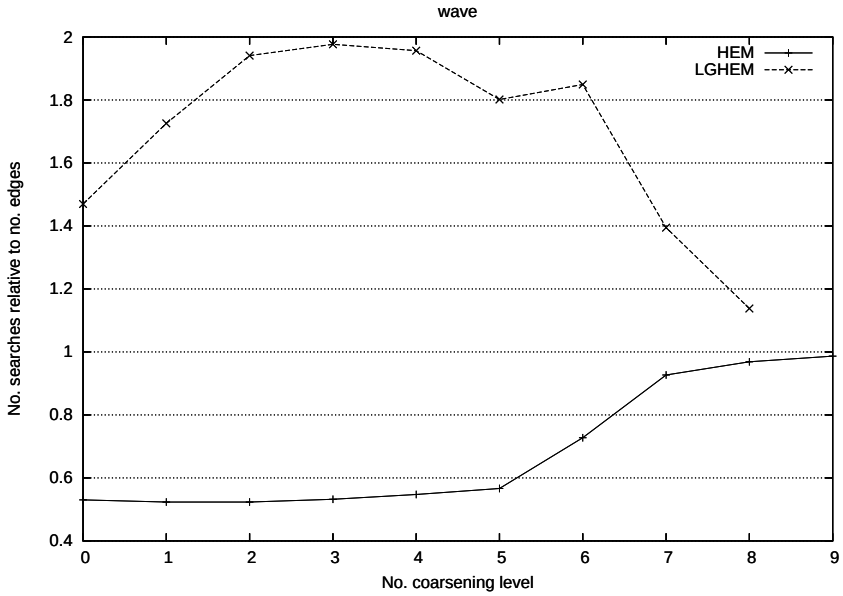


Figure 33: The number of edges searched per coarsening iteration for *wave*

We see from Figure 33 that to begin with LGHEM consider much more edges than HEM. However at the same time we see from Figure 34 that LGHEM has a higher matching ratio. After the coarsening level 4, the average set of neighbour have decreased significantly. The result can be seen in Figure 33, where the number of searches start to decrease.

For the final coarsening levels, we see that the number of searches in-

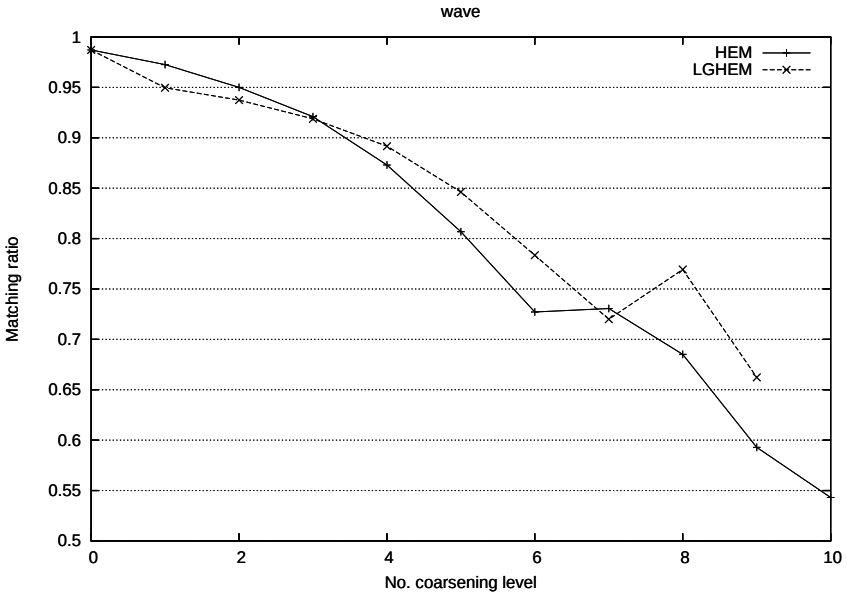


Figure 34: The matching ratio for *wave*

creases for HEM. Infact, we see that it almost reaches the upper number of searches it can possible do ($|E|$). At the same time the number of searches decrease for LGHEM, almost reaching the upper number of searches of HEM.

For the initial partition, we see that partitioning the coarser graphs produced by LGHEM generally are faster than if they are produced by HEM. [7]. This is reasonable, since we have seen that LGHEM produces smaller graphs (in terms of edges and nodes) than HEM.

For the uncoarsening phase and refinement, we see from Figure 36 that if LGHEM have been used for coarsening, then less time is spent refining the graph. This can also be seen in the analysis from the edge-cut in Section 4.3.1, where we see that coarser graph where HEM has been used, tends to have a larger set of neighbours. This increases the ability to refine the partition (i.e. makes it possible to perform more moves).

By examining the three phases, we see that it is not unatural that the difference between the two heuristics are so low.

