Using Domain Decomposition to find Graph Bisectors

Cleve Ashcraft* Joseph W.H. Liu†

November 8, 1995

Abstract

In this paper we introduce a three-step approach to find a vertex bisector of a graph. The first step finds a *domain decomposition* of the graph, a set of connected subgraphs, the *domains*, and a *multisector*, the remaining vertices that separate the domains from each other. The second step uses a block variant of the Kernighan-Lin scheme to find a bisector that is a subset of the multisector. The third step improves the bisector by bipartite graph matching. Experimental results show this domain decomposition method finds graph partitions that compare favorably with a state-of-the-art multilevel partitioning scheme in both quality and execution time.

1 Introduction

Graph partitioning is a well-known practical problem that has many important applications, such as task allocation for parallel computations [13] and circuit partitioning for VLSI design [22]. Our driving interest is to find low-fill orderings for sparse matrix computation [4], [6], [15], [19].

An effective approach to find fill-reducing orderings is nested dissection [8]. Based on a divide-and-conquer paradigm, it is a *recursive bisection* method, where the graph associated with the given sparse matrix is decomposed into two roughly equal halves by removing a subset of vertices (called the separator). The graph is reordered so that the vertices in each half are numbered contiguously and the vertices in the separator are numbered last. The reordered matrix is bordered block diagonal and the zero off-diagonal blocks will be preserved after factorization. The dissection can be applied recursively to each subgraphs. The success of such reorderings depends crucially on finding a small separator set in each subgraph.

1.1 Algorithmic Approaches

Finding a partition of a general graph that satisfies some size constraints is an NP-hard problem [5], and so practical partitioning algorithms are heuristic in nature. Graph partitioning methods can be broadly classified into two categories: a *direct* approach constructs a partition while an *iterative* approach improves a partition. Direct approaches include level set techniques (used in the automatic nested dissection algorithm [9]) and the spectral methods [21], [23]. The classic Kernighan-Lin algorithm [17] and its variants [6], [7], use an iterative approach by exchanging vertices of an existing partition. The

*Boeing Information and Support Services, P. O. Box 24346, Mail Stop 7L-22, Seattle, Washington 98124.
This research was supported in part by the ARPA Contract DABT63-95-C-0122.

†Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant A5509.

algorithm in [19] is an iterative scheme that improves a given partitioning with bipartite graph matching.

Many recent partitioning methods [6], [12], [14], [16] make use the idea of *blocking* to reduce the complexity of finding an improved partition. Typically, they first construct a coarse graph from the original graph, where each vertex in the coarse graph is usually a connected subset of vertices in the original one. A bisector of this smaller coarse graph is found, then projected back to the original graph and (optionally) improved.

Often these three phases of coarsening, partitioning and uncoarsening are applied recursively to a hierarchy or multiple levels of coarse/fine graphs. Hence, they have been referred to as *multilevel methods*. These multilevel methods [6], [12], [14], [16] find a bisector on the coarse graph by recursion until the size of the coarse graph is small enough that a constructive method can find a good bisector. These methods differ in the way the coarsening, partitioning, and uncoarsening phases are implemented. The `CHACO` [12] and `METIS` [16] software packages provide a variety of methods for these three phases.

Multilevel methods have a strong similarity to the multigrid method for solving PDE's. There is usually a smooth transition between a fine and coarse graph; the latter has around half the vertices of the former. The projection of the coarse graph's bisector is usually close to a good bisector in the fine graph, so relatively simple improvement methods are adequate in the uncoarsening step.

In this paper we propose a different approach, analogous to the *domain decomposition* methods for solving PDE's, the main competitor for multigrid. Instead of multiple levels we use a *two-level* approach. Our coarse graph consists of *domains* (large connected subsets of vertices) and the *multisector* (the remaining vertices that separate the domains from each other). The entire scheme consists of three major steps:

Step 1: Construct a domain decomposition of the graph.

Step 2: Construct a bisector (a subset of the multisector) using a block Kernighan-Lin scheme from Section 5.

Step 3: Improve the bisector using a graph matching scheme from Section 6.

We use a very simple method to construct the domain decomposition; see [4],[11] for more sophisticated algorithms. Since our coarse graph usually has fewer than a couple of hundred domains, we are able to construct a bisector on this coarse graph using a block variant of the Kernighan-Lin algorithm. We then improve the bisector on the fine graph using a powerful graph matching algorithm.

For solving PDE's, multigrid is generally felt to be more efficient than domain decomposition for smooth operators and regular discretizations. In the presence of anisotropies, i.e., non-homogeneities in the operator and/or the geometry, domain decomposition usually proves to be more robust. We have observed the same relative behavior between a multilevel method and the domain decomposition method we introduce in this paper. For relatively homogeneous graphs, a multilevel method finds better partitions; in the presence of irregularities, the domain decomposition method is better.

1.2 An Outline of the Paper

In Section 2 we give the background material in the graph partitioning problem. We define the problem in terms of unit-weight and weighted graphs. The notion of a partition is formalized and different evaluation functions to compare partitions are described. In Section 3, we present a generic version of the classic Kernighan-Lin scheme. A *move* operation

defines a partition transformation. A number of existing schemes in the literature that are variants of the Kernighan-Lin algorithm are discussed relative to our generic version.

In Section 4, we first formally introduce domain decomposition (domains, multisectors and segments) and then describe a simple algorithm to find a domain decomposition. Section 5 contains the description of the block Kernighan-Lin scheme based on the generic algorithm in Section 3. An example is used to illustrate the block scheme. Section 6 reviews the use of bipartite graph matching in improving a given bisector.

We have developed `DDSEP`, a graph partitioning code based on domain decomposition, block Kernighan-Lin and graph matching. In Section 7 we compare `DDSEP` with `METIS`, a state-of-the-art multilevel code from the University of Minnesota [16]. For homogeneous graphs, `METIS` creates extremely well-balanced partitions with small bisectors, while `DDSEP` produces acceptable partitions. However, for graphs with irregular geometry or graphs with high variability in degrees, we find that `DDSEP` produces better partitions. The CPU time required to find the partitions are comparable between `METIS` and `DDSEP`. Section 8 contains our concluding remarks.

2 Background

Let $G = (V, E)$ be a given undirected graph. Without loss of generality, assume the graph is connected. A vertex subset S is a *vertex separator* if the subgraph induced by the vertices in V but not in S has more than one connected component. An *edge separator* is a set of edges whose removal disconnects the graph. A separator is *minimal* if no subset of it forms a separator. A *bisector* is a separator whose removal gives two portions; a *multisector* is a separator that subdivides the graph into two or more portions. Although we want to find good partitions with vertex separators, in this and the next section, we have overloaded the use of S to mean either a vertex separator or an edge separator.

Many applications, including structural analysis and computational fluid mechanics, give rise to *weighted* graphs. In the underlying physical problem there might be several *degrees of freedom* associated with a location in space. Each degree of freedom can be considered as a vertex in the graph. Very often all these degrees of freedom will have the same adjacency structure in the graph, so for our purposes we can treat all the degrees of freedom associated with the same location as a single weighted vertex in a *compressed graph*, where the weight of the vertex is the number of its degrees of freedom. In practice, significant amount of time and space can be saved using such weighted graphs [2]. We shall encounter a problem in Section 7.2 where the original graph has 30237 vertices and its compressed weighted graph has only 6611 vertices. Using the weighted graph we find a graph bisector six times faster than we do by working on the original one.

For a weighted graph, each vertex has a positive weight $wt(v)$. Any subset $U \subseteq V$ has a weight, namely $|U| \equiv \sum_{u \in U} wt(u)$. For a unit-weight graph, each vertex has a weight of $wt(v) = 1$ and $|U|$ is just the cardinality of U.

We shall use the notation $[S, B, W]$ to represent a 2-set partition, where the removal of the bisector S will give two disconnected portions B and W. Vertices in B have the color black, and W with color white. One way to measure the *imbalance* of a partition is the quantity $||B| - |W||$, another is the ratio $\max\{|B|, |W|\}/\min\{|B|, |W|\}$. A partition with perfect balance has $|B| = |W|$.

2.1 An Evaluation Function for Partition Comparison

Intuitively, a good 2-set partition $[S, B, W]$ is one with a small separator size $|S|$, and two roughly equal portions $|B| \approx |W|$. Given an initial partition, we iteratively improve it until satisfied. We need to be able to compare the quality of the modified and the original partitions. To this end, we introduce an evaluation function and provide some justification for its use.

Consider a graph G with a 2-set partition $[S, B, W]$. The literature contains a number of different evaluation functions. We shall describe some of them here.

- One common evaluation function is based on the separator size $|S|$ subject to some size constraints on the two portions B and W. The evaluation function is:

$$cost_1[S, B, W] = \begin{cases} |S| & \text{if } \max\{|B|, |W|\} < \beta n \\ \infty & \text{otherwise,} \end{cases}$$

  where β is some fraction such as $2/3$. Evaluation functions similar to $cost_1$ are found in [12], [16], [18]. In practice, $cost_1[S, B, W]$ performs well but it has an undesirable discontinuity around partitions close to the size constraint.

- An evaluation function can measure the *distance* from a given partition to the ideal partition. An ideal partition would have $|S| = 0$, and imbalance $|B| - |W| = 0$. We have used a weighted 2-norm metric in [3]

$$cost_2[S, B, W] = \beta|S|^2 + (1 - \beta)(|B| - |W|)^2,$$

  where β is some fraction between 0 and 1. There are analogous 1-norm and ∞-norm cost functions. A disadvantage is that $cost_2[S, B, W]$ takes the weighted average of two quantities that are often of different orders of magnitude in sizes.

