Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations

Bradford L. Chamberlain

October 13, 1998

Abstract

This paper surveys graph partitioning algorithms used for parallel computing, with an emphasis on the problem of distributing workloads for parallel computations. Geometric, structural, and refinement-based algorithms are described and contrasted. In addition, multilevel partitioning techniques and issues related to parallel partitioning are addressed. All algorithms are evaluated qualitatively in terms of their execution speed and ability to generate partitions with small separators.

1 Introduction

In its most general form, the *graph partitioning problem* asks how best to divide a graph's vertices into a specified number of subsets such that: (*i*) the number of vertices per subset is equal and (*ii*) the number of edges straddling the subsets is minimized. Graph partitioning has several important applications in Computer Science, including VLSI circuit layout [8], image processing [43], solving sparse linear systems, computing fill-reducing orderings for sparse matrices, and distributing workloads for parallel computation. Unfortunately, graph partitioning is an NP-hard problem [13], and therefore all known algorithms for generating partitions merely return approximations to the optimal solution. In spite of this theoretical limitation, numerous algorithms for graph partitioning have been developed during the past decade that generate high-quality partitions in very little time. This paper provides an overview of these algorithms, focusing on their application to parallel computing.

One of the fundamental problems that must be addressed in every parallel application is that of *workload distribution*—the distribution of data and computation across a processor set. Optimal distributions

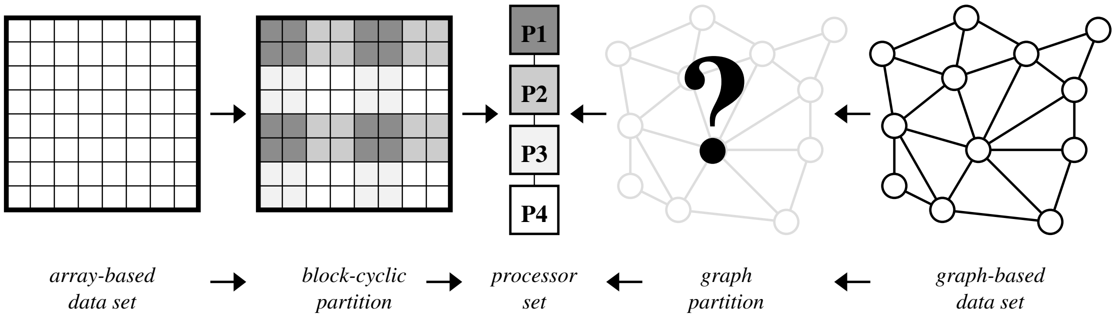
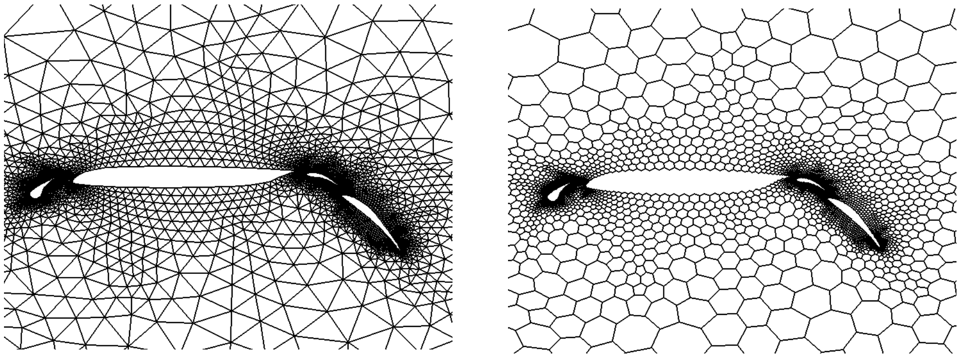


Figure 1: An illustration of the workload distribution problem. Elements of the array (left) and the unstructured graph (right) need to be distributed among four processors (center). Due to its regular structure, the array is amenable to straightforward distributions such as the block-cyclic one shown here. The graph has no obvious mapping to the processor set and therefore requires a graph partitioning algorithm.

minimize an application's overall runtime, typically by ensuring that each processor has an equal amount of work while minimizing the parallel overhead induced by the distribution (most notably in the guise of interprocessor communication). Some applications are easy to distribute. For example, dense array-based problems typically have a high degree of regularity, allowing the array elements to be distributed using straightforward *blocked*, *cyclic*, or *block-cyclic* schemes. These distributions are advantageous due to their simplicity and their ability to take advantage of an array's regular structure.

Unfortunately, many parallel computations involve data sets that are not so regular in structure, thereby necessitating more sophisticated partitioning methods (Figure 1). Such data sets include those used by finite volume methods for computational fluid dynamics, and by finite element and finite difference methods for structural analysis. For example, the finite volume mesh in Figure 2(a) stores data values at each triangle in order to compute airflow past a four-element wing.

These unstructured data sets induce workloads that can be represented in a graph-based form. Each node of a *workload graph* represents a unit of data and the computation that must be performed on it, while each edge represents a data dependence between two vertices. For example, in the computation of Figure 2(a), each triangle's values are iteratively recomputed using the values of its neighboring tri-



(a) (b)

Figure 2: Example of a finite volume simulation used to compute airflow past a four-element wing. (a) The mesh defined for the problem. Each triangle has an associated set of data values. Flux is computed for each edge using the values of its neighboring triangles. (b) The mesh's dual graph, used to partition the computation. Each vertex represents a triangle's data. Edges connect triangles that need to refer to each others' values (*source:* [44]).

angles. Thus, the workload induced by this mesh can be represented using a graph in which vertices represent mesh triangles and edges connect nodes whose triangles share a common edge (Figure 2(b)). For workloads described in this graph-based framework, a parallel distribution can be computed using a graph partitioning algorithm. Since workload distribution strives to divide computation (graph vertices) evenly across a processor set, while minimizing interprocessor communication (edges that straddle two subsets), it can be phrased as a graph partitioning problem in which the number of partitions is equal to the number of processors.