4.4 Comparing LGHEM to GGHEM

In this section we compare the difference between the edge-cut and load imbalance when using GGHEM and LGHEM. Since our implementation of GGHEM requires three different stages (datastructure building, sorting

[7]Since the initial partitioning is very fast, there could be a result of inaccuratcy of the measurements

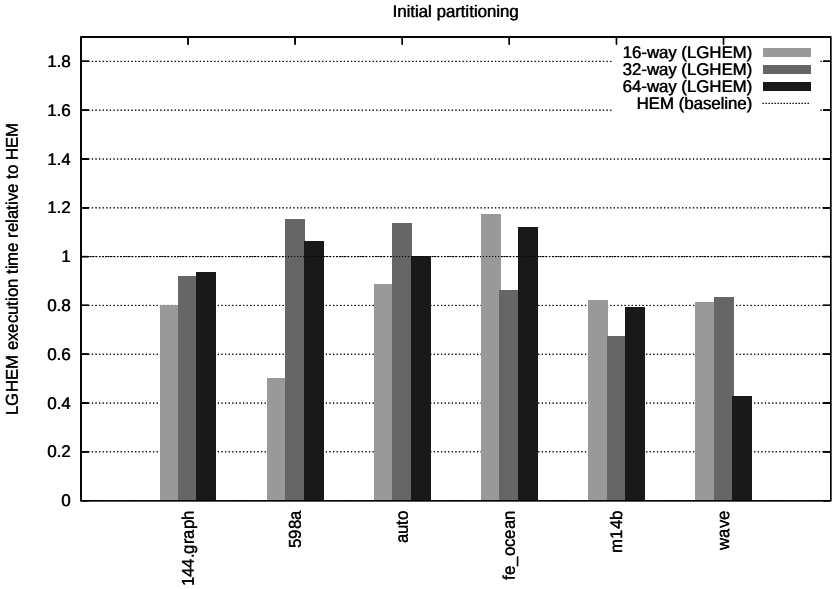


Figure 35: The execution time for the initial partition for 16-, 32- and 64-way partitioning

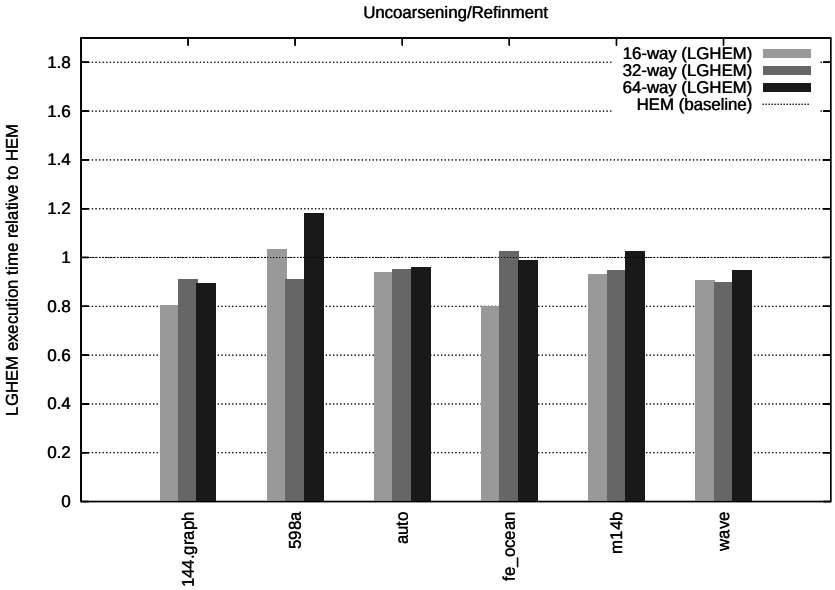


Figure 36: The execution time for the uncoarsening and refinement for 16-, 32- and 64-way partitioning

and matching) we also measure the difference in execution time for the two methods.

4.4.1 Comparing the final edge-cut when using the LGHEM matching heuristic and the GGHEM matching heuristic

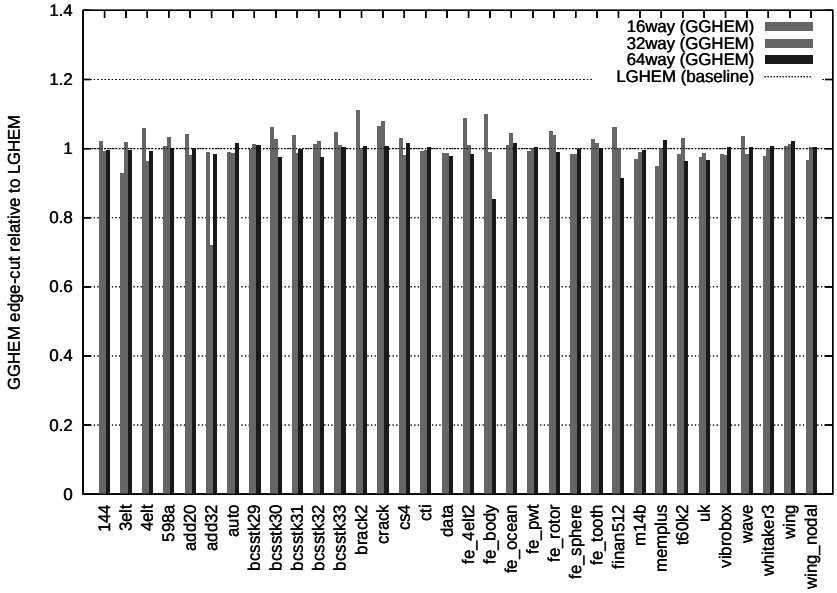


Figure 37: the edge-cut for 16, 32 and 64-way GGHEM partitions relative to LGHEM partitions for the datasets in table 1.

