Partitioning Algorithms for Simultaneously Balancing Iterative and Direct Methods*

Irene Moulitsas and George Karypis

University of Minnesota, Department of Computer Science and Engineering
and Digital Technology Center and Army HPC Research Center
Minneapolis, MN 55455

{moulitsa, karypis}@cs.umn.edu

Abstract

This paper focuses on domain decomposition-based numerical simulations whose subproblems corresponding to the various subdomains are solved using sparse direct factorization methods (*e.g.*, FETI). Effective load-balancing of such computations requires that the resulting partitioning simultaneously balances the amount of time required to factor the local subproblem using direct factorization, and the number of elements assigned to each processor. Unfortunately, existing graph-partitioning algorithms cannot be used to load-balance these type of computations as they can only compute partitionings that simultaneously balance numerous constraints defined a priori on the vertices and optimize different objectives defined locally on the edges. To address this problem, we developed an algorithm that follows a *predictor-corrector* approach that first computes a high-quality partitioning of the underlying graph, and then modifies it to achieve the desired balancing constraints. During the corrector step we compute a fill reducing ordering for each partition, and then we modify the initial partitioning and ordering so that our objectives are satisfied. Experimental results show that the proposed algorithm is able to reduce the fill-in of the overweight sub-domains and achieve a considerably better balance.

Keywords: graph partitioning, parallel algorithms, domain decomposition, direct solvers, iterative solvers

*This work was supported in part by NSF CCR-9972519, EIA-9986042, ACI-9982274, ACI-0133464, and ACI-0312828; the Digital Technology Center at the University of Minnesota; and by the Army High Performance Computing Research Center (AHPCRC) under the auspices of the Department of the Army, Army Research Laboratory (ARL) under Cooperative Agreement number DAAD19-01-2-0014. The content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

1 Introduction

Existing graph partitioning algorithms aim at computing partitionings that balance constraints defined *a priori* on the vertices and optimize objectives defined locally on the edges. However, emerging high-performance computational simulations require that the resulting partitioning satisfies multiple constraints and objectives that are not local on the vertices and edges of the graph, but are defined on the overall structure of the resulting partitioning.

Examples of such computations are domain decomposition-based numerical simulations where the subproblems corresponding to the various subdomains are solved using a sparse direct factorization e.g., Finite Element Tearing and Interconnecting (FETI) domain decomposition method [5]. Effective load-balancing of such computations requires that the resulting partitioning simultaneously balances the amount of time required to factor the local subproblem using direct factorization, and the number of elements assigned to each processor. In this example, the constraints and objectives cannot be modeled by assigning different weights on the vertices and/or edges of the graph, as they depend on the overall structure of the partitioning. For example, in the case of a simulation using FETI, the amount of time required by each subdomain, depends on the fill-in of the factors in each subdomain, which can only be determined once a partitioning has been computed, and the corresponding matrices have been re-ordered using a fill-reducing ordering. Therefore, this optimization leads to constraints that are dynamic in nature, as they cannot be evaluated unless a partitioning has been computed.

The limitations of traditional partitioning algorithms for solving problems with dynamic constraints and objectives has been recognized some time ago [24, 13, 14]. For example, many researchers have recognized that computing partitionings that balance the number of vertices, and minimize the edgecut, is not sufficient to balance the computations performed by a per-subdomain direct factorization [24, 22]. Despite that, none of the approaches that were proposed to solve the problem led to solutions that are both computationally efficient and/or are guaranteed to ensure load balance. To a large extent, one of the reasons that these issues have not yet been rigorously addressed is that it was not until recently that the problem of computing high-quality partitionings was well understood, and fast and high-quality algorithms were developed for solving them. Moreover, developing algorithms for solving such problems is particularly challenging, as it requires the partitioning algorithm to balance and optimize quantities that can be measured only after a partitioning has already been computed.

In this paper we present a graph-partitioning algorithm for balancing and optimizing multiple constraints and objectives that are defined on the overall structure of the partitioning solution. Our research has focused on developing such partitioning algorithms to address the classes of problems where we need to compute a partitioning and a per-subdomain fill-reducing ordering that simultaneously balances the number of elements assigned to each partition, and the fill-in of the subdomain matrices, while minimizing the edgecut. Since these constraints and objectives cannot be determined/evaluated prior to actually knowing the partitioning of the graph, the proposed algorithms follow a *predictor-corrector* approach. That is, they first compute a high-quality partitioning of the underlying graph and then modify this partitioning to achieve the desired balancing

and optimization goals.

The rest of the paper is organized as follows. Section 2 gives a brief overview of graph partitioning and definitions of basic terminology. In Section 3 we explain why existing techniques are deemed insufficient for some types of scientific problems. We propose a new approach and we describe the variations and details of our algorithm. The experimental evaluation of the algorithms is shown in Section 4. Finally, Section 5 provides some concluding remarks.

2 Graph Partitioning Background

In this section we will give the definitions of graph partitioning terms that we will use later on, and review briefly some existing graph partitioning techniques.