- In our present experiments we use a different evaluation function. The separator size $|S|$ is the primary metric, but the imbalance also has an influence. We use the "dimensionless" ratio $\max\{|B|, |W|\}/\min\{|B|, |W|\}$ to measure imbalance, where the perfect value would be 1. The imbalance enters as a "penalty" multiplicative factor, namely

$$\gamma[S, B, W] = |S| \left(1 + \alpha \frac{\max\{|B|, |W|\}}{\min\{|B|, |W|\}}\right),$$

  where α is some constant greater than 0. A large value of α places a large emphasis on the balance. In practice, setting α to 1 generally results in good partitions.

All these evaluation functions can be used either with an edge separator $S \subset E$ or a vertex separator $S \subset V$. In the end, any evaluation of a partition is subjective, based on the reader's sense of separator weight and imbalance. We have experimented with the 1-norm, 2-norm and the penalty cost functions. The first two suffer from a strong sensitivity to the β parameter and an inability to allow a move from a well-balanced partition to one with smaller separator weight and moderate imbalance. We have used the penalty cost function $\gamma[S, B, W]$ with $\alpha = 1$ in all the experiments in Section 7. This value for α puts more emphasis on separator weight but still allows a move from a well-balanced partition if the separator weight will decrease.

3 The Generic Kernighan-Lin Improvement Algorithm

Many practical heuristic partitioning methods are variants of the scheme by Kernighan and Lin [17]. Our block scheme described in Section 5 is another. The original Kernighan-Lin algorithm tries to find a small edge separator by executing a sequence of vertex-pair exchanges between two portions of an initial partition. We now present a generic version of the Kernighan-Lin scheme for a 2-set partition by an edge or vertex separator.

The generic partition improvement scheme has two nested loops. The outer loop is given in Figure 1. Inside this loop, the partition improvement function GKL-IMPROVE is called until no improvement to the partition can be made.

```
GENERIC-KL
    Initialize a partition [S, B, W]
    repeat
        [S*, B*, W*] = [S, B, W]
        [S, B, W] = GKL-IMPROVE [S*, B*, W*]
    until [S*, B*, W*] = [S, B, W]
return [S, B, W]
```

FIG. 1. GENERIC-KL: *Generic Kernighan-Lin Scheme.*

The basic operation is a *move* which we now define. Let Q be either B or W and let $Z \subset V$. The move of Z to the portion Q, written as $Z \mapsto Q$, merges Z and Q. We sometimes refer to Q as the *destination* of Z in this move. Implicit is the assumption that $Z \cap Q = \emptyset$. Moving Z to Q will induce other changes to the partition. We shall denote the new partition as a result of this move by:

$$[S, B, W]_{Z \mapsto Q} \equiv [S_{Z \mapsto Q}, B_{Z \mapsto Q}, W_{Z \mapsto Q}].$$

The function GKL-IMPROVE is given in Figure 2. The inner loop makes a sequence of moves until there are no more movable objects left to be moved. In effect, it generates a sequence of partitions:

$$[S_0, B_0, W_0] = [S, B, W] \longrightarrow [S_1, B_1, W_1] \longrightarrow ... \longrightarrow [S_m, B_m, W_m].$$

The best partition $[\widehat{S}, \widehat{B}, \widehat{W}]$ in this sequence is returned by this function.

To define any variant of this generic algorithm, we need to answer three questions:

- What are the movable objects?
- How to select a move?
- When is one partition better than another?

Examples are provided later in this section. In this paper, we use the penalty function $\gamma[S, B, W]$ as our evaluation function.

The most time consuming step is the selection of an unmarked movable object and its destination for the next move. To improve efficiency, often some eligibility conditions for selection are imposed on unmarked objects. This limits the size of the search space to select an eligible object for the next move.

3.1 Edge Separators: Kernighan-Lin and Fiduccia-Mattheyses

Both these methods are designed to find small edge separators. They usually use an initial partition $[S, B, W]$ where $|B| \approx |W| \approx n/2$.

```
GKL-IMPROVE [S, B, W]
    Unmark all movable objects
    [Ŝ, B̂, Ŵ] = [S, B, W]
    while there are unmarked movable objects left do
        for all unmarked objects and possible destinations do
            select a move: an object Y with destination Q
        [S, B, W] = [S, B, W]_{Y↦Q}
        mark Y
        if [S, B, W] is a better partition than [Ŝ, B̂, Ŵ]
            then [Ŝ, B̂, Ŵ] = [S, B, W]
    end while
return [Ŝ, B̂, Ŵ]
```

FIG. 2. GKL-IMPROVE: *Generic Kernighan-Lin Improvement Scheme.*

3.1.1 The Kernighan-Lin Scheme. A move is based on a pair of vertex exchanges. Consider a pair of vertices $b \in B$ and $w \in W$. The exchange of this pair of vertices can be expressed as the move operation: $(b, w) \mapsto (W, B)$. It modifies the partition as follows:

$B_{(b,w)\mapsto(W,B)} = B \cup \{w\} \setminus \{b\}$
$W_{(b,w)\mapsto(W,B)} = W \cup \{b\} \setminus \{w\}$

and $S_{(b,w)\mapsto(W,B)}$ is the edge separator determined by the two new portions.

In terms of the generic method, the Kernighan-Lin algorithm has:

- Movable objects: a pair of vertices (b, w) with $b \in B$ and $w \in W$ and the move operation $(b, w) \mapsto (W, B)$.
- Move selection: find a pair of unmarked vertices $b \in B$ and $w \in W$ that maximizes the reduction in separator size: $|S| - |S_{(b,w)\mapsto(W,B)}|$.
- A partition $[S^*, B^*, W^*]$ is better than $[S, B, W]$ if $|S^*| < |S|$.

3.1.2 The Fiduccia-Mattheyses Scheme. Fiduccia and Mattheyses [7] made some practical improvements to the basic Kernighan-Lin algorithm. Instead of exchanging a pair of vertices, a single vertex is moved at a time. This helps to reduce the time required to select the next move. In terms of the generic method, Fiduccia-Mattheyses has:

- Movable objects: a vertex x and the move operation $x \mapsto Q$, where $x \notin Q$ ($Q = B$ or W).
- Move selection: find an unmarked vertex x from the larger portion that maximizes the reduction in separator size ($x \notin Q$): $|S| - |S_{x\mapsto Q}|$.
- A partition $[S^*, B^*, W^*]$ is better than $[S, B, W]$ if $|S^*| < |S|$.

3.2 Vertex Separators: Using Primitive and Composite Moves

In [3], improvement schemes are considered for partitions with vertex separators. The basic operation is the move of a subset of the separator into either B or W. Consider the move $Z \mapsto W$ of a subset Z of S to the portion W. While W absorbs Z, there are also changes to B and S:

$$W_{Z\mapsto W} \quad = \quad W \cup Z,$$

$$\begin{aligned} B_{Z\mapsto W} &= B \setminus Adj(Z), \\ S_{Z\mapsto W} &= V \setminus (B_{Z\mapsto W} \cup W_{Z\mapsto W}) = (S \setminus Z) \cup \{Adj(Z) \setminus W\}. \end{aligned}$$

In [3], this operation has been referred to as a *composite move*. When Z is a single vertex, it is called a *primitive move*.

In terms of the generic method, composite move methods have:

- Movable objects: a subset $Z \subseteq S$ and the move operation $Z \mapsto Q$ where $Q = B$ or W.
- Move selection: find a subset $Z \subseteq S$ that maximizes the reduction in separator size: $|S| - |S_{Z\mapsto Q}|$.
- A partition $[S^*, B^*, W^*]$ is better than $[S, B, W]$ if $\gamma[S^*, B^*, W^*] < \gamma[S, B, W]$.

4 Finding a Domain Decomposition of a Graph

Domain decomposition is a common approach in the solution of partial differential equations in the area of scientific computation. To put it in our context, for an undirected graph $G = (V, E)$, consider a partition of the vertex set V:

$$V = \Phi \cup \Omega_1 \cup \Omega_2 \cup \ldots \cup \Omega_d,$$

where each Ω_i is a *domain* and Φ is the set of *interface* vertices. Each domain Ω_i is a connected subgraph of G with its *boundary* $Adj(\Omega_i)$ contained in the interface set Φ. No two domains are adjacent, they are separated from one another by interface vertices. The set Φ is a *multisector*, for it generalizes the notion of a bisector. Without loss of generality, we shall assume that the multisector partition is nontrivial; that is, $d \geq 2$ and Φ is nonempty.

Consider the 6×6 grid graph in Figure 3. The vertices are partitioned into six domains and a multisector $\Phi = \{2, 4, 6, 7, 8, 10, 14, 16, 18, 19, 20, 21, 22, 23, 27, 33\}$. Multisector vertices are represented by squares, domain vertices by circles.