In the past decade, several novel approaches have been developed for partitioning graphs for parallel computation. This paper describes several of the most compelling algorithms as well as the techniques that led to their development. Each algorithm is evaluated in terms of the time it takes to compute a partition as well as the quality of the partitions it generates (measured by the number of straddling edges). Secondary considerations include whether the algorithm has theoretical guarantees of its partition quality, and whether it has an efficient parallel implementation.

The rest of the paper is organized as follows: The next section provides a more complete introduction to

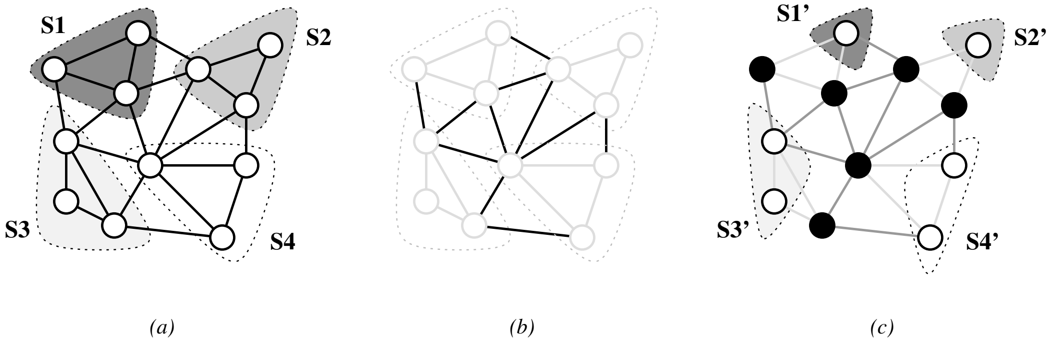


Figure 3: (a) A balanced four-way partition of the graph from Figure 1. (b) The set of cut edges, E_c resulting from this partition ($|E_c| = 11$). (c) A vertex separator induced by E_c that partitions the remaining vertices into four roughly balanced subsets.

the graph partitioning problem and its application to workload distribution. Sections 3–5 present a survey of graph partitioning algorithms, dividing them into three classes: those that utilize the geometric properties of a graph, those that operate on a graph's combinatorial structure, and those that use local refinement techniques. Section 6 describes general strategies that accelerate the partitioning process by *contracting* the input graph. Issues related to computing partitions in parallel are addressed in Section 7. The final section provides an overall evaluation of the algorithms and reconsiders their application to the problem of workload distribution.

2 Graph Partitioning

In this paper, a graph $G = (V, E)$ is defined in terms of a set of vertices, V, and a set of edges, E. Edges connect vertices from V pair-wise and are undirected. Self-loops are not permitted. In some graphs, the vertices may have spatial coordinates associated with them that define their relative positions in $\mathbb{R}^d$. A *p-way partition* of a graph is a mapping $\mathcal{P} : V \to [1 \ldots p]$ of its vertices into p subsets $S_1, S_2, \ldots, S_p$, where $\bigcup_i S_i = V$ and $S_i \cap S_j = \emptyset$ wherever $i \neq j$ (Figure 3(a)). Every partition generates a set of *cut edges*, E_c, defined as the subset of E whose endpoints lie in distinct partitions

($E_c = \{(v_i, v_j) \mid (v_i, v_j) \in E,\ \mathcal{P}(v_i) \neq \mathcal{P}(v_j)\}$—see Figure 3(b)). The *weight* of each subset, $|S_i|$, is defined to be the number of vertices mapped to that subset by $\mathcal{P}$.

Given a graph as input, the graph partitioning problem seeks to find a p-way partition in which each subset contains roughly the same number of vertices ($|S_i| \leq \lceil |V|/p \rceil$) and the number of cut edges, $|E_c|$, is minimized. For input graphs that represent workload as described in the introduction, each partitioned subset represents data and computation that should be assigned to a single processor. The cut edges represent the interprocessor communication required by the distribution. Thus, the graph partitioning problem attempts to find a distribution that balances the computation done by each processor while minimizing the total interprocessor communication.

As described here, graph partitioning is performed using an *edge separator* E_c—a group of edges whose removal breaks the graph into disjoint subsets. A related problem tries to break the graph into subsets using a *vertex separator* $V_s \in V$ of minimum size (Figure 3(c)). Vertex separators are used for performing *nested dissection* [14], a technique useful for reordering a matrix's rows and columns to benefit its parallel factorization. Since workload graphs use vertices to represent data, vertex separators are inappropriate since they correspond to data being eliminated from the computation. Note that most algorithms for computing vertex separators do so by first finding an edge separator and then computing a (potentially minimum) vertex cover for the graph induced by E_c.

2.1 Recursive Bisection

An instance of graph partitioning that deserves special attention is the *graph bisection problem*. This is simply a variation on graph partitioning in which G must be divided into two subsets. Although bisection seems considerably easier than general p-way partitioning, it is still NP-hard.