The graph partitioning problem can be defined as: Given a graph $G = (V, E)$, where V is the set of vertices, $n = |V|$ is the number of vertices, and E is the set of edges in the graph, partition the vertices to p sets $V_1, ..., V_p$ such that $V_i \cap V_j = \emptyset$ for $i \neq j$. This is called a p-way partitioning and we denote it by P. Every one of the subsets V_i, of the vertices of G, is called a partition or subdomain. The graph G is a weighted graph if every vertex and edge has an associated weight. By the **edgecut** of a partition, we mean the sum of the weights of the edges in E that were cut among different subdomains. If we look among all the individual partitions, we define **maxcut** as the maximum cut in edges from among all subdomains. The **total vertex weight** of graph G is the sum of the weights of all the vertices in V and is denoted by $w(V)$. The **partition weight** of partition V_i is the sum of the weights of the vertices assigned to V_i and is denoted by $w(V_i)$. The **load (im)balance** of a p-way partitioning is defined as

$$LoadImbalance(P) = \frac{max_i(w(V_i))}{w(V)/p},$$

which is the ratio of the highest partition weight over the average partition weight.

For the traditional graph partition problem the goal is to compute a p-way partitioning P, such that for ϵ adequately small, and $\epsilon > 0$, $LoadImbalance(P) \leq 1+\epsilon$, and $edgecut(P)$ is minimized. The load imbalance is the constraint we have to satisfy, and the edgecut is the objective we have to minimize. Therefore graph partitioning can be viewed as an optimization problem.

Graph partitioning is an NP complete problem [7]. Therefore, many heuristic algorithms have been developed [12, 4, 2, 3, 6, 9, 15, 16, 19, 20, 25, 23] to address it. A very popular approach to handle the graph partitioning problem is the multilevel paradigm. Such algorithms [16, 19, 20, 25, 23] are very popular due to the fact that they can produce high quality partitions, they are fast, and they can be extended to run in parallel environments, therefore scaling to graphs that have several million of vertices.

Multilevel graph partitioning algorithms consist of three different phases :

1. Graph Coarsening
2. Initial partitioning of the coarse graph
3. Uncoarsening of the partitioned coarse graph

The idea is that since the original graph might be too large to work with, we first try to coarsen the graph down to another graph with fewer vertices, by collapsing together vertices of the original graph [16, 19, 20]. Once the graph is coarse enough, then this new graph can be split into p parts fairly easily. During the uncoarsening phase, the initial partitioning of the coarse graph is projected back to the original fine graph. Since a finer graph has more degrees of freedom, vertices can move among partitions so that the initial partitioning can be improved [16, 19, 20].

3 Problem Definition and Challenges

Consider a graph $G = (V, E)$ with $n = |V|$ vertices corresponding to an $n \times n$ symmetric matrix A, and let P be a partition of its vertices into p parts $V_1, V_2, ..., V_p$. For $i = 1, 2, ..., p$, let $G_i = (V_i, E_i)$ be the induced subgraph of G, and A_i be the matrix corresponding to that induced subgraph. Given these definitions, the particular partitioning problem that we are trying to solve is as follows: Compute a partitioning P of G and a fill-reducing ordering of each submatrix A_i such that:

1. each partition has n/p vertices;
2. the fill incurred by each submatrix is the same;
3. the fill of each submatrix is minimized; and
4. the edge-cut is minimized.

Note that this partitioning problem definition accounts both for minimizing the amount of time spent during the iterative phases (by balancing the partitions and minimizing the edge-cut), and for minimizing the amount of time spent in the various direct factorizations and subsequent triangular solves (by balancing and minimizing the fill).

Even though it is quite straight-forward to compute a partitioning and the associated fill-reducing orderings that satisfies conditions $1, 3$, and 4, it is quite hard to simultaneously satisfy condition 2. This is because the amount of fill incurred by a particular subdomain depends on the topological properties of its corresponding subgraph and the characteristics of the fill-reducing algorithm that is used. In most cases involving irregular matrices, there is no easy way to determine the work without actually computing a fill-reducing ordering followed by symbolic factorization. Moreover, computing fill-reducing orderings and their associated symbolic factorizations is quite expensive and cannot be done repetitively.

3.1 Predictor-Corrector Partitioning and Fill-Reducing Ordering Algorithm

To address these issues we developed a three-phase approach for solving this problem. During the first phase we compute a p-way partitioning of the graph that satisfies conditions 1 and 4. Then, in the second phase we compute a fill-reducing ordering of each subdomain and determine the fill by computing a symbolic factorization. Finally, in the third phase, we modify both the initial partitioning and the fill-reducing orderings of each subdomain to correct any observed load-imbalances. These last two steps allow us to compute a partitioning that also satisfies conditions 2 and 3.

The key idea of this approach is motivated by the observation that in the case of problems derived from finite element meshes, even though the exact fill-ins of each submatrix are hard to determine a priori, they tend to follow a function that has in order terms a similar asymptotic growth for the various partitions. That is, the fill incurred by each partition V_i will be of the order of $|V_i|^{\alpha+\epsilon_i}$, where α is fixed across the partitions and ϵ_i is a small constant ($\epsilon_i << \alpha$) that depends on the particular topological characteristics of G_i. Thus, the fill imbalance of the initial k-way partitioning will not be arbitrarily bad. For this reason, a scheme that starts from that partitioning and attempts to perform relatively small modifications can still achieve load balance, in terms of fill-in. We will refer to this as a *predictor-corrector* approach, as we use the first two phases to predict the partition-fill-reducing ordering, and the third phase to correct it so that it becomes balanced.