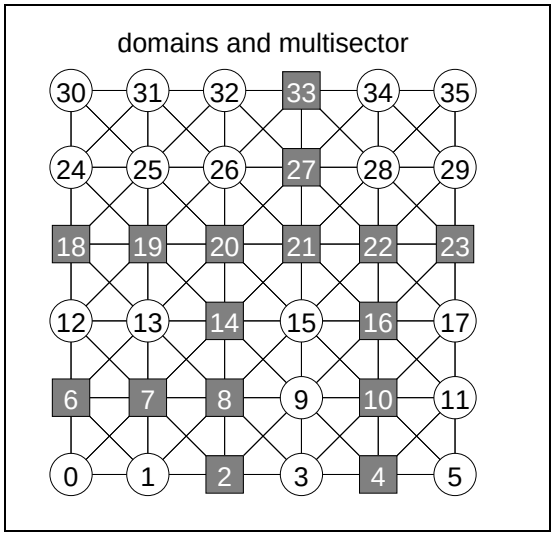


FIG. 3. *Grid example of domains and multisector.*

4.1 Constructing a Domain Decomposition

There are many different ways to find a multisector and its domain partitioning. One could use some geometric knowledge (such as the locations of mesh points) or some other substructuring information of the given graph to determine domains. Recent work by Goehring and Saad [11] determines a domain decomposition based on special set of vertices called *centers*. Another approach uses a minimum degree ordering [10] of the graph. As a byproduct of the ordering process, the minimum degree ordering creates a natural tree structure of the vertices. To each subtree is associated a connected subgraph, and by design, the subgraph should have a small adjacent set. It is natural to take a number of disjoint subtrees to define the domains and the remaining vertices form the multisector. This approach generally gives very effective multisectors. We have used this approach in [3], but the process is relatively expensive for the execution time is dominated by the time to generate a minimum degree ordering.

We want to find a reasonably effective multisector efficiently. Let ω_{min} and ω_{max} be the desired minimum and maximum bounds on domain weights. We construct the multisector in the following way:

Step 1: Initialize the multisector

Initialize the multisector Φ to include all vertices of degrees greater than some multiple of the median degree.

Step 2: Grow the Domains

Let G' be the subgraph containing vertices outside the current multisector and domains. Choose a random vertex x from G'. Perform a breadth-first search in G' starting from x to grow a domain until its weight has reached ω_{max}. Add the adjacent vertices of this new domain to the multisector Φ. Repeat step 2 until no vertex is left.

Step 3: Absorb Small Domains into the Multisector

For each domain formed in step 2, if its weight is smaller than ω_{min}, add the vertices in this domain to the multisector.

Step 4: Absorb Excess Multisector Vertices into the Domains

Choose a multisector vertex that is adjacent to only one domain, and merge it to its neighboring domain. Priority is given to those with smaller degrees. Repeat step 4 as long as possible.

Step 1 is important for graphs with a large variability in degrees (see Section 7.3 for an example). Its effect is to exclude vertices of high degrees during the formation of the domains in step 2. In step 3, we absorb small domains into the multisector to reduce the number of domains. This helps to reduce the execution time of the block Kernighan-Lin partitioning scheme, since its complexity is largely a function of the number of domains. Step 4 generates a minimal multisector.

4.2 Find a Separator using the Domains and Multisector

The block Kernighan-Lin scheme to be described in the next section will determine a partition $[S, B, W]$ from a given domain decomposition $\{\Phi, \Omega_1, \ldots \Omega_d\}$. The partition obtained in this phase satisfies the following conditions: $S \subseteq \Phi$ and $\Omega_i \subseteq B$ or $\Omega_i \subseteq W$, for $i = 1, \ldots, d$. Note, usually $S \subset \Phi$, so the multisector is split among S, B and W. We find a

partition by coloring the domains and multisector vertices. We shall adopt the convention that separator vertices in S are colored gray, vertices in B black and vertices in W white.

In our block scheme, domains are our movable objects. Vertices of a domain are moved from component to component together instead of moving a single vertex. and so all vertices in a domain have the same color. Therefore, we can view the block scheme as an assignment of black and white colors to the domain sets.

How about the separator set? Our goal is to induce the colors of the multisector vertices from the given domain colors, and hence determine the separator set. A simple multisector coloring scheme colors each multisector vertex white, black, or gray depending on the color of its adjacent domains.

(*) If a vertex $v \in \Phi$ is adjacent to domain vertices of only one color C, then we color v with C; otherwise color it with gray.

Unfortunately, this simple coloring rule (*) may not always produce a separator set. A counterexample is shown in Figure 4. The grid on the left shows the domain interface partition: multisector vertices are squares colored light gray, domain vertices are circles with a given black/white black coloring, The grid on the right illustrates the coloring of the multisector vertices using the coloring rule (*). The gray colored vertices do *not* form a separator, for black vertex 15 is adjacent to white vertex 20. Note that we have used light gray to denote multisector vertices, and darker gray to indicate separator vertices.

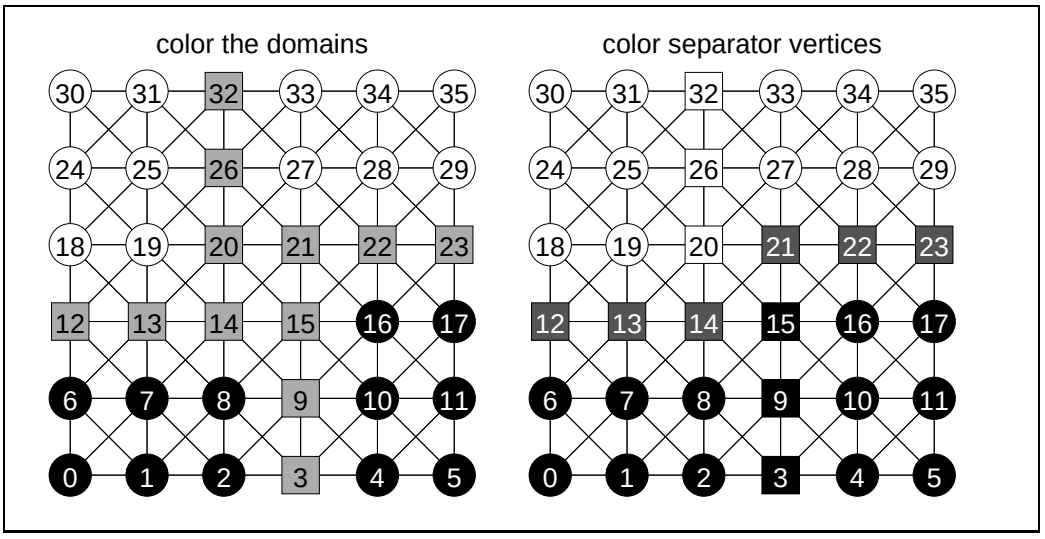


FIG. 4. *A situation where the coloring rule fails*

The simple coloring rule (*) fails to define a separator for this particular domain/interface partition. The reason is subtle, and is explained by the following theorem.

THEOREM 4.1. *For a given domain decomposition* $\{\Phi, \Omega_1, ..., \Omega_d\}$*, the coloring scheme* (*) *will generate a separator for all colorings of the domains if and only if every adjacent pair of multisector vertices that are adjacent to some domain has a common adjacent domain.*

Proof. "*If Part*": Assume for contradiction that the set of gray-colored multisector vertices does not form a separator. Then there must be an edge (x, y) where $x \in B$ and $y \in W$. We first show that x and y must both be in Φ. Assume that x is in a domain (colored black). If y were in a domain, it must be the same domain that contains x, so

y would be colored black, a contradiction. If y were in the multisector, then y must be colored either black or gray by (*), another contradiction. Therefore x must be in Φ. A similar argument shows that y is also in Φ. If x and y share a common domain, then each would be colored either gray or the same color as the domain, a contradiction. Therefore, x and y have no common adjacent domain.

"*Only If Part*": Let x and y be two adjacent multisector vertices that do not have a common adjacent domain. Then introduce the following coloring of the domains. Color all adjacent domains of x white and all adjacent domains of y black. Apply the multisector coloring scheme. The vertex x will be colored white and y black. This implies that the gray-colored multisector vertices will not form a separator. ■

Theorem 4.1 provides a necessary and sufficient condition for the multisector coloring scheme to produce a separator. For problems not satisfying this condition, the multisector coloring scheme will not work properly. Figure 4 illustrates the situation. Black vertex 15 is adjacent to white vertex 20, for these two adjacent vertices have no common adjacent domain.

Obviously we cannot depend on applying the coloring rule (*) to single vertices. We must use a coloring rule for subsets of the Φ vertices.

> (**) If a subset $\sigma \subseteq \Phi$ is adjacent to domains of only one color C, then color all vertices in σ with C; otherwise color gray.

There is no need to color a vertex more than once, so these sets can be disjoint; let $\Sigma = \{\sigma_1, \ldots, \sigma_r\}$ be a partition of Φ. We also need the following generalization of Theorem 4.1 to subsets of multisector vertices.

THEOREM 4.2. *For a given domain decomposition $\{\Phi, \Omega_1, \ldots, \Omega_d\}$ and a partition Σ of Φ, the coloring scheme (**) will generate a separator for all possible colorings of the domains if and only if σ_i is adjacent to σ_j implies there exists a domain that is adjacent to both σ_i and σ_j.*

Proof. Very similar to Theorem 4.1. ■

We are searching for some partition Σ of Φ that will satisfy the coloring rule (**) but contain as few σ sets as possible. It turns out that a simple two-step process will find such a partition.

4.3 Find the Maximal Segment Partition

The notion of blocking the multisector vertices into *segments*, first introduced in [1], is the key to speed up the multisector coloring scheme. Instead of repeatedly using (*) to evaluate the color of a single vertex, we use (**) to evaluate the color of a segment, a subset of Φ. We want to use the smallest number of segments as practical. Finding the maximal segment partition Σ is a two step process.