We have measured from Figure 37 and found out that if using GGHEM for coarsening, it creates the lowest edge-cut for exactly 51% of the cases. If GGHEM is used creates the lowest edge-cut, then it is on average 3% lower. If LGHEM is used, and creates the lowest edge-cut then it is on average 2% lower.

This indicates that the two heuristics almost gives identical result. The overall objective of LGHEM and GGHEM is to decrease the edge-weight of the coarser graph such that it will be easier to find a partition with a low edge-cut [17]. We have found that LGHEM consistently produces coarser graphs where the sum of the edge-weights are lower(however, on average they are no more than 2% lower). This is unexpected. What makes it even stranger, is that the initial edge-cut, using GGHEM creates a lower edge-cut for 53% of the cases. However, we would expect LGHEM would create the lowest initial edge-cut for almost all test cases. Then perhaps the coarser graph produced by GGHEM had a property such that it was easy to refine it.

We have found that the reason why there are fewer edges for LGHEM than for GGHEM, is that GGHEM has a much higher matching ratio than LGHEM. By measuring the coarsening levels, we found that GGHEM usually uses 1 coarsening level less than LGHEM to reach the coarsening threshold. This means that the coarser graph where GGHEM is used have almost twice as many edges. As we have seen above, the edge-weight is higher for GGHEM, which is logical since we use less coarsening levels. But we also note that it is on average only 2% higher.

However, we know that the edge-weight is consistenly lower when LGHEM is used. We also know that the set of neighbours is smaller for the coarser graph produced by GGHEM than for LGHEM. From the experiments with HEM and LGHEM heuristic, we saw that the larger set of incident nodes and the larger number of uncoarsening levels where important for HEM in order to decrease the edge-cut. GGHEM has none of these properties, it has a higher sum of edge-weights, a smaller number of incident nodes and fewer coarsening levels than LGHEM. In addition we known the the initial edge-cut for GGHEM is lower than the final edge-cut. From this we can conclude that there is a difference between a large number of edges with a low edge-weight and a small number of edges with a higher edge-weight.

4.4.2 Comparing the load imbalance when LGHEM and GGHEM is used as the matching heuristic

In Figure 38 we see the difference in load imbalance between LGHEM and GGHEM.

4.4.3 Comparing the execution time of LGHEM and GGHEM

GGHEM consists of three parts; building the edge datastructure, sorting the edges according to their weight and creating a maximal matching. The complexity of this operation is $O(E) + O(E \times log(E)) + O(E)$. However complexity analysis says more about the upper (and often unlikely) bounds.

We would expect that the edge building part is slower than the matching part, even though they have the same complexity. This because, we have to consider all the edges in order to build a datastructure, but we might get away with considering less edges in the matching part, since previous matchings block future matchings.

We expect tthe sorting part to be slower than building the datastructure and creating a maximal matching, since we atleast must consider all the edges in order to sort them(how else would we know if they were sorted).

We have measured the time for the three parts, and found out that the time is divided among the three parts the following way: Building datastructure = 35%, sorting the edges = 48% and creating a maximal matching = 17%. We also see that this is reasonable. To build a datastructure we

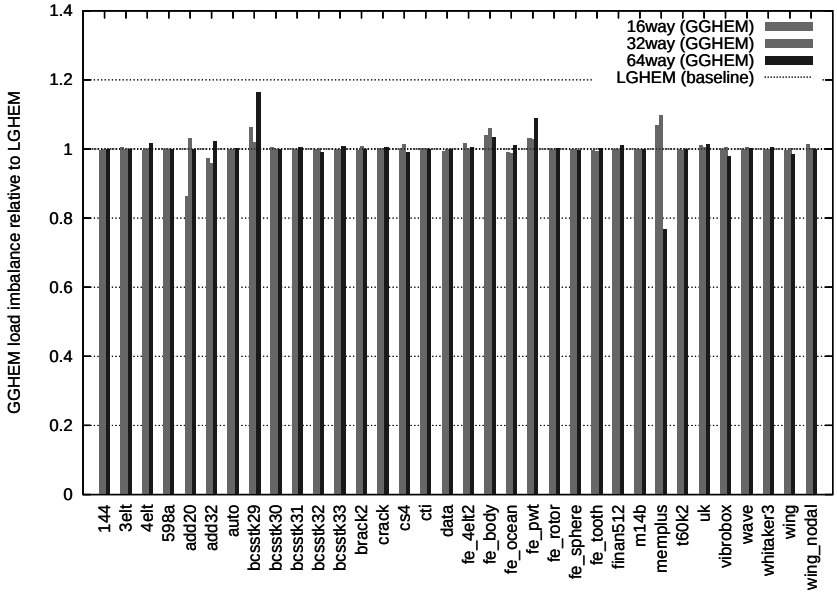


Figure 38: the 16-way, 32-way and 64-way load imbalance of partitions where GGHEM has been used relative to cases LGHEM have been used as the matching heuristic for the datasets in Table 1