Achieving the first phase of the proposed approach is quite straightforward and any of the existing graph partitioning algorithms can be used. For our implementation we relied on the multilevel k-way partitioning algorithm developed in [20], and that is available in the METIS 4.0 [18] package (`pmetis`). This algorithm is known to produce partitionings that are well balanced, have a low edgecut, and low computational requirements. Therefore, we start with an initial partitioning that satisfies conditions 1 and 4.

For the second phase we used an ordering algorithm based on multilevel nested dissection described in [20], and implemented in METIS 4.0 [18]. Given a graph, nested dissection algorithms compute a fill-reducing ordering using a divide-and-conquer approach. For each graph, they first find a minimum size vertex-separator S that partitions the graph into two relatively balanced sets, A and B such that vertices in A are not connected to vertices in B. The vertices in A are ordered first, followed by the vertices in B, followed by the vertices in S. The exact ordering of the vertices in S is not important, but the exact ordering of the vertices in A and B is determined by performing the same vertex-separator discovery process recursively in each of the subgraphs induced by A and B.

The choice of a nested dissection based ordering algorithm is critical for two reasons. First, in the last few years, fill-reducing ordering algorithms based on nested-dissection have improved significantly (primarily due to the development of high-quality multilevel algorithms for finding small vertex separators [20, 17, 18]), and are now considered the state-of-the-art method for computing fill-reducing orderings. Second, they provide a reasonably easy way to account for the amount of fill that is created as a function of the size of the separators. In particular, each vertex separator translates to a dense submatrix whose size is equal to the number of vertices in the separator. Thus, the fill incurred by a nested-dissection ordering can be modified by manipulating the size of these separators.

We use this observation to develop an algorithm that simultaneously modifies the partitioning and the fill-reducing orderings for the third phase of the proposed algorithm. In particular, we develop partitioning refinement algorithms that move vertices between partitions such that the size of the top-level separators of the domains with high fill is reduced; thus, reducing the overall load-imbalance. Note that the proposed approach does not try to reduce the size of a separator by refining it locally within the subdomain, because these separators are already very small and

in general cannot be reduced any further. Instead, the proposed partitioning refinement algorithm tries to move vertices that are at the separators of an overweight domain to one of the adjacent subdomains, which reduces the size of the separator S and ensures that the subsubdomains (e.g., A and B in the previous example) remain disconnected.

3.2 Separator Refinement Algorithm

The key challenge in the above predictor-corrector scheme is how we select which vertices to move between partitions so that we balance the fill, while ensuring that the characteristics of the existing partition in terms of conditions 1, 2, and 4 remain the same (i.e. each partition has the same number of vertices, the fill-in and the edgecut are minimized).

To address this challenge we developed an iterative heuristic refinement algorithm that first estimates by how much each top level separator needs to be decreased (if any), and then follows a randomized greedy refinement framework. The refinement framework moves vertices between partitions so that it achieves the desired top level separator sizes, without significantly degrading the quality characteristics of the initial partitioning. Details on how the above steps are performed are provided in the rest of this section.

3.2.1 Determining the Size of the Separators

We determine which of the partitions have a high fill-in and by how much it should be reduced as follows. First we compute the average fill-ins over all the partitions. For these partitions that have a fill-in that is less than 5% over the average fill-in we do not attempt to lower their fill-in, (i.e., we assume that they are more or less balanced). For the remaining partitions, we try to reduce their fill-in, so that it becomes equal to the average fill-in. This fill-in reduction is achieved by moving vertices out of their top-level separators. In order to decide how many vertices we need to move, we rely on the fact that there is a quadratic relation between the size of a separator and the amount of the fill-in that it generates. Specifically, if F is the fill-in due to a top level separator of size s, we assume that there is a simple $F = c * s^2$ relation between F and s for some constant c, and moreover c does not change as we decrease s. Based on this assumption, the new top level separator size s' that will decrease the fill-in from F to F' is given by

$$F' = c * {s'}^2 \Leftrightarrow F' = \frac{F}{s^2} * {s'}^2 \Leftrightarrow s' = \sqrt{\frac{F'}{F}} * s \quad (1)$$

Thus, by knowing the average fill-in of an “overweight” partition we can use Equation 1 to estimate the size of the top level separator that will result in the desired fill-in reduction (assuming the rest of the partition remains fixed).

3.2.2 Determining which Vertices to Move

Since the size of the separator is by construction minimal, the candidate vertices will need to be moved to a different partition. However, there are multiple potential problems that can arise if these moves are not made with extreme care. The first problem is that most of the times these moves translate into an increase in the edgecut. The second problem is that by moving a vertex

into a different partition, we perturb the fill of that partition. Even though the fill of the originating partition is guaranteed to go down, the fill-in of the destination partition can be increased. The third problem is that the load imbalance of the partitions can be perturbed. Even though the initial load imbalance of the partitioning, as given to us by METIS, was good, by moving vertices around we run the risk of creating a partition with high load imbalance.

In order to avoid encountering the above problems, our refinement algorithm has to follow some rules. These rules vary depending on, whether, the vertex that can potentially be moved, belongs to a separator or not. The reason we do not follow the same procedure in both cases is the following. A separator vertex is moved out of necessity, therefore we are willing to make a move even if it somehow increases the edgecut, or perturbs the load imbalance. On the other hand, we move non-separator vertices only to reduce edgecut or improve the balance. As mentioned already, we also need to establish some metrics. One of them is the maximum partition weight, that we do not allow to be more than 3% above the average partition weight. This metric will be used in order to ensure that our partitions will be load balanced. As we mentioned before, for every partition we have the vertex separator S, that divides the partition into two relatively balanced sets, A and B. In order to ensure that A and B remain balanced we set the maximum subsubdomain weight not to be more than 10% above the average.