1. First we construct Ψ, a partition of Φ, such that Theorem 4.2 holds. If two adjacent multisector vertices have no common adjacent domains, they should belong to the same subset in Ψ. Formally, consider the graph $G(\Phi, E \cap (\Phi \times \Phi))$, the subgraph over Φ, and delete all edges (x, y) where x and y are adjacent to a common domain. If we define
$$E_\Phi = (E \cap (\Phi \times \Phi)) \setminus \bigcup_{i=1}^{m} (Adj(\Omega_i) \times Adj(\Omega_i)),$$
then the Ψ partition is simply the connected components in $G(\Phi, E_\Phi)$ and it satisfies Theorem 4.2.

2. Given a partition Ψ that satisfies Theorem 4.2, we find the maximal segment partition Σ as follows: if ψ_1 and ψ_2 in Ψ are adjacent to exactly the same sets of domains, then ψ_1, and ψ_2 belong to the same subset in Σ.

Note, the partition Ψ satisfies Theorem 4.2 by construction, and the segment partition Σ will still satisfy the conditions of Theorem 4.2. Furthermore, this partition is maximal in some sense, for by construction, no two segments are adjacent to exactly the same set of domains.

Consider the example in Figure 5. The left hand grid shows the edges in $E_\Phi = \{(14, 21), (15, 20)\}$. The first partition

$$\Psi = \{\{3\}, \{9\}, \{12\}, \{13\}, \{14, 21\}, \{15, 20\}, \{22\}, \{23\}\}$$

contains eight singleton vertices and $\{14, 21\}$ and $\{15, 20\}$. The right hand grid shows the segments partition

$$\Sigma = \{\{3, 9\}, \{12, 13\}, \{14, 15, 20, 21\}, \{22, 23\}, \{26, 32\}\}\,.$$

Vertices 26 and 32 form one segment since they are adjacent to the same two domains. Subsets $\{14, 21\}$ and $\{15, 20\}$ of Ψ are both adjacent to the same four domains, so they form a new segment $\{14, 15, 20, 21\}$. All adjacent segments have at least one common adjacent domain so the coloring rule `(**)` holds.

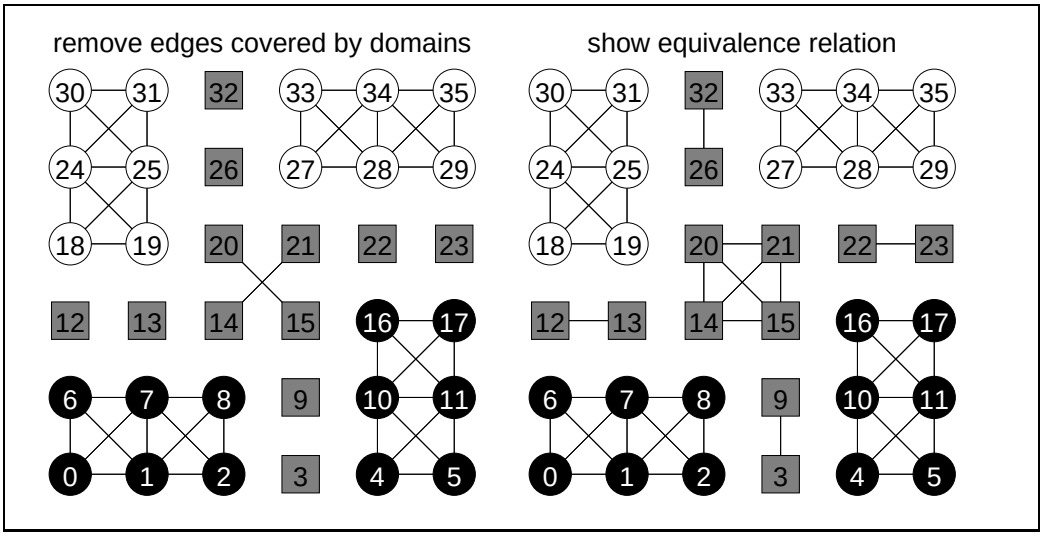


FIG. 5. *A two step process to define the segments*

5 The Block Kernighan-Lin Scheme

With the notions of domains and segments introduced, we are now ready to describe the block Kernighan-Lin scheme that we use to construct the initial bisector. Recall from Section 3, we need to specify the movable objects, the selection of the next move, and the evaluation function.

5.1 Movable Objects and Move Selection

Let $\{\Phi, \Omega_1, \ldots, \Omega_d\}$ be a domain decomposition of the graph and let $\Sigma = \{\sigma_1, \ldots, \sigma_s\}$ be the segment partition of the multisector Φ. Domains are colored black or white, and

segments are colored black, white or gray (the separator color) using the coloring rule (**) of Section 4.2. This will induce a partition $[S, B, W]$ of the graph, where B is the set of vertices in black domains and segments, W the set of vertices in white domains and segments, and S the set of multisector vertices in gray segments.

The block Kernighan-Lin scheme improves an initial partition based on the given domain decomposition of the graph. It is a variant of the generic algorithm of Section 3 where the movable objects are domains. For a domain Ω, the basic move operation is $\Omega \mapsto B$ or $\Omega \mapsto W$. The move *flips* the color of the domain. Furthermore, we use the standard technique where a domain is *eligible* to be moved if it has not yet been flipped during this inner iteration and if it is adjacent to the current bisector.

In terms of the generic method, our block Kernighan-Lin method has:

- Movable objects: All unflipped domains that are adjacent to the current bisector.
- Move selection: Select an unflipped domain Ω and its destination Q such that $\gamma[S, B, W]_{\Omega \mapsto Q}$ is minimum among the eligible moves.
- Partition comparison: We use the penalty function $\gamma[S, B, W]$.

We choose to select the best move in terms of the evaluation function among the movable domains. If d is the number of domains, it will take $O(d^2)$ work in the worst case to select the moves. This seems like an excessive amount of work, but in practice d is relatively small and we rarely have to make a selection from all d domains at once. Overall the block Kernighan-Lin method takes $5\% - 15\%$ of the total time to find a bisector. We can afford the expensive move selection process and can avoid the usual tricks to reduce the execution time, e.g., an early bailout of the inner loop after no improvement has been observed for a certain number of steps.

5.2 Implementation Details in the Block Scheme

The generic Kernighan-Lin scheme of Figure 1 starts with an initial partition, then calls GKL-Improve to iteratively improve the partition. In our implementation of the block method, we construct an initial partition based on a level structure of the domains. Domains in the first levels are colored black until the black domains exceed half the weight of all the domains.

Another important detail is the evaluation of $\gamma[S, B, W]_{\Omega \mapsto Q}$ for each unflipped domain Ω for move selection. Let Ω be an unflipped domain in B. Consider the evaluation of the function value $\gamma[S, B, W]_{\Omega \mapsto W}$. The new partition after the move differs from $[S, B, W]$ only locally; Ω has moved from B to W and some of its adjacent segments may change color. To study such local change, we define:

$$\Delta|S| = |S_{\Omega \mapsto W}| - |S|, \ \Delta|B| = |B_{\Omega \mapsto W}| - |B|, \text{ and } \Delta|W| = |W_{\Omega \mapsto W}| - |W|.$$

These terms are the changes in the respective sizes of the three subsets in the partition if the move $\Omega \mapsto W$ were to be performed. Note also that these quantities can be positive or negative.

Figure 6 shows how we compute the Δ values for the move $\Omega \mapsto W$. Evaluating a move from W to B is similar. For each unflipped domain Ω and its destination Q, we compute the three values $\Delta|S|$, $\Delta|B|$ and $\Delta|W|$. that measure the changes in the partition weights if the domain should flip its color. Then the function value $\gamma[S, B, W]_{\Omega \mapsto Q}$ can be computed using the new partition weights $|S| + \Delta|S|$, $|B| + \Delta|B|$ and $|W| + \Delta|W|$.

FIG. 6. *Evaluate the move* $\Omega \mapsto W$

```
EVAL(Ω ↦ W)
    Δ|S| = 0, Δ|B| = −|Ω|, Δ|W| = |Ω|
    for each segment σ adjacent to Ω do
        if σ is colored black then
            σ will move into S, Δ|S| = Δ|S| + |σ|, Δ|B| = Δ|B| − |σ|
        else if Ω is the only black domain adjacent to σ then
            σ will move into W, Δ|S| = Δ|S| − |σ|, Δ|W| = Δ|W| + |σ|
        end if
    end for
```

The three Δ values are quantities local to its domain Ω. They remain unchanged until a move is made of a domain $\tilde{\Omega}$ that shares a common adjacent segment with Ω. Only at this time must the Δ values of Ω be updated. We add one line to the generic scheme GKL-IMPROVE of Figure 2; after a move of a domain is made, we update the three Δ values of all unflipped domains that share a boundary with the domain.

5.3 A Grid Example

We use the graph in Figure 3 as an example to illustrate the execution of the block scheme. The six domains and eleven segments are given below.

$$\begin{aligned}
\Omega_1 &= \{0, 1\} \\
\Omega_2 &= \{3, 9, 15\} \\
\Omega_3 &= \{5, 11, 17\} \\
\Omega_4 &= \{12, 13\} \\
\Omega_5 &= \{24, 25, 26, 30, 31, 32\} \\
\Omega_6 &= \{28, 29, 34, 35\} \\
\Sigma &= \{\{2\}, \{6, 7\}, \{8\}, \{4, 10, 16\}, \{14\}, \{18, 19\}, \{20\}, \{21\}, \{22\}, \{23\}, \{27, 33\}\}
\end{aligned}$$

The algorithm begins with the partition $[S, B, W] = [\emptyset, V, \emptyset]$. We use the parameter value of $\alpha = 1$ for the penalty function, and the cost of this initial partition is set at infinity.