must consider every edge . On the other hand, to create a maximal matching, we only have to count the endpoints processed. If all the endpoints are processed, we know that there is no point searching further for any edges to be included in the matching. Since there are likely that there are more edges than nodes (i.e endpoints) then it is likely that it we will find a maximal matching before we have considered all the edges. It is also reasonable that the sorting is the part that requires the most time, since we must consider more than all the edges. Then it is logical since the sorting is slower than the datastructure building and the maximal matching.

We see from Figure 39 that overal execution time is higher when GGHEM is used. LGHEM has an unfair advantage, since we do not need to build a datastructure for each coarsening level. However we have found that if we multiply the execution time of GGHEM by $1 - 0.35$ (since we found that the datastructure represent 35% of the execution time) then LGHEM still is on average 20% faster than GGHEM.

It is not unreasonable that LGHEM is faster than GGHEM, however we recall from the analysis of LGHEM and HEM, than for certain cases LGHEM uses less overall execution time than HEM. Then it seems unfair that the difference between GGHEM and LGHEM isn't larger. We have explained this the following way:

First, if we count the number of edges considered like we did in Figure

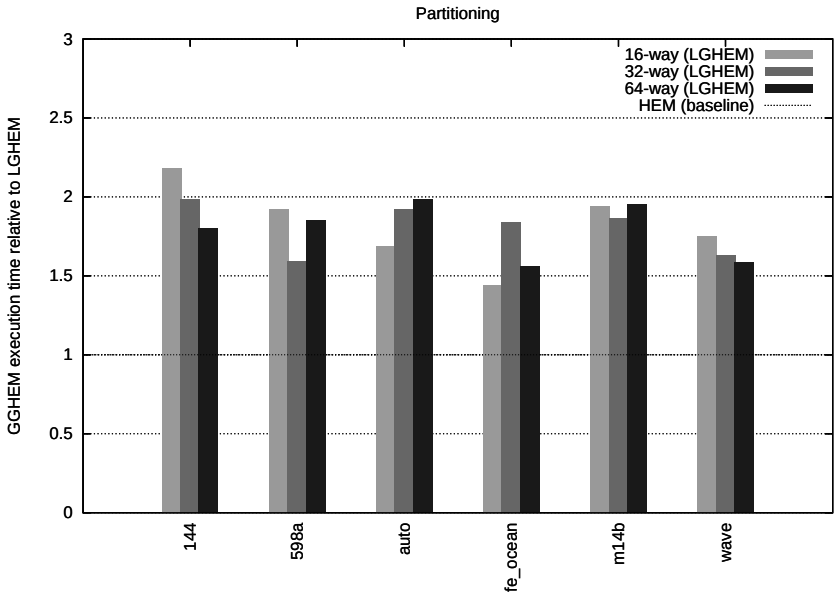


Figure 39: The total execution time of GGHEM relative to LGHEM.

33, we have found out that in the beginning LGHEM considers much fewer edges than GGHEM (Figure 41). But as the number of coarsening levels increase, so does the number of edges LGHEM considers. As seen in Figure 41, GGHEM only needs to consider more edges than LGHEM for the first coarsening level.

In the edge-cut analysis, we saw that GGHEM used fewer coarsening levels to reach the coarsening threshold. We also saw (as a consequence of fewer coarsening levels) that the average number of incident nodes where smaller for the coarser graph produced by GGHEM than LGHEM. In the analysis where we compared LGHEM and HEM, we saw that a larger number of incident nodes and a larger number of coarsening levels, were important in order to be able to refine the graph efficently. Since coarser graphs produced by GGHEM doesn't have those properties, we found that we needed to spend less time refining than LGHEM.

In Figure 41, we see that difference between the time spent on refining is generally lower for GGHEM than for LGHEM.

4.5 Comparing LGHEM and GGHEM to the graph partitioning archive (*GPA*)

The graph partitioning archive [29] is an collection of 2-way,4-way,8-way,16-way,32-way and 64-way partition vectors with the lowest known edge-cuts for a series of popular graphs.