For these vertices that do not belong to a separator we set the following rules. If moving this vertex will increase the destination partition weight beyond the maximum allowed, then this move should not be performed. Also, the move is not be allowed, if it increases the subsubdomain weight beyond the maximum. By following these two rules we ensure the load balance of our partitions. Another rule is the following, if the vertex has more than one candidate partitions as destinations, then we choose the partition to which the vertex is most connected, and at no time is a move allowed to increase the edgecut. Finally, if the separator size of the destination partition is larger than what we want it to be, then we do not allow any moves into this partition. The reason we do not allow such moves is because we try to balance the fill of the partitions. Therefore, we would not like to cause unnecessary fill to partitions that already have a fill higher than the average.

With regards to the load balance of the partitions, we follow the same rules for separator vertices, as with non-separator vertices. Therefore, we do not allow separator vertices to move to a partition, if this move will increase the partition weight, or the subsubdomain weight above our preset limits. In our effort to keep the edgecut as low as possible, we follow the described tactic. As long as the separator size of the partition has not been reduced to the desired point, we allow separator vertices to move to other partitions, even if such moves increase the edgecut. However, once the separator size has been reduced as much as we wanted, then we do not allow any moves that will increase the edgecut. Whenever a separator vertex can be moved to more than one partitions, we choose the partition that is most connected to it. In order to ensure the fill balance of the partitions, a separator vertex is not allowed to move to a partition that has a higher fill. Also, in order to ensure that the fill of every partition is kept minimal, separator vertices are not allowed to move to partitions that have a fill that is above the average fill.

3.2.3 Determining the Fill-in of Each Partition

In our vertex move selection procedure one of the criteria utilized the fill of the originating and/or destination partition. Initially, we have a very good estimate of the fill-in through the symbolic factorization. However, once we start moving vertices between partitions, then the fill-in is perturbed. Therefore we need to find a way to estimate the fill of every partition after every move. To achieve this, we have two options. One option is to perform a symbolic factorization after the move. Even though this is a straightforward solution, it is a time consuming one too. The second option is to model how a vertex move will affect the fill-in of its original and its target partition. Let F be the fill of a partition let n be the number of its vertices. Then it is true that

$$F = n^{\alpha} \text{ where } \alpha = \text{constant}$$

Therefore for the constant α we can say that

$$\alpha = \frac{\log F}{\log n}$$

Let F be the original fill-in before the move, and F', be the fill-in after the move. Similarly, let n be the original partition weight, and n', be the weight after the move. Then, assuming that both F and F' grow in the same way we have

$$F' = n'^{\alpha} \Leftrightarrow F' - F = n'^{\alpha} - n^{\alpha} \Leftrightarrow F' = F + n'^{\frac{logF}{log\,n}} - n^{\frac{logF}{log\,n}} \tag{2}$$

Therefore, after every vertex move we can update the fill-in information of the original and destination partition by using Equation 2. For the original partition $n' = n - w(v)$, and for the destination partition $n' = n + w(v)$, where $w(v)$ is the vertex weight, of the moved vertex.

3.2.4 Putting Everything Together

Based on the above key points we have developed five different refinement algorithms that give different emphasis on the above parameters.

RS1 The first scheme is a crude refinement scheme. First we find by how much the separator size should be reduced, using Equation 1. Once we know this, then we start the refinement procedure, exactly as described before. This algorithm is the most aggressive of all, in terms of the way it handles the edgecut. The reason is that it allows separator vertices to move to other to partitions even when the increase in the edgecut is high. After every move we recompute the fill using Equation 2.

RS2 The algorithm moves vertices in different passes. The criterion of moving a separator vertex, with regards to edgecut, is more relaxed with every successive pass. During the first pass only the separator vertices that incur the lowest cost are moved. But, in later passes, nodes of increasing cost are allowed to be moved. The target separator size is determined by using Equation 1. The rest of the refinement procedure is exactly as described before and the fill-in is updated using Equation 2.

RS3 In this algorithm we follow the relaxation procedure of the previous version. However, separator vertex moves are made in a more relaxed fashion not only based on the pass, but also based on the size of the separator compared to the average separator size. Therefore, if a partition has a large separator then we will make more relaxed moves out of this partition. In other words, we are more willing to increase the edgecut if this move is to help a partition with a large separator. The rest of the refinement procedure follows the tactic we described in the general discussion before, and the fill-in is updated using Equation 2.

RS4 In this scheme we compute the target separator size using Equation 1. The algorithm makes relaxed moves, exactly as described in RS2. But, instead of using our model in Equation 2 to estimate the fill-in of a partition, we perform a symbolic factorization at the end of every move.

RS5 Just as in the previous schemes, the target separator size is computed using Equation 1. In this scheme, we perform relaxed moves, just as in RS2. However, all of our decisions are based on the size of the separator, rather than the fill-in of the partition. In this scheme we do not need to use the model that describes the fill-in update, since we consider a partition to be ”overweight” based only on the size of the top level separator.

4 Experimental Results

We evaluated the performance of our algorithms using a wide variety of graphs whose description and characteristics are shown in Table 1. The size of these graphs ranged from 14K to 1.1M vertices.