In Figure 7, we illustrate the coloring process in the execution of GKL-IMPROVE. Initially, all domains are colored black. In the first step, we evaluate what would happen if each domain is flipped. Of all the domains, the best evaluation function is obtained if we flip Ω_5. The new evaluation function value is 30. In the second step, flipping the domain Ω_6 results in the best evaluation function value of 15. The process is repeated until no more domains are left unflipped in this round. The function value of the partition is found above each grid. The algorithm compares the best cost (15) obtained from this round with the old cost (infinity) and an improvement is detected. The initial partition is then replaced by the improved partition with the new cost.

The block Kernighan-Lin algorithm again calls the improvement scheme GKL-IMPROVE, but uses the new partition as its initial partition in the second round. Figure 8 gives the detailed illustration of the coloring process of the domain and multisector vertices. In this round, a better cost value of 14.5 is encountered, and this better partition will be returned. The next call to GKL-IMPROVE does not find a better partition, and so the

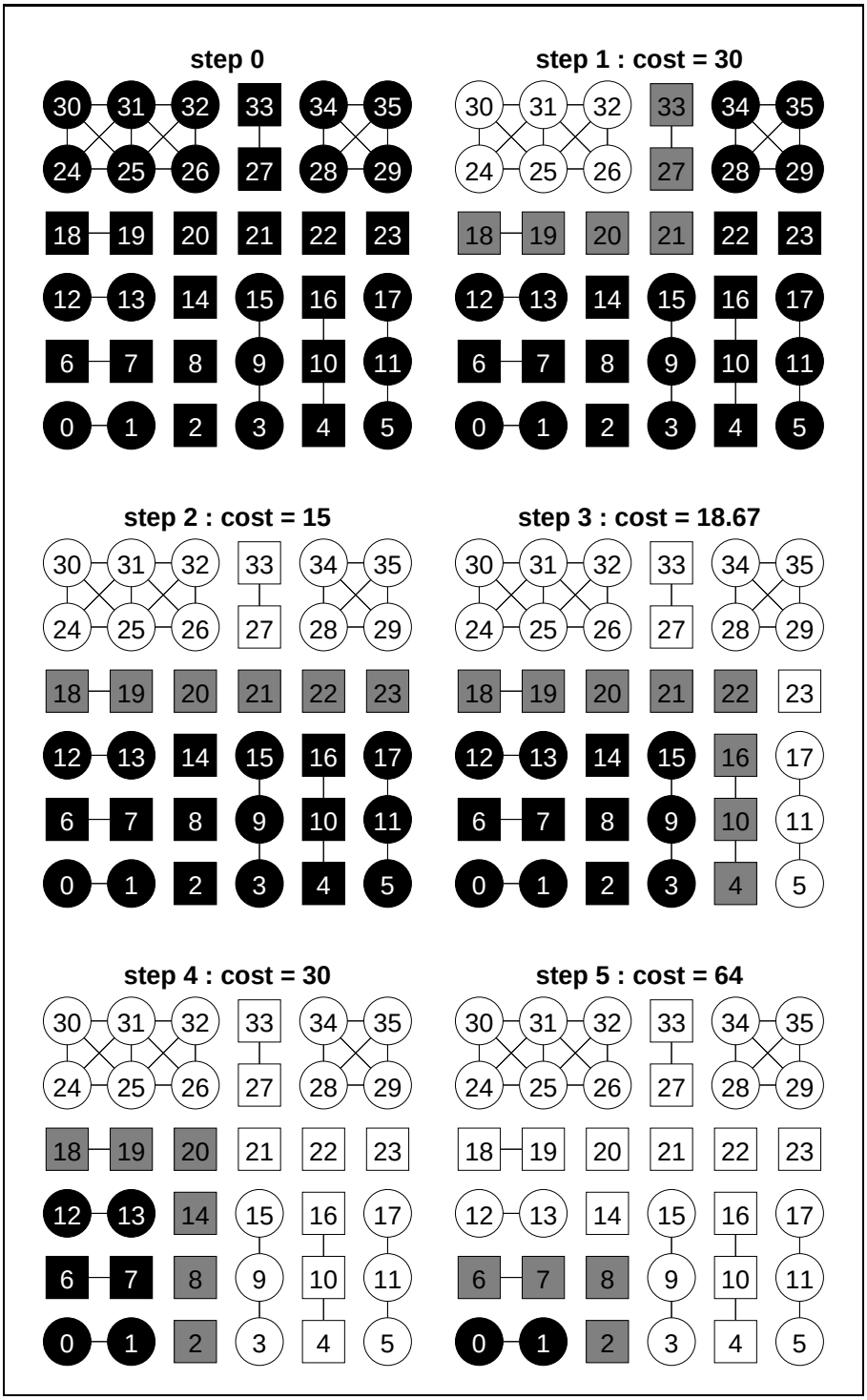


FIG. 7. *First Execution of* GKL-IMPROVE.

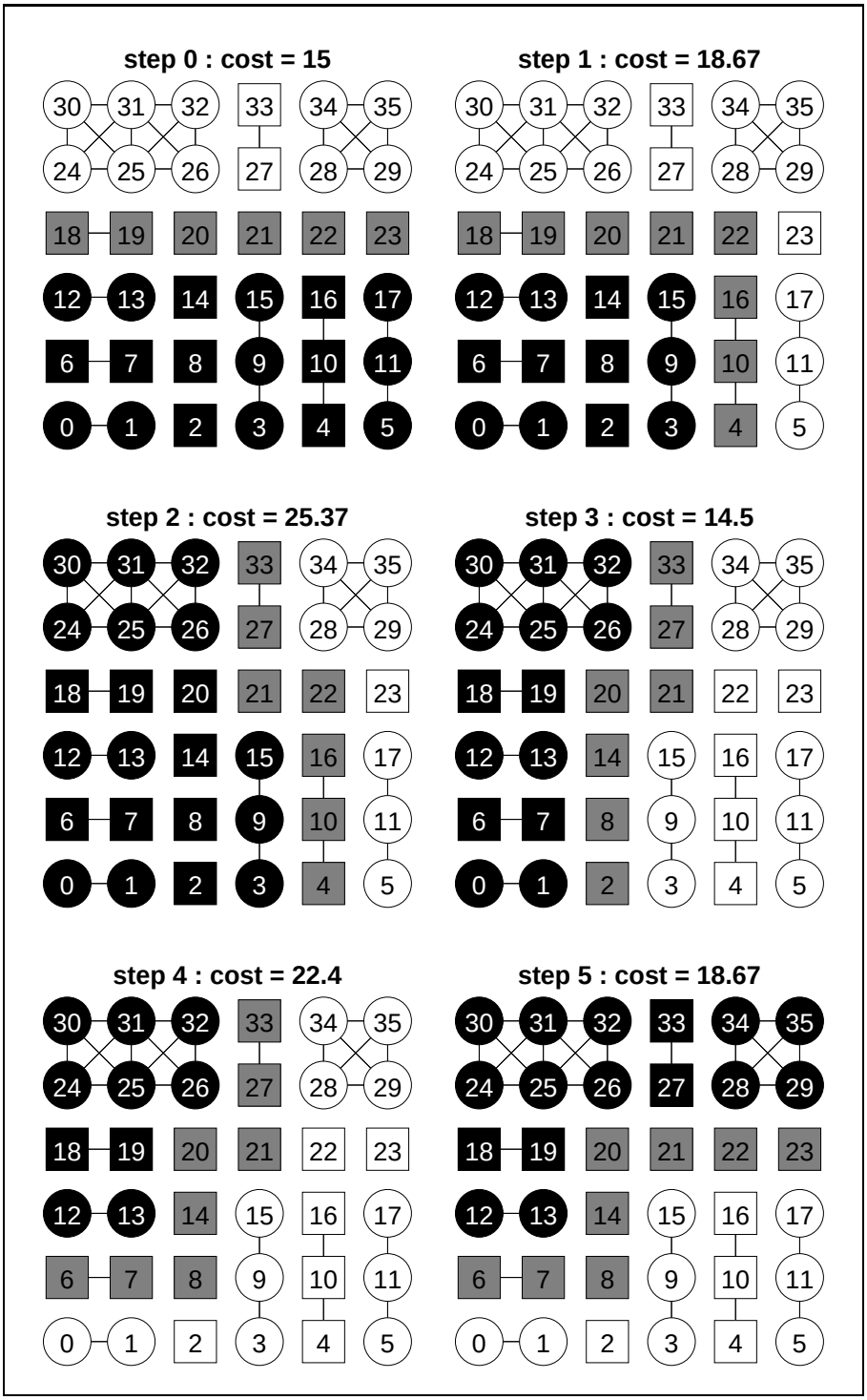


FIG. 8. *Second Execution of* GKL-IMPROVE.

method returns the partition with cost value 14.5.

6 Partition Improvement by Bipartite Graph Matching

After the first two steps of the partitioning scheme, a bisector is found that is a subset of the initial multisector. Because of the block nature of domains and segments, it is to be expected that the bisector can be improved. We call this third and final step the "bisector smoothing" step.