Most p-way partitioning algorithms utilize a divide-and-conquer approach known as *recursive bisection*. This technique generates a p-way partition by performing a bisection on the original graph and then

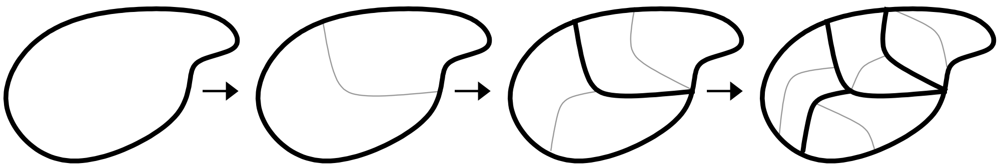


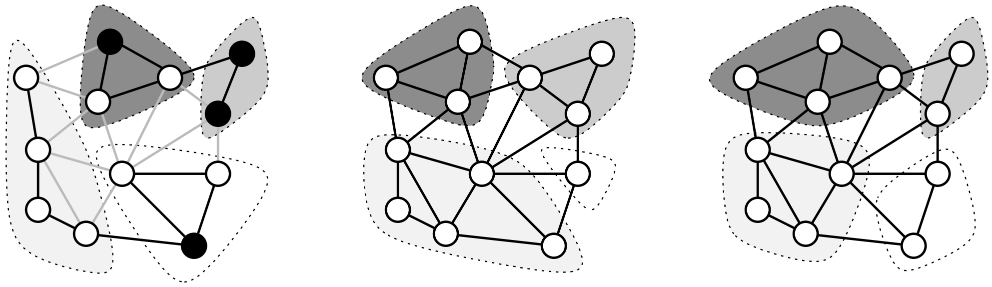
Figure 4: An example demonstrating the use of recursive bisection to compute an eight-way partition for an abstract graph.

recursively considering the resulting subgraphs (Figure 4). It has been shown that even if recursive bisection is performed using an optimal bisection algorithm, it can still result in a suboptimal p-way partition [45]. In spite of this theoretical limitation, recursive bisection remains the primary graph partitioning strategy due to its simplicity compared to computing p-way partitions directly. The majority of the algorithms in this paper rely on recursive bisection.

2.2 Graph Partitioning Variations

Several variations of the basic graph partitioning problem exist. The *weighted graph partitioning problem* allows weights to be associated with the vertices and edges of G. In this problem, a good partition is one in which the total vertex weight of each subset is roughly equal, and the total weight of the cut edges is minimized (Figure 5(a) shows an example). In the context of workload graphs, node weights can be used to encode differing computation expenses across the graph (*e.g.*, boundary locations vs. internal locations). Similarly, edge weights can signify the volume of communication required between two nodes. For the sake of simplicity, this paper will concentrate on the unweighted version of the problem, though most of the algorithm descriptions can be trivially extended to handle weighted graphs.

The *δ-partitioning problem* is one in which some imbalance in the subset weights is tolerated in hopes of significantly reducing the number of cut edges. In this problem, a tolerance δ is supplied as input where $1/p \leq \delta \leq 1$. Legal partitions are those that yield subsets with weight $|S_i| \leq \lceil \delta |V| \rceil$ (Figure 5(b)). Note



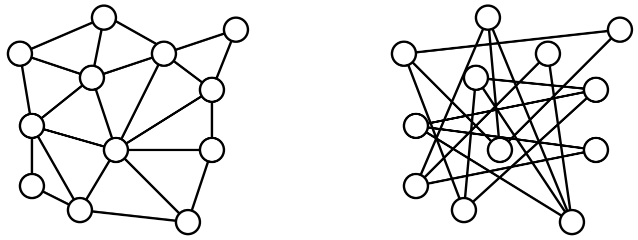
(a) *(b)* *(c)*

Figure 5: Variations on the graph partitioning problem. (a) A weighted partitioning. Assume that darker edges and vertices have weight 2, while lighter components have weight 1. The partition shown demonstrates a balanced 4-way partition where $|S_i| = 4$ for each subset. Although difficult to verify, it probably minimizes $|E_c|$ since few of the heavy edges straddle the subsets. (b) An example of δ-partitioning for $\delta = 5/12$. Note that permitting imbalance in the subsets allows $|E_c|$ to be smaller than in the balanced partition of Figure 3(a). (c) Skewed partitioning in which the target weights are $2 : 1 : 2 : 1$.

that the standard graph partitioning problem is simply a δ-partitioning problem in which $\delta = 1/p$. Another variation that allows for unbalanced subsets is the *skewed partitioning problem*. In this version, user-supplied weights are associated with each subset to specify a desired imbalance in the partition. Algorithms for this problem compute partitions whose subsets are weighted proportionally to those specified by the user (Figure 5(c)).

In the context of workload distribution, these generalizations allow the characteristics of a parallel machine to have some bearing on the partitioning. For example, if a machine's interprocessor communication overhead is sufficiently high, it might be worthwhile to tolerate some degree of load imbalance in order to drastically reduce the communication volume. This tradeoff can be described using δ-partitioning. As another example, if a parallel computation is to be performed on a heterogeneous processor set, the relative performance of the processors can be described by specifying appropriate target weights in the skewed partitioning problem.