Table 1: Characteristics of the test data sets.

	Name	# Vertices	# Edges	Description
1	144	144,649	1,074,393	Graph corresponding to a 3D FEM mesh of a parafoil
2	auto	448,695	3,314,611	Graph corresponding to a 3D FEM mesh of GM's Saturn
3	bcsstk29	13,992	302748	Graph corresponding to Buckling model of a Boeing 767 rear pressure bulkhead
4	brack2	62,631	366,559	Graph corresponding to a 3D FEM mesh of a bracket
5	cylinder93	45,594	1,786,725	Graph of a 3D stiffness matrix
6	f16	1,124,648	7,625,318	Graph corresponding to a 3D FEM mesh of F16 wing
7	f22	428,748	3,055,361	Graph corresponding to a 3D FEM mesh of F22 wing
8	finan512	74,752	261,120	Graph of a stochastic programming matrix for financial portfolio optimization
9	inpro1	46,949	1,117,809	Graph corresponding to a 3D stiffness matrix
10	m6n	94,493	666,569	Graph corresponding to a 3D FEM mesh of M6 wing

In the rest of this section we compare the results for all five variations of our proposed algorithm. Labels RS1, RS2, RS3, RS4, RS5, correspond to our fill balancing algorithms as described in Section 3.2. Whenever used, Original corresponds to the results of the traditional `pmetis` partitioning algorithm [20]. We choose to use these names so that the following tables and graphs are easily readable.

4.1 Quality of the Results

Our first set of experiments focused on evaluating the quality of the partitions produced using the different variations of our proposed algorithm. In particular, the upcoming figures show the results

for the graph partitioning algorithms: Original, RS1, RS2, RS3, RS4, RS5. The evaluation was performed on the ten data sets, numbered $1 - 10$, as they appear in Table 1. Due to space limitations we only present the results for a 64-way partition.

For every data set there are five different qualities that we are going to assess in order to decide which of the proposed fill balancing algorithms works best most of the times.

4.1.1 Fill Balance Assessment

Similarly to the LoadImbalance, defined in Section 2, we define the **fill-in (im)balance** of a p-way partitioning as

$$FillImbalance(P) = \frac{max_i(Fill(V_i))}{(\sum_i Fill(V_i))/p},$$

which is the ratio of the highest partition fill-in over the average partition fill-in. Figure 1(a) shows the overall fill-in balance of the resulting partitioning for all different methods, as well as the fill-in balance of the Original (pmetis) algorithm.

Looking at the results in this figure we can see that for all the datasets, Original produces partitions with a higher fill imbalance. On the other hand our proposed methods are able to improve the balance of the resulting partitioning, verifying the validity of our approach. For example, for the 64-way partition of the sixth data set, we were able to improve the imbalance from 1.28 to 1.10 which is an improvement of 64%. In general we can observe that RS5 (moves made based on the separator size) tends to attain a worse fill balance, although this is not always the case. Based on all the experiments we have made, and not just the ones presented, RS1 (aggressive) gives a better balancing in general.

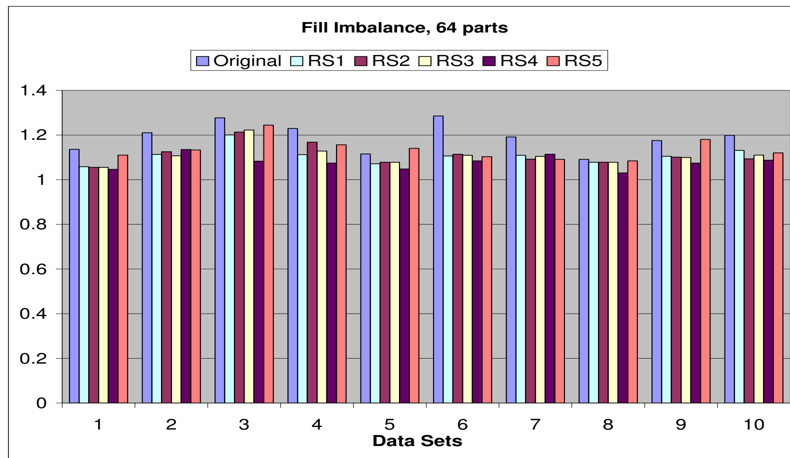
4.1.2 Average Fill Assessment

For each one of our experiments we computed the average of the fills over all the individual partitions. In Figure 1(b) we present the average fill of our methods relative to that produced by the Original `(pmetis)` algorithm.

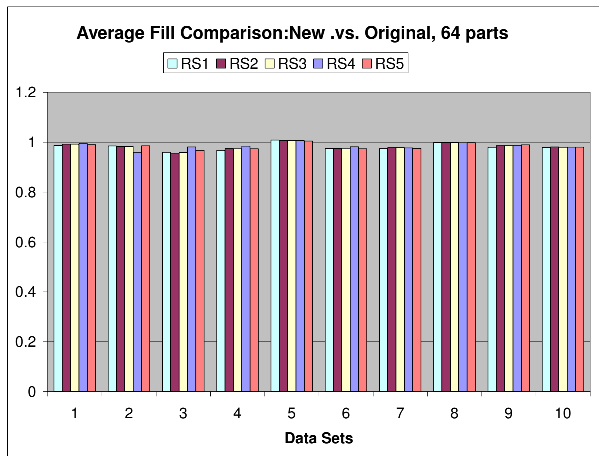
From this figure we can easily see that almost all of the time our algorithms produce partitions with lower fill (ratio < 1). In fact only once were we not able to do so. This is a particularly important achievement because we understand that one of the leading limitations of solving large problems is the memory of the underlying architecture. Therefore, by lowering the fill, we are able to lower the memory needs for the rest of the execution of the scientific application. From the figure, and our overall experience, we can make one more observation. Different methods seem to perform better or worse than others, depending on the data set and the k-way partition (8-way, 64-way etc.). However, by carefully looking at our results, we concluded that RS1 (aggressive) is the winner since it outperformed all other methods most of the time.