Consider a given partition $[S, B, W]$. Let Z be a subset of the bisector S such that

$$|Adj(Z) \cap B| < |Z|,$$

that is, the weight of Z is larger than that of its adjacent set in B. We then consider the *move* operation that moves the subset Z from S to W, and replaces Z by $Adj(Z) \cap B$ in S. More formally, as before let $[Z, B, W]_{Z \mapsto W}$ be the partition after this move operation $Z \mapsto W$ is performed, we have:

$$B_{Z \mapsto W} = B \setminus Adj(Z),\ W_{Z \mapsto W} = W \cup Z, \text{ and } S_{Z \mapsto W} = (S \setminus Z) \cup (Adj(Z) \cap B).$$

Note that this move operation is different from the move operations used in the block Kernighan-Lin scheme in Section 5.

It should be obvious that the new bisector $S_{Z \mapsto W}$ has smaller weight than that of S since $|Adj(Z) \cap B| < |Z|$. On the other hand, subsets Z where $|Adj(Z) \cap B| = |Z|$ can also be useful. If $|B| > |W|$, then moving such a set Z to W will not decrease the separator weight but it may improve the balance of the partition. Our improvement scheme is based on finding and moving such bisector subsets.

```
SEP-IMPROVE [S, B, W]
    Improved = true
    while Improved do
        if |B| < |W| then interchange B and W // make B the larger portion
        if a subset Z of S is found with |Adj(Z) ∩ B| ≤ |Z|
            and γ[S, B, W]_{Z↦W} < γ[S, B, W] then
                [S, B, W] = [S, B, W]_{Z↦W}
        else
            if a subset Z of S is found with |Adj(Z) ∩ W| < |Z|
                and γ[S, B, W]_{Z↦B} < γ[S, B, W] then
                    [S, B, W] = [S, B, W]_{Z↦B}
            else
                Improved = false
            end if
        end if
    end while
```

FIG. 9. *Partition Improvement Scheme.*

Figure 9 contains a high-level description of the improvement algorithm. The method first attempts to improve the partition by reducing the size of the larger portion B. If no such subset/move can be found, the algorithm tries to improve the partition by moving a

subset of the separator into the larger portion B. In both cases, a move is made only if it improves the partition in terms of the evaluation function. The process continues until no reduction can be found. Central to the algorithm is the determination of a subset Z of S such that $|Adj(Z) \cap Q| \leq |Z|$ where $Q = B$ or W.

In [19], the technique of bipartite graph matching is used to find such bisector subsets Z. We have improved this technique to find larger subsets to move (and thus reduce the number of steps and the execution time). We use the Dulmage-Mendelsohn decomposition [20] to find the Z sets. Although the Dulmage-Mendelsohn decomposition is defined only for unit-weight graphs, we are able to work with the weighted graph, thus greatly reducing the execution time. The details in the use of matching and the Dulmage-Mendelsohn decomposition are beyond the scope of this paper. Readers are referred to [19], [20] and a future paper by the authors for such details.

Figure 10 contains an example to illustrate a bisector smoothing step. On the left we see that the subset $Z = \{7, 13, 19, 25\}$ of S has an adjacent set in B of $Adj(Z) \cap B = \{14, 20\}$. The size of its adjacent set is 2 which is smaller than $|Z| = 4$. On the right we see the new partition obtained by adding the set Z to W, removing $\{14, 20\}$ from B, and replacing Z by $\{14, 20\}$ in S. The new bisector now has size two less than before. It should be noted that there are different subsets of S that satisfy the condition on its adjacent set size. Moving $\{7, 13, 19\}$ or $\{7, 19, 25\}$ both decrease the separator size by one.

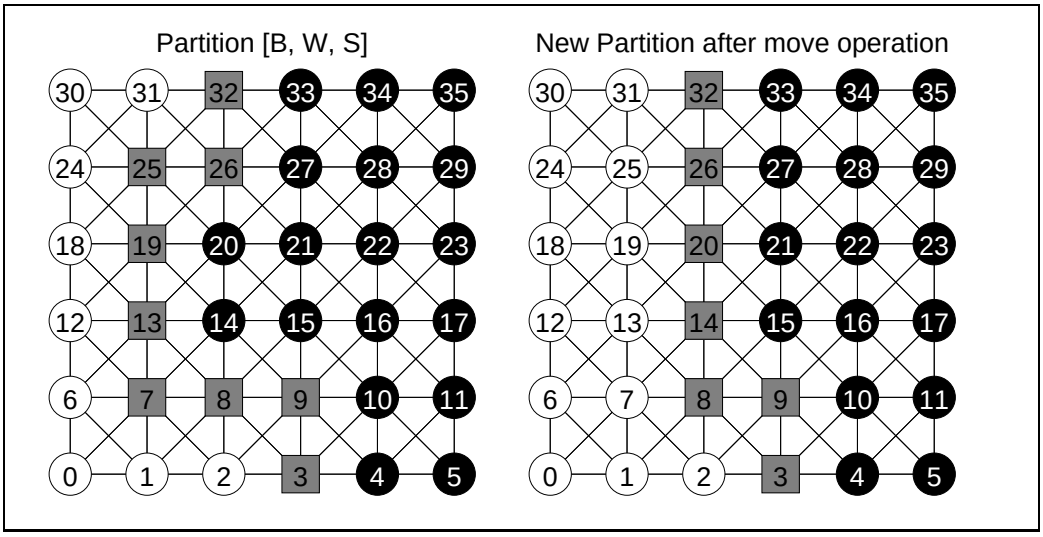


FIG. 10. *Grid example of bisector smoothing.*

7 Experimental Results

In this section we describe some experiments with our prototype domain decomposition code `DDSEP` and a state-of-the-art multilevel code `METIS` available from the University of Minnesota [16]. Neither code is consistently better than the other. Each has strengths and weaknesses that we will illustrate.[1]

[1] The options we used for `METIS` were recommended to us by the author, George Karypis, namely SHEM (sorted heavy edge), BGKLR (combination of boundary greedy and boundary Kernighan-Lin) and GGPKL (graph growing followed by boundary Kernighan-Lin). `DDSEP` used $\omega_{\min} = 40$, $\omega_{\max} = 80$ and $\alpha = 1$.

We have experimented with the graphs of many sparse matrices. However, to illustrate our preliminary findings, we only present the partition results for three graphs.

- In Section 7.1 we examine R2D7905, an unstructured but relatively homogeneous graph. `METIS` consistently finds extremely well-balanced partitions with small bisectors while `DDSEP` finds partitions with small bisectors and an acceptable amount of imbalance.

- In Section 7.2 we study BCSSTK35 which displays a number of "local minima" partitions where most do not lie near perfect balance. `METIS` finds extremely well-balanced partitions at the expense of larger bisectors. `DDSEP` consistently returns partitions with smaller separators and fairly significant but acceptable imbalance.

- In Section 7.3 we look at BCSSTK08 which has a number of vertices that have extremely high degree. For reasons that we do not completely understand, this causes problems for both `METIS` and `DDSEP`.

Most algorithms to partition a graph have some sensitivity to the order vertices are visited or an adjacency list is traversed. In many cases, the quality of the resulting partition will show a large variability. To study such variability, our prototype `DDSEP` code uses a random number generator internally to shuffle vertex lists and find seed vertices for a traversal; `METIS` has a similar capability. For BCSSTK35, due to randomization we have observed a factor of six difference in separator weights from `DDSEP` and a factor of three difference for `METIS`. On the other hand, some graphs show little variability; `DDSEP` and `METIS` consistently return the optimal bisector for 2-D and 3-D regular grids.

We recommend that for any graph, particularly if it is unstructured, several partitions be created and the best used. "Several" is hard to quantify. If k runs have been made, there is one chance in $(k+1)$ that a better partition will be found by making another run, but *how much better* it is not possible to say. Multilevel and domain decomposition methods have so dramatically reduced the times to partition a graph that it is easy to amortize multiple runs when the partition variability is large. We generally advise two or three runs, then choose the best partition.

For a given graph and method, what partition can we expect? There are many ways to answer, and we present a very simple approach. For each graph we have made 100 runs with a particular method and collected the following statistics:

- separator weight $= |S|$,

- imbalance $= \max\{|B|, |W|\} / \min\{|B|, |W|\}$,

- cost $= |S|(1 + \alpha \max\{|B|, |W|\} / \min\{|B|, |W|\})$, and

- elapsed CPU time (not counting any I/O).

We have used $\alpha = 1$ in our experiments, and since imbalance is usually close to one in value, cost is almost linear in $|S|$. CPU time usually shows little variation, for the bulk of the time of both `METIS` and `DDSEP` is devoted to creating the coarse graph(s). We study the data in two ways.

- We plot the $(|S|, \text{imbalance})$ data points as circles in a *weighted* scatter plot. (See Figures 11, 12 and 13.) Some partitions occur many times in the one hundred runs we made, so to represent their multiplicity we use larger circles. The area of a circle is proportional to the number of times the partition occurred. The center of the circle is located at the $(|S|, \text{imbalance})$ point.

- A production code would make k trials and use the best of the k partitions. While the partition using k trials will generally be better than that found using one trial, the cost will be k times as great. So we are interested in the partition quality as a function of the number of trials. Towards this aim, we define the k-median as follows.

  > If we make k trials and take the best partition, the probability is 1/2 that we have a separator weight (or imbalance or cost) less than or equal to the k-median value of separator weight (or imbalance or cost).

  We made 100 runs of the algorithm on each graph, collected the data and sorted the data into ascending order. We then find approximations to the 1-median at entry 50, the 2-median at entry 30 and the 3-median at entry 21. (If there are m sorted observations, an approximation to the k-median is found at location $m\left(1-(0.5)^{1/k}\right)$.)