The closer one gets to modeling the behavior of actual machines, the more complicated things become. One could imagine further generalizations of the graph partitioning problem that attempt to minimize the



(a) (b)

Figure 6: (a) A graph that suits the assumptions made by geometric algorithms since its vertices are connected to their nearest neighbors. (b) A graph whose vertices are connected to distant nodes rather than close ones. Geometric algorithms would not be expected to partition it well.

number of processors with which each processor communicates, or that attempt to model the non-uniform communication costs that occur in clusters of multiprocessors. Though these are interesting and important issues, they are beyond the scope of this paper, and will be touched on again only briefly in the discussion section.

3 Geometric Algorithms

Geometric approaches to the graph partitioning problem are those that use the geometric coordinates associated with a graph's vertices to aid in the creation of a partition. By nature, these algorithms are fairly straightforward to understand and to implement. Unfortunately, since not all graphs have coordinates associated with their vertices, geometric algorithms are not completely general. However, many common parallel computations such as finite volume, finite element, and finite difference methods do produce graphs that are geometrically embedded in d-dimensional space. In these cases, the extra geometric information provides a straightforward global relationship between a graph's vertices which can be used to achieve quick runtimes and high-quality partitions. It should be noted that most geometric algorithms assume that spatial proximity and vertex connectivity are strongly correlated—that is, vertices which are closer together in $\mathbb{R}^d$ will tend to have shorter connecting paths in E (Figure 6).

3.1 Coordinate Bisection and its Variations

The most straightforward instance of a geometric approach is a technique known as *recursive coordinate bisection* or *recursive orthogonal bisection*. In this algorithm, vertices are sorted according to their coordinates in each of the d dimensions. The algorithm considers a candidate hyperplane for each dimension which is perpendicular to the coordinate axis and located at V's median coordinate value. By construction, each of these d hyperplanes will bisect the graph. The algorithm then selects the candidate hyperplane that cuts the smallest number of edges. The subgraphs are then considered recursively using the same technique. Partitions generated by coordinate bisection are illustrated in Color Plate 1(a) and (b). Variations on coordinate bisection have been considered by several authors [17, 5].

One important improvement to coordinate bisection, *recursive inertial bisection*, does a better job of getting an overall gestalt for the graph by computing its *principal axis of inertia*, I. By definition, I is the axis around which the graph's vertices have a minimum rotational angular momentum. The algorithm then selects a bisecting hyperplane perpendicular to this axis. The intuition behind this approach is that vertices in V will tend to lie along I and in close proximity to it, causing planes perpendicular to the axis to cut a small number of edges.

All variations on coordinate bisection are straightforward to understand and implement, but are simplistic: each algorithm considers only a small number of candidate bisecting hyperplanes, and in all but the inertial method, the choice is done in an ad hoc manner with little regard for the graph's global properties. Another disadvantage to this class of algorithms is that the separators computed are hyperplanes, which constitute a fairly restricted class of partition when one considers the exponential degrees of freedom available in a graph's combinatorial structure. In general, optimal separators are likely to involve a more complex and ragged boundary between subsets. Studies comparing hyperplane-based algorithms against the more sophisticated techniques described in the sections that follow indicate that while hyperplane bisectors are fast to compute, they generally result in partitions of considerably worse quality [44, 16, 5].

3.2 Circle Bisection

Miller *et al.* describe an algorithm called *recursive circle bisection* that addresses the drawbacks of hyperplane-based algorithms [33]. It uses global information about the graph's vertices to compute separators using circles and spheres rather than lines and planes. Since circle bisectors add an additional degree of freedom to the bisecting surface (a variable radius), they can be used to express more complex separators.

The two main concepts required to understand circle bisection are *stereographic projection* and *centerpoints*. Stereographic projection is a means of mapping points to a higher dimension by projecting them from $\mathbb{R}^d$ to the surface of the unit sphere in $\mathbb{R}^{d+1}$. This is done by first embedding the point in $\mathbb{R}^{d+1}$ by setting its $(d+1)^{\text{st}}$ coordinate to 0. Next, a line is projected from $(0,\ldots,0,1)$ through the point. The intersection of this line and the unit sphere defines the point's location in the stereographic projection.

A centerpoint for a set of points in $\mathbb{R}^d$ is defined to be a point through which *every* hyperplane is guaranteed to create two subsets whose weight ratio is no worse than $d : 1$. One key difference between centerpoints and centers of mass is that centerpoints are relatively unaffected by distant outliers in the set. Every set of points in $\mathbb{R}^d$ is guaranteed to have a centerpoint, and its location can be computed using linear programming methods [33].

The circle bisection algorithm proceeds as follows: map the vertices of V to the unit sphere in $\mathbb{R}^{d+1}$ using a stereographic projection and then compute a centerpoint for the projected points. Next, move the points around on the sphere's surface in order to map the centerpoint to the origin: first by rotating them until the centerpoint is at $(0,\ldots,0,r)$; then by dilating them along the sphere's surface (equivalent to scaling by a factor of $\sqrt{(1-r)/(1+r)}$ in $\mathbb{R}^d$). Next, choose a random *great circle*, C_g, on the unit sphere to split the points into two subsets. Note that each great circle corresponds to the intersection of a plane through the centerpoint (origin) and the sphere; thus, G_c represents a partition of the points with a worst-case weight ratio of $(d+1) : 1$. C_g is then mapped back to $\mathbb{R}^d$ by inverting the dilation, rotation, and stereographic projection. The result is a circle (or line in the degenerate case) that separates the original