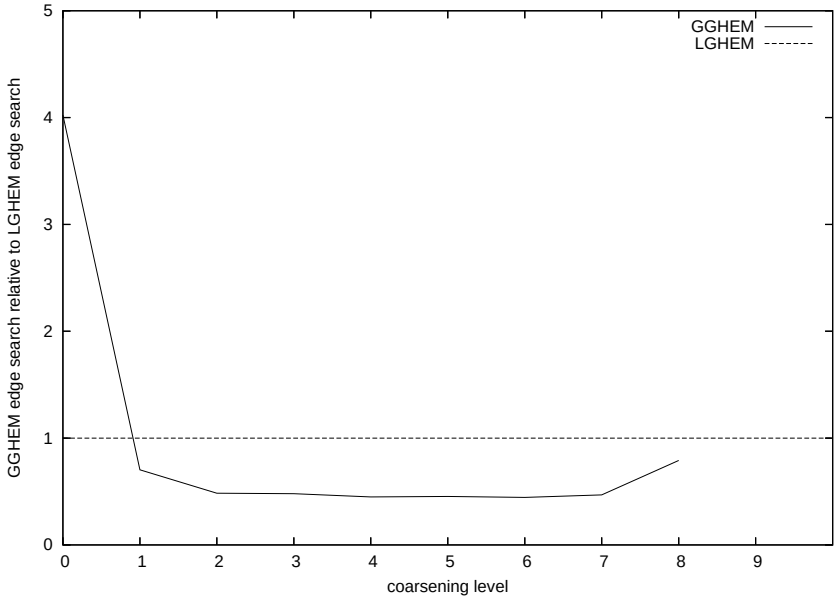


Figure 40: Number of searches relative to each edge for LGHEM and GGHEM.

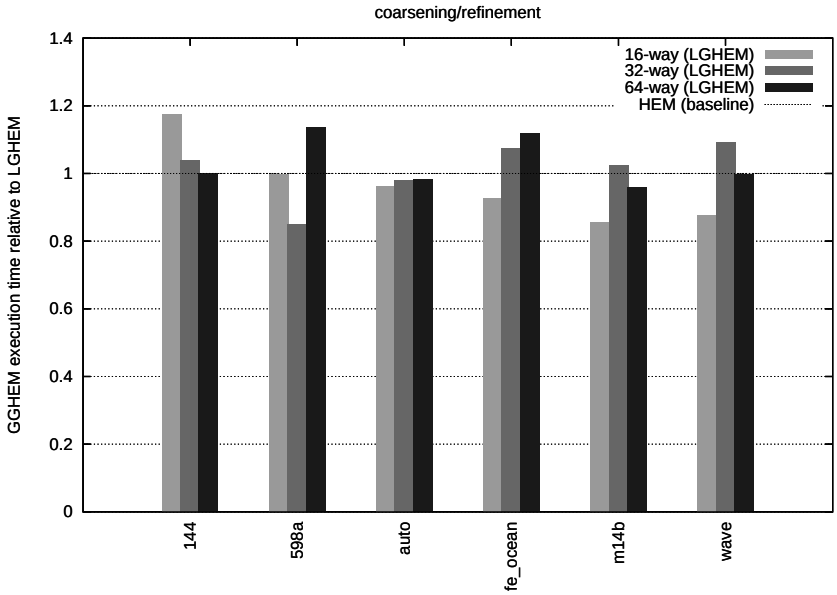
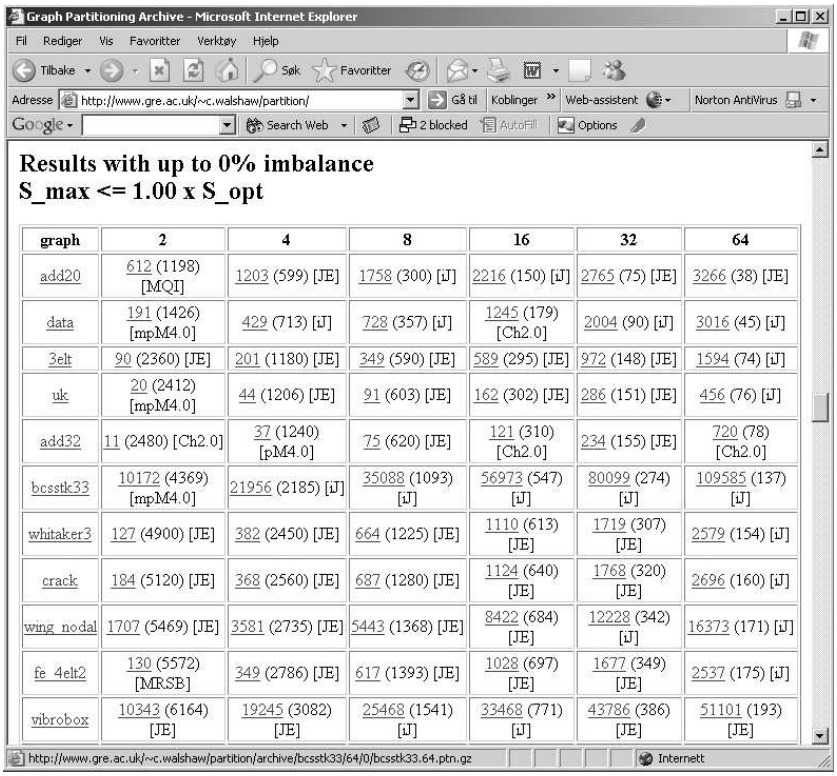


Figure 41: Execution time for refining the graph.