4.1.3 Maximum Fill Assessment

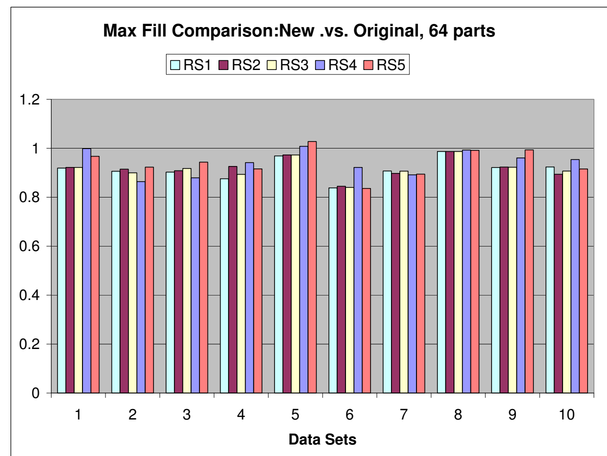
One more characteristic to compare against is the maximum fill accrued, i.e. for every experiment we found the partition that had the largest fill. We have to keep in mind that partitions with larger fills will be more time and memory consuming. Therefore it is important to be able to keep this



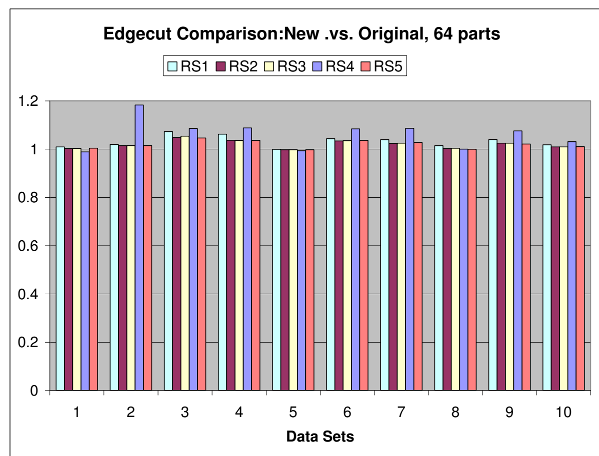
a



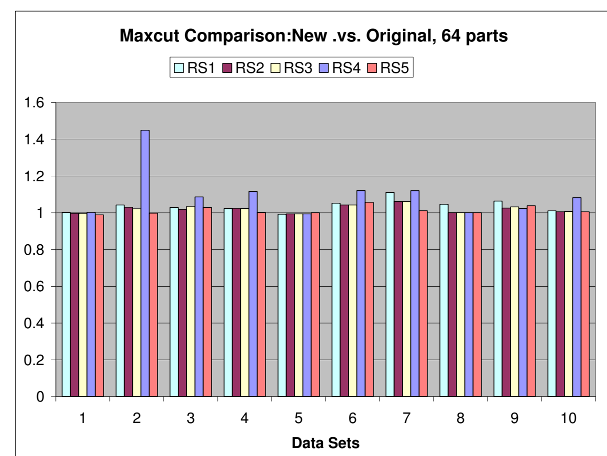
b



c



d



e

Figure 1: Characteristics of the induced 64-way partitioning.

maximum fill as low as possible. The maximum fill relative to that produced by the Original (pmetis) algorithm is graphically shown in Figure 1(c).

Again, our proposed methods seem to be successful since in 85% of the experiments we were able to lower the maximum fill. Only 4% of the times that our algorithms failed did we actually produce a larger maximum fill. In the rest of the experiments the change was actually insignificant. A more thorough examination of our results showed that RS1 (aggressive) tends to produce the best results.

4.1.4 Edge-Cut Assessment

One more measure to be evaluated is the total edgecut of the partitioning. The edgecut is an indicator of the total communication that will need to be performed later; larger edgecut indicates that more information will need to be exchanged within different partitions. The edgecut of our proposed algorithms relative to that produced by the Original (pmetis) algorithm is graphically shown in Figure 1(d).

We can see that in most of the cases our algorithms produce partitions that incur higher edgecut. However, this is something that we had expected for the simple reason that we were willing to sacrifice some communication quality as long as we obtain better fill balance. We can also say that in general RS5 (moves made based on the separator size) is the method that sacrifices the least in terms of edgecut.

4.1.5 Max-Cut Assessment

The last quality measure that we are consider is the maxcut, i.e. for each experiment we identify the highest edgecut that one single partition incurs. This measure indicates the maximum data exchange taking place within two individual partitions. The results are shown in Figure 1(e).