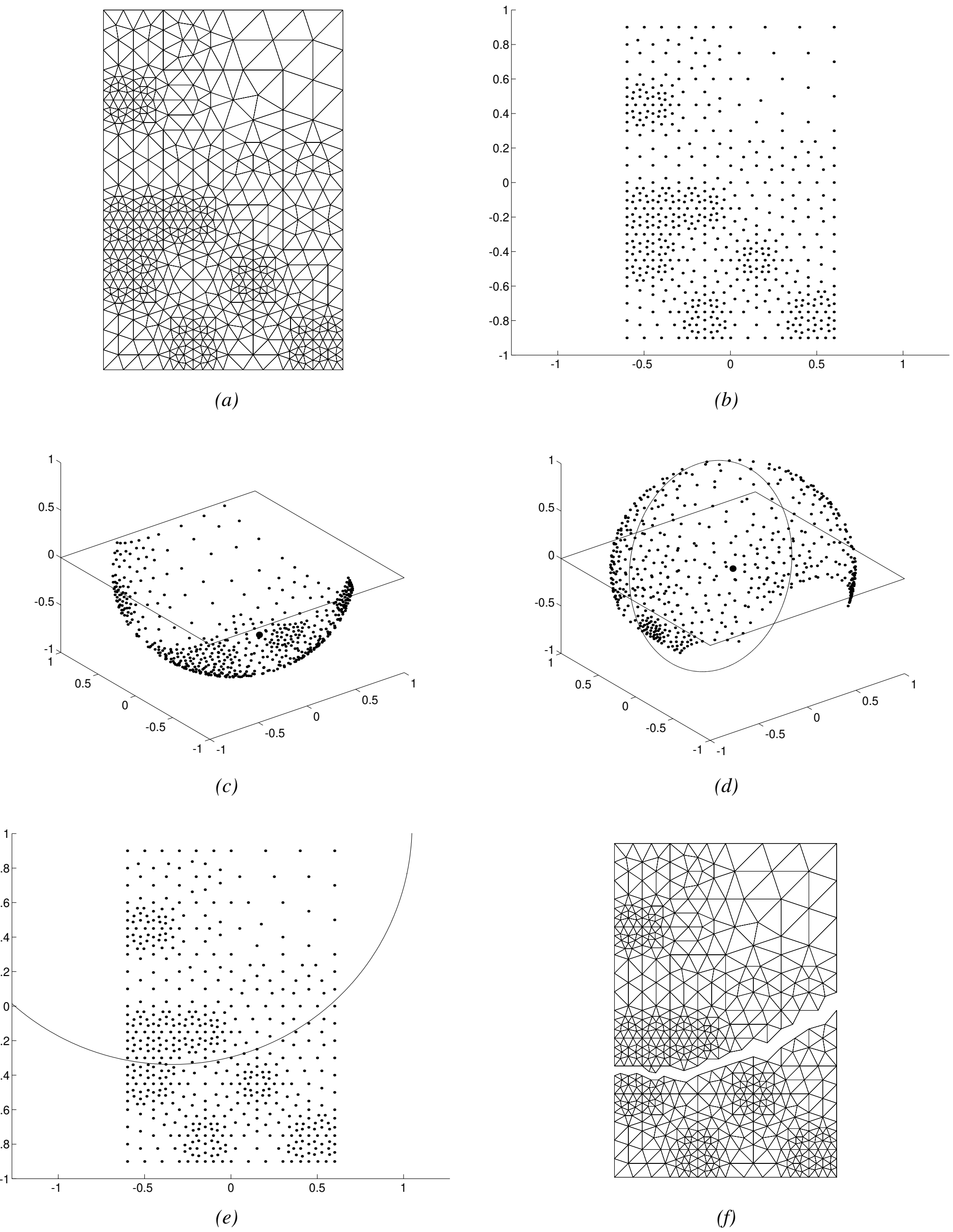


Figure 7: An example of the circle bisection algorithm. (a) A 2D input graph. (b) The graph's vertices. (c) The stereographic projection of the vertices to the unit sphere in 3D. The large dot is a centerpoint. (d) The points have been rotated and dilated on the surface of the sphere in order to move the centerpoint to the origin. A random great circle is chosen. (e) The dilation, rotation, and stereographic mapping are inverted to map the circle back to 2D. (f) The partition computed by the algorithm (*source:* [16]).

vertices into two sets: those inside and those outside the circle. See Figure 7 for an illustration of the circle bisection algorithm.

Gilbert *et al.* report on their practical experience with an implementation of the circle bisection algorithm [16]. They deviate from the pure algorithm above in three main ways to accelerate the computation while still producing a good separator. The first difference is their use of *geometric sampling* to reduce the complexity of computing a centerpoint. Rather than considering the complete set V, they select a random subset of a thousand vertices to represent the graph. Secondly, although the linear programming algorithm for finding centerpoints runs in polynomial time, it uses a large number of constraints, causing it to be too slow for practical use. Instead, the authors use a fast algorithm for computing approximate centerpoints using *radon points* [6]. Applying this algorithm to the sampled vertex set results in a vastly accelerated computation that produces good centerpoints in practice.

The third difference comes in the selection of a partition. To find their great circle, Gilbert *et al.* generate a number of approximate centerpoints and a few circles for each, selecting the one that cuts the fewest edges. Each circle is generated by randomly constructing a normal vector, weighted slightly towards the inertial axis I. Although the great circles are not guaranteed to result in an even partition, in practice most splits were within 20% of balanced. To achieve an exact bisection, the separating circle's plane is nudged in the direction of its normal vector until a balance is reached. The authors report that this does not dramatically affect the cut size.