The archive is organised into four categories, partitions with a 0%,1%,3% and 5% load imbalance. Let P_{max} be the maximal weighted subset of a given



graph	2	4	8	16	32	64
add20	612 (1198) [MQI]	1203 (599) [JE]	1758 (300) [iJ]	2216 (150) [iJ]	2765 (75) [JE]	3266 (38) [JE]
data	191 (1426) [mpM4.0]	429 (713) [iJ]	728 (357) [iJ]	1245 (179) [Ch2.0]	2004 (90) [iJ]	3016 (45) [iJ]
3elt	90 (2360) [JE]	201 (1180) [JE]	349 (590) [JE]	589 (295) [JE]	972 (148) [JE]	1594 (74) [iJ]
uk	20 (2412) [mpM4.0]	44 (1206) [JE]	91 (603) [JE]	162 (302) [JE]	286 (151) [JE]	456 (76) [iJ]
add32	11 (2480) [Ch2.0]	37 (1240) [pM4.0]	75 (620) [JE]	121 (310) [Ch2.0]	234 (155) [JE]	720 (78) [Ch2.0]
bcsstk33	10172 (4369) [mpM4.0]	21956 (2185) [iJ]	35088 (1093) [iJ]	56973 (547) [iJ]	80099 (274) [iJ]	109585 (137) [iJ]
whitaker3	127 (4900) [JE]	382 (2450) [JE]	664 (1225) [JE]	1110 (613) [JE]	1719 (307) [JE]	2579 (154) [iJ]
crack	184 (5120) [JE]	368 (2560) [JE]	687 (1280) [JE]	1124 (640) [JE]	1768 (320) [JE]	2696 (160) [iJ]
wing_nodal	1707 (5469) [JE]	3581 (2735) [JE]	5443 (1368) [JE]	8422 (684) [JE]	12228 (342) [iJ]	16373 (171) [iJ]
fe_4elt2	130 (5572) [MRSB]	349 (2786) [JE]	617 (1393) [JE]	1028 (697) [JE]	1677 (349) [JE]	2537 (175) [iJ]
vibrobox	10343 (6164) [JE]	19245 (3082) [JE]	25468 (1541) [iJ]	33468 (771) [iJ]	43786 (386) [JE]	51101 (193) [JE]

Figure 42: The graph partitioning archive website.

partition. Then the partition fits into the 3% category if $\lceil |N|/k \times 1.03 \rceil \geq P_{max}$.

We have tested the partitions found by GGHEM and LGHEM against the existing results in *GPA*. To do this we have done the following; First we check if our load imbalance is small enough to compare it to one of the four categories. If our load imbalance is higher than 5% we simply disregard the result.

We found two new partitions with a lower than the existing edge-cuts, within the constraints of the load imbalance and several with the same edge-cut and the same load imbalance but where the partitions differs from those found in *GPA*

4.5.1 Comparing LGHEM to *GPA*

For LGHEM, 21% of the tests failed to find a partition that had a load imbalance smaller or equal to 5%. We found that if we find an edge-cut within the categories, then it has at most a 56% higher and 1% lower edge-cut than those found in the archive.

LGHEM produces an edge-cut that is equal to those found in *GPA* for five of the datasets, and on average, the results produced by LGHEM (counting those cases in which we where unable to create a imbalance lower than 5%) is 31% higher edge-cut than the results found in *GPA*.On average the difference between the load imbalance of *GPA* and the one found in partitions produced by LGHEM are very small (around 0.53%).

The difference between the load balance is obviously much smaller than the difference between the edge-cut. Since *GPA* is categorized by the load balance, we can at most (except for those cases that we did not include because they had a load imbalance higher than 5%) to be 2%.

The interesting part is obviously to look at edge-cut and load imbalance together. We see that for the 4-way partitioning of *m14b* the edge-cut when LGHEM is used was lower than the edge-cut in *GPA*. However we see that the load imbalance is higher than the one found in *GPA*.

A closer look reveals that *m14b* has an edge-cut equal to 13844, and that the maximal partition weight is equal to 53971. The best known found in *GPA*, produced by Chaco2.0 software [13] has an edge-cut equal to 14013 and a maximal partition weight equal to 53692. Clearly the result in GPA have a lower maximal weight, but our result is still within the boundary of 1% imbalance, since $\lceil 214765/4 \rceil \times 1.01 = 54229 > 53971$

We have also found that a 2-way partition of *fe_4elt* has the same edge-cut and the same load imbalance[8] as the one found in *GPA*. However, by looking at the partition vector against the partition vector from *GPA* we have found that it is not the same solution.

4.5.2 Comparing GGHEM to *GPA*

When using GGHEM as the matching heuristic, we are unable to find a partition with a load imbalance lower or equal to 5% 28% of the times. We found that if we find an edge-cut within the categories, then it has at most a 72% higher and 1% lower edge-cut than those found in the archive

We have found 2 partitions with lower edge-cut than *GPA*, and 5 partition with an edge-weight equal to the ones in *GPA*.