As we see, our algorithms usually result in partitions with higher maxcut. This is again due to the fact that we sacrifice in communication quality in order to achieve a better fill balance. For the seventh data set there is some increase in maxcut, but for the rest of the experiments we ran the increase is not significant. Again, RS5 (moves made based on the separator size) is the method that usually produces the best results.

4.2 Computational Requirements

Our second set of experiments focused on timing the different variations of the proposed algorithm. In Table 2 we show the run times, in seconds, needed to compute an 8-way partition.

We see that four out of the five algorithms have comparable times. However, RS4 (explicit computation of fill-in) seems to take roughly four times that long. It should come to no surprise to us that RS4 is so expensive. The reason is inherent to the algorithmic procedure that we followed there. Since for every pass of the algorithm we computed the exact fill-in information for each partition, we spent a large amount of time for this step.

On the other hand, all of the other methods require roughly the same time. Compared to the the Original algorithm, our algorithms always perform within a factor of 2. Therefore, we have attained to solve the problem in question using algorithms that perform within expectations.

Table 2: Run times required for computing an 8-way partition (in secs).

	Original	RS1	RS2	RS3	RS4	RS5
144	14.46	27.07	27.08	27.06	**112.65**	27.00
auto	60.24	116.47	116.86	116.78	**494.00**	116.63
b29	1.05	1.57	1.62	1.59	**4.29**	1.6
brack2	3.66	6.16	6.09	6.10	**25.89**	6.07
cylinder93	9.78	15.96	15.99	16.02	**70.73**	15.37
f16	157.98	275.27	275.70	277.13	**1361.49**	275.11
f22	52.10	90.18	90.72	90.72	**434.39**	90.68
finan512	4.01	6.31	6.29	6.31	**16.24**	6.78
inpro1	5.53	8.55	8.88	8.87	**29.14**	8.90
m6n	7.94	13.50	13.49	13.49	**60.39**	13.47

We would like to note here that with the help of RS4 we were able to verify the validity of our model for computing a new approximation to the fill-in after each move, without explicitly computing the fill-in every time. Indeed, the quality (see Section 4.1 for our definitions of quality measures) of the partitions induced by RS4, was similar to the quality of the rest of our other algorithms.

4.3 Characteristics of the Induced Partitions

In this section we are going to present several characteristics of the individual partitions produced by our algorithm. We will refer to our algorithm as `kfmetis`, and for simplicity we present the results of RS1 (aggressive). We would like to note here that for all of our experiments the weight imbalance of each partition, in terms of the vertex weight, has been kept within 3%.

4.3.1 Factor Size.

An interesting comparison is the number of non-zeros of the various partitions, and how their size is affected by our algorithm. In Figure 2 we show two such examples. The first one corresponds to an 8-way partition of the `auto` data set, whereas the second one corresponds to a 16-way partition of the `144` data set.