Gilbert *et al.* compute partitions for a series of standard graphs and compare their results to those generated using coordinate bisection and spectral bisection (described in the next section). Their findings show that the circle-based method generally results in partitions that are better than coordinate bisection, yet similar in size to spectral bisection. The chief advantage to the geometric method is its speed, running approximately an order of magnitude faster than the spectral algorithm.

Circle bisection is built upon a body of theoretical work which characterizes graphs that have good separators [35] and which places theoretical lower bounds on geometric separator sizes [5, 34]. This work serves as a strong foundation for explaining why circle bisection tends to result in good separators.

4 Structural Algorithms

The geometric algorithms of the previous section all share two common weaknesses: the first is that they require vertices of the input graph to have geometric coordinates associated with them, yet this is not the case for all graphs. The second is that they never refer to the connectivity structure of the graph, E, but rather assume that spatial proximity implies a short connecting path. Although this is a reasonable assumption for many graphs, counterexamples exist. For example, in the graphs of Figure 2, vertices on opposite sides of the wing flaps are fairly close in space, but have long connecting paths in E. The shortcomings of geometric algorithms are addressed by *structural* or *combinatorial* algorithms—ones that compute partitions by referring only to the graph's connectivity. This section describes a number of structural approaches.

4.1 Graph-Walking Algorithms

Recursive level-structure bisection is a combinatorial approach that is very intuitive in nature [15]. It is similar to the coordinate bisection algorithms described in the previous section, but defines the distance between two vertices as the length of their shortest connecting path, rather than their distance in Euclidean space. The algorithm finds two vertices of near-maximal distance from one another and then performs a breadth-first search from one of the vertices, until it has reached half of the vertices in the graph. These vertices are placed in the first subset, leaving the remainder in the second. The algorithm is then applied recursively to each of the subgraphs. Color Plate 1(c) and (d) illustrate the use of level-structure bisection. Although this algorithm is relatively straightforward and fast, it tends to result in relatively poor partitions [44].

A very similar approach is *Farhat's greedy algorithm* [9]. It also accumulates sets of vertices by traversing the graph in a breadth-first manner, but differs in that it computes its p subsets directly, without resorting to recursive bisection. The subsets are constructed one at a time, starting from an arbitrary vertex. Once the traversal has reached $|V|/p$ vertices, they are assigned to a subset, S_i. The process is then repeated for S_{i+1}, starting at the boundary vertex of S_i with the smallest number of unexplored edges.

One other related approach is the *greedy graph growing algorithm* developed by Karypis and Kumar [25]. This is another algorithm for bisection that grows a subset of vertices around an arbitrary root. However, rather than walking the graph in a strict breadth-first manner, it adds vertices to the subset in an order determined by their benefit—namely, the amount that the edge cut will improve if the vertex is added to the subset. Thus, at each step, the vertex that would cause the largest decrease (or smallest increase) in the number of cut edges is added to the subset.

4.2 Spectral Algorithms

All of the graph-walking approaches above have the disadvantage of being relatively blind and localized in their consideration of the graph's structure, looking only one edge past the current frontier at a time. Pothen, Simon, and Liou introduce a *spectral* graph partitioning algorithm that addresses these shortcoming by considering a graph's global connectivity properties when computing a solution [40]. This technique is referred to as *recursive spectral bisection* (RSB).

Recursive spectral bisection utilizes the *Laplacian matrix*, L, of the input graph—a $|V| \times |V|$ matrix that encodes information about G's connectivity. Each entry of L is defined as follows:

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

For intuition as to how the Laplacian matrix is used, consider encoding a 2-way partition of a graph using a $|V|$-ary vector x in which x_i is 1 if vertex i is in the first subset and -1 if it is in the second. Note that for a balanced bisection, $\sum_i x_i = 0$. For a perfect bisection x_p (one in which no edges are cut), the matrix-vector product Lx_p yields the zero vector, since each vertex's degree will be canceled out by the combined sum of its edges. More generally, the product $x^t Lx$ for any partition vector can be shown to be equal to 4 times the number of cut edges. Thus, the graph bisection problem can be rephrased as: *Find a partition vector x where $x_i = \pm 1$ such that $\sum_i x_i = 0$ and $x^t Lx$ is minimized.*

Phrasing the graph bisection problem in this form does nothing to make the computation of a solution easier—it is still NP-hard. However, the problem *is* tractable when relaxing it from its discrete form to a continuous version—one in which elements of x can take on values in the interval $[-\sqrt{n}, \sqrt{n}]$ rather than just 1 and -1. Moreover, a minimum solution to the continuous problem is formed by the $\sqrt{n}$-length second eigenvectors of the Laplacian matrix [19].

This raises an obvious question: will a solution to the continuous problem have any bearing on the discrete problem? Fortunately, the answer turns out to be "yes." The construction of the Laplacian matrix is such that its smallest eigenvalue λ_1 is zero, with an associated eigenvector $\vec{x_1}$ of all ones. In his studies of the Laplacian matrix, Mohar determined that for connected graphs, the magnitude of the second smallest eigenvalue, λ_2, serves as a measure of G's connectivity [36]. Moreover, the magnitude of the second eigenvector's i^{th} element gives a rough indication of vertex i's distance from other vertices in G: the closer two values are numerically, the shorter the connecting path between their corresponding vertices. These special properties of the second eigenvector $\vec{x_2}$ were thoroughly investigated by Fielder [11, 12], whose work provided the theoretical justification for its use in graph partitioning. Hence, x_2 is traditionally referred to as the *Fielder vector*.