Looking at the edge-cut together with the load imbalance, we see that there are two cases where the edge-cut was lower than the result found in *GPA*: *data* and *auto*. For *data* we find an edge-cut which is equal to 1997, which is 7 lower than for the entry in *GPA*. However by a closer inspection we see that the load imbalance isn't 1% (it is 1.01%), so it doesn't fit into the 1% category in *GPA*. For the 3% category we see that the edge-cut is 1970 which is 27 lower than the edge-cut we found.

For the auto *dataset*, we found a 8-way partition with lower edge-cut and a lower imbalance for the upto 3% category. The edge-cut is 48329 and

[8]There is only one entry for the four categories.

the heaviest weighted subset of the partition is 57712, where the results in GPA is 48398 and 57769, which was found by Iterated Jostle [28].

We also found identical edge-cut for *cti* with 1% imbalance (originally found by Joustle Evolutionary [27]) but with a different partition vector. Just like LGHEM, we found a partition of *fe_4elt* with the same edge-cut and load imbalance as LGHEM and *GPA*. (however, the partition vector is different from both the one found by LGHEM and the one found in *GPA*).

Apart from the discovery of two new partitions, the results above gives us new evidence to the difference between the two heuristics, LGHEM and GGHEM.

We found that using LGHEM finds partitions that comes closer to *GPA* in terms of edge-cut and load imbalance than GGHEM. We can say this, since LGHEM finds more partitions that fits into the four categories. Also we note that if we find a partition in categories, it is on average 1% lower for LGHEM than GGHEM.

However, we found the number of cases where GGHEM creates a edge-cut that is lower or equal to *GPA* more often than LGHEM(recall that GGHEM found two partitions with a lower edge-cut and two partitions with indentical edge-cut). We could also argue that the partition found by GGHEM is more valueable than the partition found by LGHEM. *auto* has twice as many edges and nodes as the *m14b*. The partition of *auto* is a solution that creates a lower edge- cut and a lower imbalance than the one found in *GPA*, while the *m14b* vector only creates a lower edge-cut, the load imbalance is higher than the one found in GPA(but it is still within 1% category).

5 Conclusion & future work

5.1 Conclusion

In this masterthesis we have studied the multilevel k-way graph partitioning scheme. The idea is the following; We make smaller and smaller graphs. Once the graph is significantly smaller a k-way partition is found, then the graph and the partition is projected back to the original size.

To make the graph smaller, we combine nodes and edges, this is done with a matching heuristic. We have made one such heuristic, global greedy heavy edge matching based on another heuristic called local greedy heavy edge matching. We have inserted both into the Metis [20] graph partitioning software. Then we have measured the quality of the partitions by using the two heuristics compared to one of the heuristics already used in Metis.

We have found that the modifications we made to the graph partitioning software often resulted in high quality partitions, with low edge-cut and a low load imbalance. For local greedy heavy edge matching, we found that even though the heuristic is much more complex than already existing heuristics, using it did not increase execution time significantly. We also found out that global greedy heavy edge matching seems to produce better result than local greedy heavy edge matching but with the cost of higher execution time[9]

We found that the objective presented in [17], that minimizes the edge-weight of the smaller graph yields a low edge-cut for the initial partition. However, for the final partition there seems to be other factors as well: For instance if the matching ratio is low, then a large number of incident nodes and low edge-weight is important . If the matching ratio is high, then it seems that a large number of edges with a low edge-weight is important.

We have found two new entries to the graph partitioning archive [29], by using global greedy heavy edge matching and local greedy heavy edge matching.

5.2 Future work

In section 3.2.7 we explain how we implemented the GGHEM algorithm so that it works with a CRS datastructure. It is divided into three parts, building up a datastructure of edges, sorting the edges and making a maximal matching of the edges. Below is one suggestion on how to decrease the execution time of the GGHEM matching heuristic.

5.2.1 Improved sorting

We present an idea to an alternative approach to sorting the edges, which only requires $O(E)$, given that the edge-weights are integers. It is inspired

[9] We give guidelines on how to solve this in the next section.

by the bucket sort scheme [30]. The algorithm works the following way: We start by finding the heaviest edge-weight e_w. This can be done two different ways. We can either search through all the edges to find it, or we can calculate one edge with a maximal possible weight. Recall Figure from chapter 2. We see that the maximal possible edge-weight after a coarsening level, is $4\times$ the maximal edge-weight from the previous coarsening level.