7.1 Homogeneous Graphs

Some graphs are relatively homogeneous, e.g., any portion of a regular grid that does not lie on the boundary is similar to any other portion of the same size. Multilevel methods appear to be better suited to graphs of this type than are domain decomposition methods.

Our test matrix for this family of graphs is R2D7905. To generate this graph we placed 570 equally spaced vertices along the boundary of $[0,1]\times[0,8]$ and then added 7335 vertices to the interior in a quasi-random manner. We then found the Delauney triangulation of the vertices to generate the edges of the graph. While this graph is not regular, it is relatively homogeneous.

For this graph, a multilevel method generates a chain of coarse graphs that are relatively similar. The coarsening and refinement processes are smooth, and thus it is likely to create a well-balanced partition with near optimal separator weight. Indeed the results from `METIS` confirm this intuition. See Figure 11. Note the strong clustering of $|S|$ around 29 and 30 and the extremely well-balanced partitions; 99% show imbalance less than 1.008.

Our domain decomposition algorithm is much less likely to find a domain/multisector graph that is similar to the original graph. The crude way to construct the domains bounds their size but has little control of their shape or orientation. When projected back to the fine graph, the separator that is found by our block Kernighan-Lin algorithm may not be "close" to a separator of small size that defines a well-balanced partition. Often the smoother is able to reduce the separator size but cannot "move" the separator to create a well-balanced partition. On the other hand, `DDSEP` is able to locate the smallest separator, $|S| = 28$, in a partition that is still fairly well-balanced.

7.2 Not all local minima are created equal

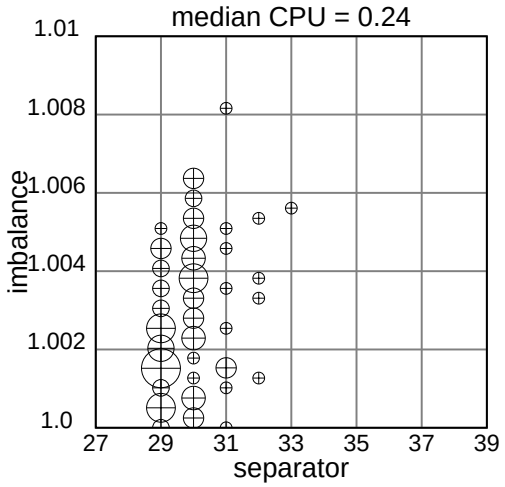
The "weakness" of domain decomposition, the inability to find a well-balanced coarse level separator, can be a strength when a graph has many separators with local minima weight. Consider a string of nine pearls. The separator with perfect balance slices through the center of the middle pearl. A nearby separator, located on the string, divides the pearls into sets of four and five. On the one hand we have a large separator with perfect balance, on the other a small separator with acceptable balance.

BCSSTK35 is not a string of pearls, but a finite element model of a car seat and frame assembly. Yet we see much the same behavior as we look at the top two plots in Figure 12. `METIS` finds extremely well-balanced partitions with relatively large separators. `DDSEP` finds a spectrum of partitions with varying degrees of imbalance and consistently

FIG. 11. *R2D7905*, $|V| = 7905$, $|E| = 54193$

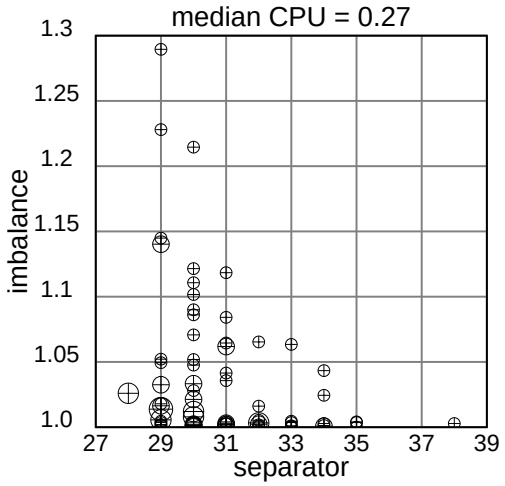
original graph, METIS(SHEM,BGKLR,GGPKL)

	1-median	2-median	3-median
Separator	30	29	29
Imbalance	1.003	1.002	1.001
Cost	60	58	58



original graph, DDSEP(40,80,1)

	1-median	2-median	3-median
Separator	30	30	29
Imbalance	1.063	1.027	1.023
Cost	62	60	59



smaller separators. In fact, twenty-three of the hundred DDSEP runs found the partition with $|S| = 90$, imbalance = 1.064, a particularly strong "attractor" partition.

The bottom plot shows the results for DDSEP acting on the natural compressed graph [2] which has 6611 vertices and 72545 edges. Note that the CPU time has dropped by a factor of six. On the other hand, the partition quality also drops. At present we do not completely understand these results. There is some effect of granularity — the original graph is like sand, the natural compressed graph like gravel — that definitely influences the construction of the domain/segment partition and possibly the smoothing process. At least the results are not that much different, and we can easily make more runs to find a better partition because the CPU time is so much reduced.

7.3 Beware of nodes of high degree

In Section 7.1 we saw that homogeneity is a desirable property for a graph as we search for a well-balanced partition with a small separator. The converse is also true — heterogeneity can cause problems. BCSSTK08 is the model of a television studio; its graph contains 1074 vertices and 12960 edges. The average degree of a vertex is 12.1, the median degree is 10, yet 19 vertices have degrees over 63, 14 over 100, and 3 over 200.

These nodes of high degree gave DDSEP much difficulty, until we found a solution. See the middle plot in Figure 13. The one hundred runs generated relatively few distinct partitions. We traced the difficulty to the generation of the domain/segment partition. Once a node of high degree was absorbed into a domain, havoc resulted. No matter how we varied the $\omega_{\min}$ and $\omega_{\max}$ parameters that constrained domain weight, we would end up with two or three very large domains and often a segment that contained over a hundred vertices. These inferior domain/segment partitions crippled the algorithm.

For a sparse matrix ordering, nodes of high degree must be put into the separator. The star graph with n vertices is a simple example. If the root of the star graph is ordered first, the factor matrix has $n(n+1)/2$ entries. If the root is ordered last, the factor matrix has $2n-1$ entries. This observation led us to Step 1 of the algorithm in Section 4.1. In the beginning, we put vertices of high degree into the initial multisector When small domains are absorbed and the separator made minimal, we make an effort to keep the nodes of high degree from being absorbed into a domain. This process pays off as we see in the bottom plot of Figure 13. Here we forced nodes with degree four times the median into the multisector. There is very little repetition of partitions in the 100 runs and the bisectors are much better.

BCSSTK08 appears to give the multilevel method difficulties. The imbalance is much larger than for the other two matrices, and the separators worse than those found by DDSEP. We conjecture that the nodes of high degree are the problem.

7.4 Effect of Input Parameters

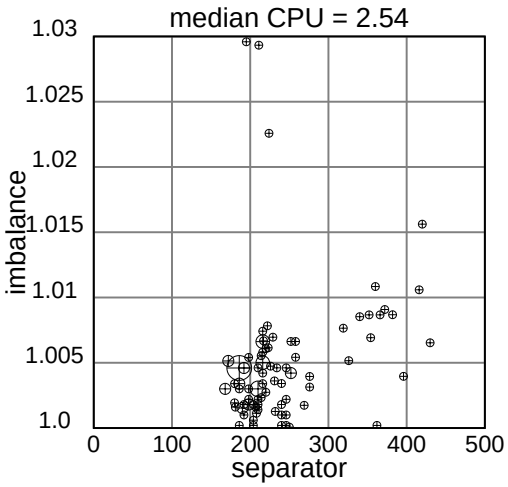
We now discuss the influence of the four input parameters to the DDSEP code.

- The bounds ω_{min} and ω_{max} control the minimum and maximum domain sizes. As these values decrease together, the time to generate the domain/segment partition drops, the block Kernighan-Lin time increases and the partition quality declines. As the values increase, the time to generate the domain/segment partition grows, the block Kernighan-Lin time decreases to almost nothing and again the partition quality declines. The required CPU time and the partition quality are concave up functions of ω_{min} and ω_{max}, but fortunately, the curves are fairly flat near their minimum. Setting

FIG. 12. *BCSSTK35*, $|V| = 30237$, $|E| = 1450163$

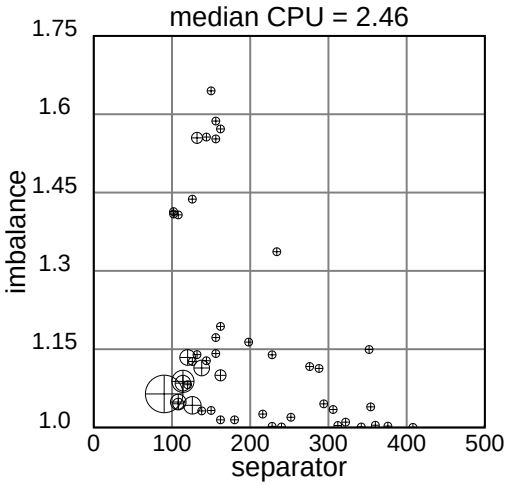
original graph, METIS(SHEM,BGKLR,GGPKL)

	1-median	2-median	3-median
Separator	216	198	186
Imbalance	1.004	1.002	1.002
Cost	480	428	408



original graph, DDSEP(40,80,1)