The recursive spectral bisection algorithm proceeds as follows: Compute the Fielder vector for G using the *Lanczos algorithm* [37], modified to avoid computing the non-Fielder eigenvectors. Next, determine the median value of the Fielder vector's components and use this to partition vertices of G into two subsets: those whose corresponding Fielder components are less than the median and those that are greater (vertices with the median value are split arbitrarily between the two groups). The algorithm is then applied recursively to each of the resulting subgraphs. Illustrations of its use are shown in Color Plate 1(e) and (f). It should be noted that in practice, L need not be explicitly represented, since its values can be quickly determined from the graph structure itself.

The principal drawback of RSB is the computation of the Fielder vector, which dominates the computation enormously. Although partitions generated by RSB are typically of very high quality, the time required by the algorithm is significant enough to be a serious impediment to its practical use [25, 16, 46]. Since its original formulation, much work has been done to accelerate the algorithm [41, 2, 1], some of which will be described in the following sections. One noteworthy extension to RSB is an implementation by Hendrickson and Leland [21] that uses additional eigenvectors to obtain simultaneous quadrisections and octasections of a graph. This often results in smaller partitions than those obtained by recursive calls to RSB, yet requires significantly less time.

5 Refinement Algorithms

One of the earliest graph partitioning algorithms has also turned out to be one of the most useful. Developed by Kernighan and Lin in the late 60's, the algorithm strives to improve an initial (possibly random) partition of the graph by trading vertices from one subset to the other with the goal of reducing the number of cut edges [31]. This general approach of refining an existing solution can be considered a class of algorithms unto itself.

The Kernighan-Lin (KL) algorithm is based on the notion of *gain*—a metric for quantifying the benefit of moving a vertex from one subset to the other. In KL, a vertex's gain is simply the total edge weight connecting it to the other subset minus that which connects it to its own. A greedy *steepest descent* refinement algorithm would simply move vertices with maximum gains until no vertices with positive gains remained. The disadvantage of this approach is that it can get trapped in a local minimum. KL avoids this by permitting *hill climbing*—vertices with negative gain are moved in hopes that they will lead to a more global minimum. KL avoids thrashing by limiting each vertex to one move per trial. During this swapping of vertices, the algorithm remembers the best partition that it encounters. Once all the vertices have moved (or a threshold of consecutive negative-gain moves has been reached), the algorithm restores the best partition found. The process can then be repeated using this new partition as a starting point.

Fiduccia and Mattheyses make some improvements to the base KL algorithm that utilize better data structures to improve the overall runtime [10]. For instance, the Fiduccia-Mattheyses (FM) algorithm minimizes the number of vertices whose gains need to be adjusted when a vertex is moved. These refinements are considered so fundamental to the base KL algorithm that the two approaches are often referred to interchangeably.

Although KL/FM can be used to refine random partitions, there is significant evidence to indicate that they work best when given a reasonably good starting partition [24, 39]. For this reason, KL/FM is often used as a local postprocessing step to improve a partition computed by a more globally-oriented algorithm [4, 22, 25]. Many uses of KL/FM modify the base algorithm to suit the programmer's specific needs [22, 25]. For instance, an implementation may only consider vertices that lie on the partition boundary since they will have the highest gain; or it may reduce the number of iterations run under the assumption that the initial partition was of high quality. Further generalizations of the algorithm extend it to refine p-way partitions, to handle weighted graphs, to use more complex measures of gain, etc. [22].

One last refinement-based technique that has been applied to graph partitioning is *simulated annealing*. Simulated Annealing is a general optimization technique in which hill climbing is permitted based on a probability that is systematically lowered as the run progresses. Although simulated annealing has proven useful in several application domains, its use in general graph partitioning has been disappointing to say the least. Studies comparing it with coordinate bisection, KL/FM, and spectral algorithms show that simulated annealing takes a significant amount of time to generate partitions of merely modest quality [24, 48]. Given the number of parameters that need to be set up and tuned for a simulated annealing run, not to mention