Declare a list L, where $|L| = e_w$. Set every element in this list to 0. Then we go through each edge, find its edge-weight and increase $L[\text{edge} - \text{weight}]$ with one. We can see an example of such a list in Figure 43

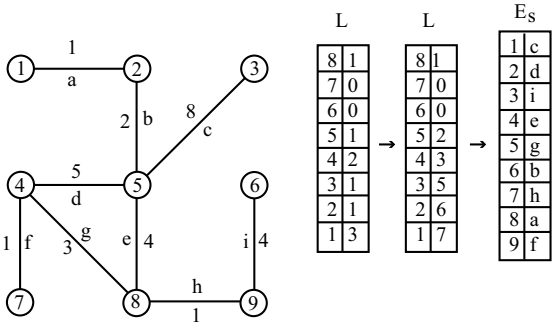


Figure 43: Sorting the edges

We declare a list E_s which contains the sorted edges. The next step is to modify L so it will serve as a lookup table when we insert the edges into E_s. We start by setting $t = 1$. Then we go through the list L from top to bottom (See Figure 43). If we at a certain position p and that $L[p] \neq 0$, then we do the following: $L[p]$ is the number of edges with a certain edge-weight. Then this will require that we allocate $L[p]$ places in E_s for the edges with this weight. (See Figure 43). More specifically do the following: If $L[p] \neq 0$, then $t2 = L[p]$, then we set $L[p] = t$.Then $t = t2$, and do this until we reach the end of the list.

Once we have modfied L, then we go through all the edges in the graph. We select one edge e and we access its edge-weight by $w(e)$. This edge should be placed in position $E_s[L[w_e]] = e$. Once the edge has been placed, then we increase $L[w_e]$ by 1. E_s will then be sorted once we have considered all the edges.

Even thought this idea has a lower complexity than introsort [23] which we currently uses, there are one uncertain factor. What will the size of L be? It is possible the L could have many empty slots, is it possible that the size of L will be so large than it will use more time than using introsort?

Bibliography

References

[1] R. Battiti and A.A. Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Trans. Comput.*, 48(4):361–385, 1999.

[2] R. Boppana. Eigenvalues and graph bisection: An average case analysis. In *In Proc. on Foundations of Computer Science*, volume 18, pages 280–285, 1987.

[3] N. Bouhmala. Greedy algorithms for partitioning graphs. Technical report, Vestfold University College, 1998.

[4] N. Bouhmala and X. Cai. Partition of unstructured finite element meshes by a multilevel approach. In *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pages 187–195. Springer-Verlag, 2001.

[5] P.K. Chan, M. D. F. Schlag, and J.Y. Zien. Spectral k-way ratio-cut partitioning and clustering. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 749–754, New York, NY, USA, 1993. ACM Press.

[6] J.L Deneboug and S.Goss. Collective patterns and decision making. *Ethology Ecology and Evolution*, 1(4):295–311, 1989.

[7] K. A. Dowsland. Simulated annealing. *Modern Heuristic Techniques for Combinatorial Problems*, pages 20–69, 1993.

[8] C. Farhat. A simple and efficient automatic fem domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[9] C.M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions, 1982.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman, 1979.

[11] F. Glover and M. Laguna. Tabu search. *Modern Heuristic Techniques for Combinatorial Problems*, pages 70–150, 1993.

[12] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallell computations. Technical report, Sandia National Laboratories, 1992.

[13] B. Hendrickson and R. Leland. The chaco user's guide, version 2.0. Technical report, Sandia National Laboratories, 1994.

[14] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 28, 1995.

[15] J.H Holland. *Adaptation in natural and artificial systems*. University of Michigan Press. Ann Arbor, MI, 1975.

[16] R. Grimes I. Duff and J. Lewis. The user's guide for the harwell-boeing sparse matrix collection (release i). http://math.nist.gov/MatrixMarket/collections/hb.html.

[17] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 29, 1995.

[18] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, 1996.

[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[20] G. Karypis and V. Kumar. Metis a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reduced orderings of sparse matrices 4.0, 1998.

[21] B.W Kernighan and S.Lin. An efficient heuristic procedure for partitioning graphs, 1970.

[22] P. Korosec, J. Silc, and B. Robic. Solving the mesh-partitioning problem with an ant-colony algorithm. *Parallel Comput.*, 30(5-6):785–801, 2004.

[23] D. R. Musser. Introspective sorting and selection algorithms, 1997.

[24] S.W Otto R. Morrison and. The scattered decomposition for finite elements problems. *Journal of Scientific Computing*, 2, 1986.

[25] H.D Simon. Partitioning of unstructured problems for parallel processing. *Computer Systems inEngineering*, 2:611–614, 1995.

[26] H.D. Simon and S.Teng. How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[27] A. J. Soper, C. Walshaw, and M. Cross. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *J. of Global Optimization*, 29(2):225–241, 2004.

[28] C. Walshaw. A Multilevel Approach to the Travelling Salesman Problem. *Oper. Res.*, 50(5):862–877, 2002. (originally published as Univ. Greenwich Tech. Rep. 00/IM/63).

[29] C. Walshaw. The graph partitioning archive, 2005. http://staffweb.cms.gre.ac.uk/∼c.walshaw/partition/.

[30] M.A. Weiss. *Data Structures and Algorithm Analysis (Second Edition)*. Addison-Wesley, 1995.

[31] E.W. Weisstein. Variation coefficient. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/VariationCoefficient.html.

[32] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. In *Concurrency: Practice and Experience*, volume 3, pages 457–481, 1991.