	1-median	2-median	3-median
Separator	126	108	90
Imbalance	1.068	1.049	1.042
Cost	257	221	186



compressed graph, DDSEP(40,80,1)

	1-median	2-median	3-median
Separator	138	114	108
Imbalance	1.088	1.049	1.043
Cost	303	256	237

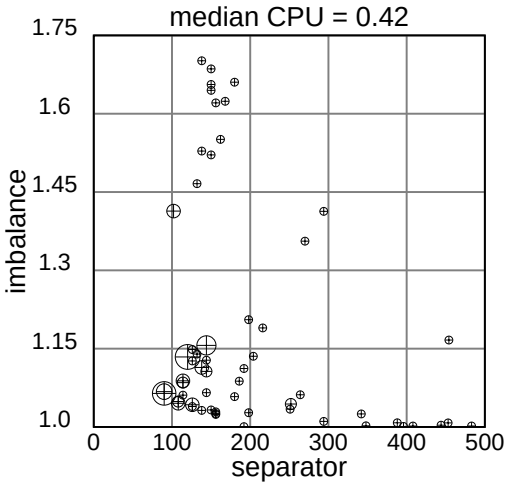
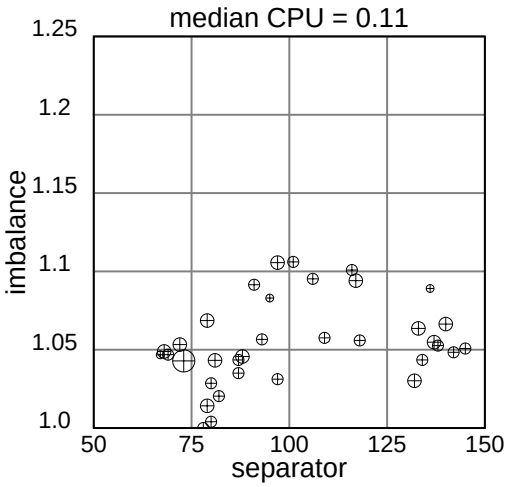


FIG. 13. *BCSSTK08*, $|V| = 1074$, $|E| = 12960$

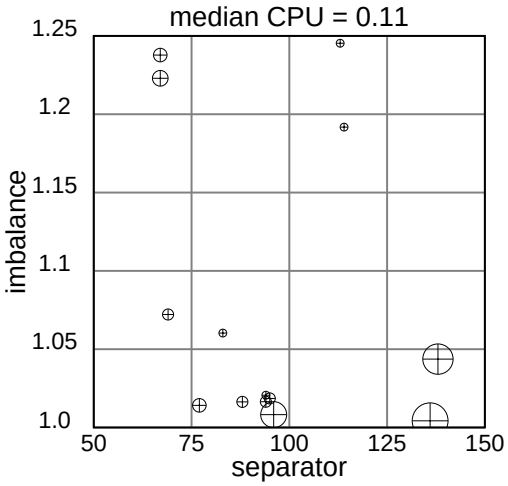
METIS(SHEM,BGKLR,GGPKL)

	1-median	2-median	3-median
Separator	97	80	79
Imbalance	1.053	1.043	1.043
Cost	204	163	159



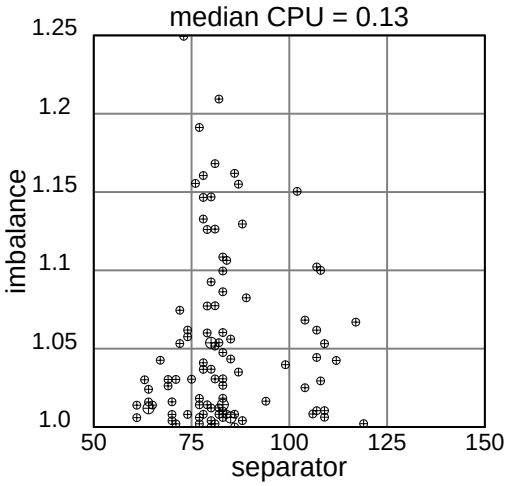
no freeze, DDSEP(40,80,1)

	1-median	2-median	3-median
Separator	136	96	94
Imbalance	1.024	1.008	1.004
Cost	273	193	190



freeze = 4, DDSEP(40,80,1)

	1-median	2-median	3-median
Separator	81	77	74
Imbalance	1.031	1.012	1.008
Cost	167	159	152



$(\omega_{min}, \omega_{max})$ to (40,80) has worked well for graphs with 1000 to 30000 vertices; larger graphs will require larger $(\omega_{min}, \omega_{max})$ settings to keep the domain/segment graph moderate in size.

- Forcing nodes of high degree to lie in the multisector is extremely important for some matrices, e.g., BCSSTK08. This operation cost very little, all that is required is the external degree of each node, and that is needed elsewhere. By default we force nodes into the multisector that have degree four times the median degree. In the future we intend to automate the selection of this parameter by sorting the external degrees in ascending order, then search for a jump to locate the vertices of high degree to be placed in the multisector.

- The α parameter defines the behavior of the penalty function. Recall the distribution of the data points in the bottom plot of Figure 13 that corresponds to $\alpha = 1$. This cloud of data points changes little for $\alpha \in [0.5, 5]$. For $\alpha < 0.5$ the cloud is shifted slightly to the left (smaller separators) but expands greatly in the vertical direction (larger imbalance). For $\alpha \geq 10$ the cloud is shifted slightly to the right (larger separators) but compressed in the vertical direction (better balance). The transition as a function of α is rather smooth, unlike the 2-norm evaluation functions of Section 2.2 that are very sensitive to the β parameter.

At present, `DDSEP` spends 60%-85% of its time creating the domain/decomposition, 5%-15% to construct the initial bisector using block Kernighan-Lin, and 20%-40% to improve the separator using graph matching.

8 Concluding Remarks and Future Work

In this paper, we have presented a new three-step graph partitioning method. It first uses domain decomposition to define a coarse graph, from which a good partition is obtained by a block variant of the Kernighan-Lin scheme. The partition and its bisector are then projected back to the original graph where bipartite graph matching is used to improve the projected bisector and partition. The method has proved to be effective and it compares favorably with existing partitioning schemes.

Most of the time is spent constructing the domain decomposition of the graph. This is understandable for the other two steps work with graphs that are much smaller in size than the original. Nonetheless, we feel there are improvements to be made, both to reduce the execution time as well as to construct a decomposition that recognizes non-homogeneities of the graph. Forcing nodes of high degree into the multisector is only a first simple step.

The bisector smoothing can also be improved. Our graph matching technique currently uses the "coarse" Dulmage-Mendelsohn decomposition to locate movable sets. If in addition we were to use the "fine" decomposition [20], the bisector weight would not decrease further but the partition's balance would improve. We intend to implement these changes in our software and report on the results in a future paper.

Finding a graph bisector is useful in a recursive bisection algorithm to partition a graph. The cost of such a partition is roughly linear in the number of levels of the recursion. Methods that find "quadri-sectors", "octa-sectors" and higher separators can find good partitions at less cost [13]. Here domain decomposition can be profitably used. We are presently extending our software to find "multisectors".

Acknowledgements

We would like to recognize David Keyes and David P. Young for many interesting conversations on domain decomposition and multigrid. We need to thank Bruce Hendrickson and George Karypis for correspondence and conversation about this work. Fritz Scholz and Antonio Possolo have answered our questions about statistics.

References

[1] C. C. Ashcraft. The domain/segment partition for the factorization of sparse symmetric positive definite matrices. Technical Report ECA-TR-148, Boeing Computer Services, Seattle, WA, 1990.

[2] C. C. Ashcraft. Compressed graphs and the minimum degree algorithm. Technical Report BCSTECH-93-024, Boeing Computer Services, Seattle, WA, 1993. To appear in SIAM J. Sci. Comput., November, 1995.

[3] C. C. 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.

[4] C. C. Ashcraft and J.W.H. Liu. Generalized nested dissection: some recent progress. In *Fifth SIAM Conference on Applied Linear Algebra*, Snowbird, Utah, June 18, 1994.

[5] T. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42:153–159, 1992.

[6] T. Bui and C. Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *Proceeding of Sixth SIAM Conference on Parallel Processing*, pages 445–452, 1993.

[7] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partition. In *ACM IEEE Proc. 19th Design Automation Conference*, pages 175–181, Las Vegas, Nevada, June 14-16, 1982.

[8] J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

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

[10] J.A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[11] T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. Technical report, Computer Science Department, University of Minnesota, Minnesota, 1995.

[12] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[13] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16:452–469, 1995.

[14] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. Technical Report TR 95-037, Department of Computer Science, University of Minnesota, Minnesota, 1995.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minnesota, 1995.

[16] G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, Minnesota, 1995.

[17] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[18] C. E. Leiserson and J. G. Lewis. Orderings for parallel sparse symmetric factorization. In *Parallel Processing for Scientific Computing*, pages 27–31, 1989.

[19] J. W. H. Liu. A graph partitioning algorithm by node separators. *ACM Trans. on Math Software*, 15:198–219, 1989.

[20] A. Pothen and C. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. on Math Software*, 16:303–324, 1990.

[21] A. Pothen, H. Simon, and K.P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Analysis and Applic.*, 11:430–452, 1990.

[22] J. D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, Rockville, Md, 1984.

[23] L. Wang. *Spectral nested dissection.* PhD thesis, The Pennsylvania State University, Department of Computer Science and Engineering, 1994.