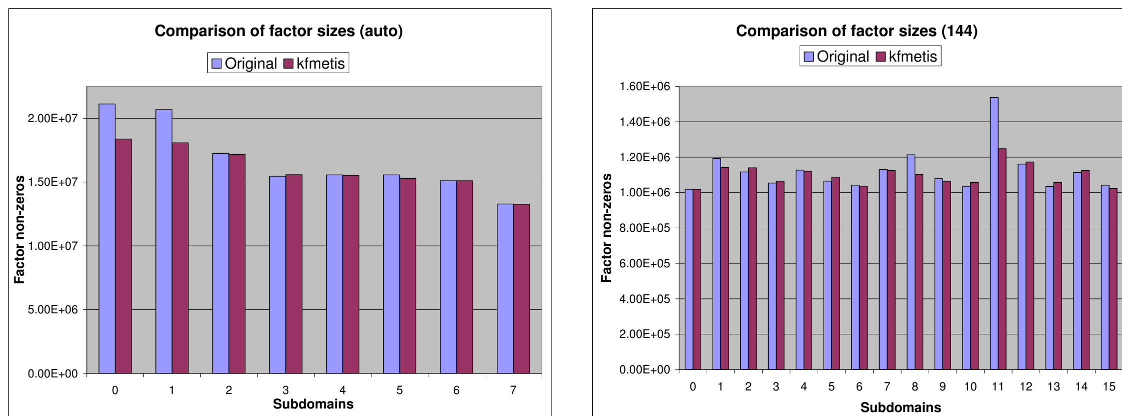


Figure 2: Load balancing improvements achieved by **kfmetis.**

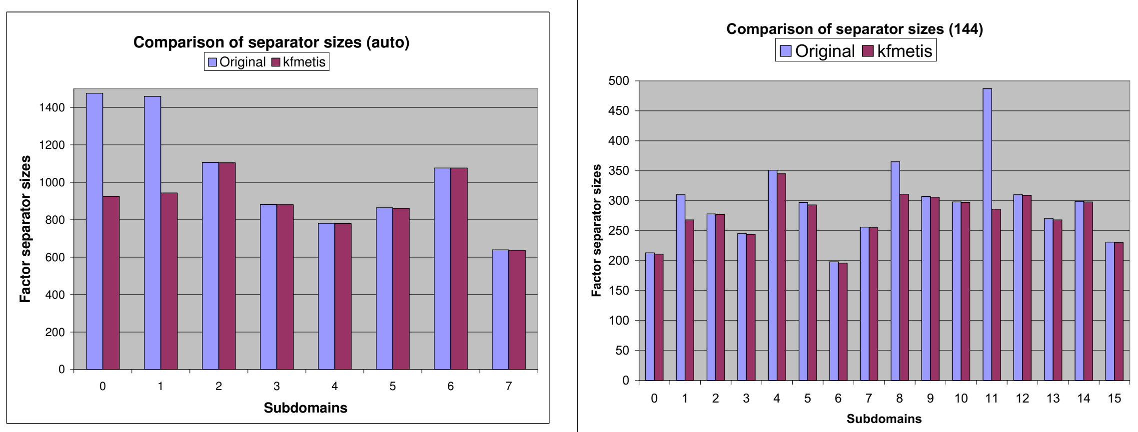


Figure 3: Separator Size balancing improvements achieved by **kfmetis.**

We see that in both cases the proposed algorithm is able to substantially reduce the fill-in of the overweight subdomains. For `auto`, the overweight subdomains are 0 and 1 and are reduced roughly by 13%. For data set `144`, partition 11 is the overweight subdomain and is reduced roughly by 19%.

4.3.2 Separator Size.

Another interesting comparison is to look at the number of vertices that are on the separator of the various partitions, and see how the separator size is affected by kfmetis. In Figure 3 we show the separator size of each partition for the same experiments that were presented in Section 4.3.1.

In both cases kfmetis was successful in cutting down the size of the larger separators. For `auto` the decrease in the separator sizes of the first two subdomains was roughly 37%. For data set `144` the decrease in the separator size of partition 11 is 40%.

Finally it is worth pointing out the similarity between the graphs that appear in Figure 2 and those of Figure 3. Partitions with larger separators tend to have a larger fill and the reduction in the larger separator size is reflected in the reduction of the fill-in of that partition.

5 Conclusions and Directions for Future Research

In this paper we presented a set of *predictor-corrector* graph partitioning algorithms that addresses the classes of problems where we need to compute a partitioning and a per-subdomain fill-reducing ordering that simultaneously balances the number of elements assigned to each processor and the fill-in of the subdomain matrices while minimizing the edgecut. Our results show that our algorithm has been successful in balancing the fill-ins, when unbalanced, while being able to maintain the quality of the partition with regards to the edgecut.

The algorithms presented here can be improved along two directions. First, the refinement approach can be extened to take account separators at lower levels of the nested dissection tree. This will allow for improved balance and edgecut. Second, the refinement algorithm can be applied in a multilevel fashion, increasing its effectiveness by achieving the desired balance, while limiting the cut degradation. We are currently investigating both of these directions.

References

[1] Cleve Ashcraft and J. W. H. Liu. A partition improvement algorithm for generalized nested dissection. Technical Report BCSTECH-94-020, Boeing Computer Services, Seattle, WA, 1994.

[2] Stephen T. Barnard. Pmrsb: Parallel multilevel recursive spectral bisection. In *Supercomputing 1995*, 1995.

[3] Stephen T. Barnard and Horst Simon. A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 627–632, 1995.

[4] Pedro Diniz, Steve Plimpton, Bruce Hendrickson, and Robert Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proceedings of the seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 615–620, 1995.

[5] Charbel Farhat and Francois-Xavier Roux. An unconventional domain decomposition method for an efficient parallel solution of large-scale nite element systems. *SIAM J. Sci. Stat. Comput.*, 13(1):379–396, 1992.

[6] C. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *In Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, CA, 1979.

[8] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[9] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. In *Proceedings of International Parallel Processing Symposium*, 1995.

[10] Todd Goehring and Yousef Saad. Heuristic algorithms for automatic graph partitioning. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

[11] M. T. Heath, E. G.-Y. Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991. Also appears in K. A. Gallivan et al. *Parallel Algorithms for Matrix Computations*. SIAM, Philadelphia, PA, 1990.

[12] M. T. Heath and Padma Raghavan. A Cartesian parallel nested dissection algorithm. *SIAM Journal of Matrix Analysis and Applications*, 16(1):235–253, 1995.

[13] Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proc. of Irregular 1998*, 1998.

[14] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000.

[15] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, Sandia National Laboratories, 1992.

[16] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[17] Bruce Hendrickson and Edward Rothberg. Improving the runtime and quality of nested dissection ordering. Technical Report CS-96-000, Sandia National Laboratories, 1996.

[18] George Karypis and Vipin Kumar. METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1998. Available on the WWW at URL *http://www.cs.umn.edu/˜metis*.

[19] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998. Also available on WWW at URL http://www.cs.umn.edu/˜karypis.

[20] George Karypis and Vipin Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999. Also available on WWW at URL http://www.cs.umn.edu/˜karypis. A short version appears in Intl. Conf. on Parallel Processing 1995.

[21] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Algorithm Design and Analysis*. Benjamin Cummings/ Addison Wesley, Redwod City, 1994.

[22] Ali Pinar and Bruce Hendrickson. Partitioning for complex objectives. In *Proceedings of International Parallel and Distributed Processing Symposium*, pages 121–123, 2001.

[23] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editor, *CRPC Parallel Computing Handbook.* Morgan Kaufmann, 2000.

[24] Denis Vanderstraeten, Ronald Keunings, and Charbel Farhat. Beyond conventional mesh partitioning algorithms and the minimum edge cut criterion: Impact on realistic applications. In et. al David Baile, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 611–614, San Francisco, CA, 1995. SIAM.

[25] Chris Walshaw and Mark Cross. Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.