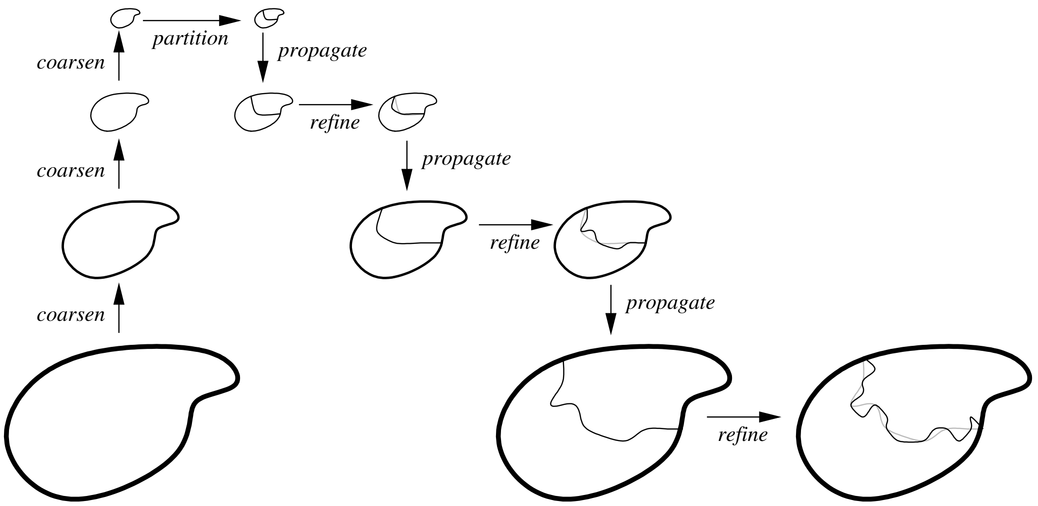


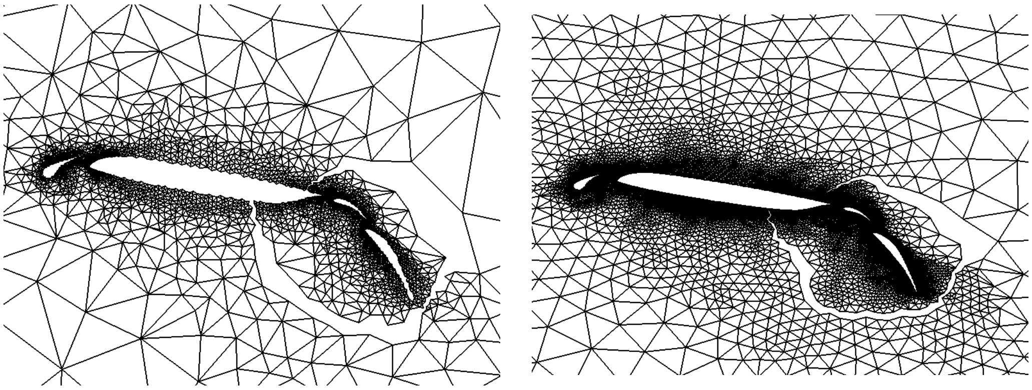
Figure 8: A schematic of the multilevel technique. The original graph (bottom left) undergoes a series of coarsening steps that reduce it to a smaller graph. This coarsest graph is partitioned using a standard algorithm. The partition is then propagated down to the finer graphs, potentially refining it at each level to account for the additional degrees of freedom. The result is a partition for the original graph.

the abundance of more successful graph partitioning techniques, simulated annealing has largely fallen out of favor.

6 Multilevel Techniques

One recent approach that has greatly accelerated the partitioning of graphs is the use of *multilevel* techniques. These techniques are analogous to *multigrid methods* for solving numerical problems. Both approaches construct a hierarchy of approximations to the original problem so that a coarse solution can quickly be generated. This solution is then progressively refined at the more detailed levels of the hierarchy until a solution for the original problem is reached. In the context of graph partitioning, this translates into creating a simplified graph that approximates the input graph, finding a partition for it, and then refining that partition to create a partition for the original graph.

All multilevel techniques for graph partitioning share the same general computational structure, though



(a) (b)

Figure 9: An example of multilevel bisection using MRSB on the graph of Figure 2. (a) The bisection of the coarsest approximation. (b) The resulting partition for the original graph (*source:* [2]).

the details may vary:

- **Coarsen:** Given the input graph G, construct a series of increasingly smaller graphs $G_1, G_2, ..., G_m$, each of which retains some sense of G's global structure.
- **Partition:** Partition the coarsest graph, G_m, using a standard algorithm.
- **Interpolate:** Propagate the solution for G_m down to the finer graphs, potentially refining it at each level.

This process results in a partition for the original graph (Figure 8). The hope is that multilevel techniques will reduce the time required to compute partitions without sacrificing quality. In practice, the use of multilevel techniques has proven not only to accelerate partition generation, but also to produce better partitions than traditional single level techniques [4, 22, 25].

6.1 Early Approaches

Barnard and Simon performed some of the initial multilevel graph partitioning work in order to accelerate the computation of the Fielder vector for RSB [2]. The result is an algorithm known as *multilevel recursive spectral bisection* (MRSB). MRSB coarsens graphs using *maximal independent sets* to recursively eliminate vertices and edges from the original graph. Having constructed the coarsest graph, MRSB computes a Fielder vector for it using the Lanczos algorithm, as in RSB. It then *expands* the Fielder vector

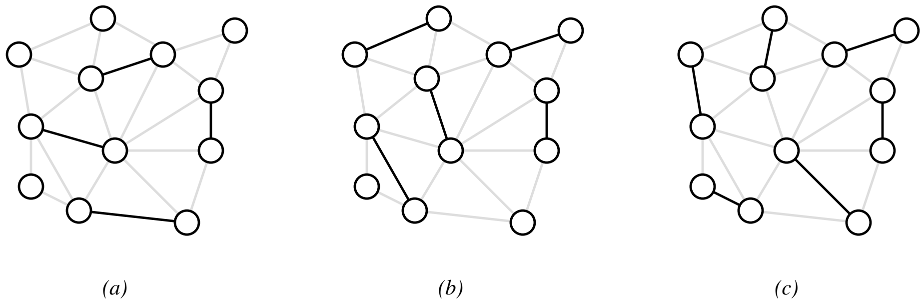


Figure 10: (a) A matching for the graph of Figure 1. No two edges are incident on the same vertex. (b) A maximal matching for the same graph. No more edges can be selected without breaking the matching. (c) A maximum matching. No matching could contain more edges since all the vertices are matched.

for the next finer graph and refines it using *Rayleigh quotient iteration* (RQI) [37]. This interpolation step is then repeated for the finer graphs until the original graph G is reached. The resulting Fielder vector is used to partition G as in RSB. Figure 9 illustrates a partition of a coarse graph and its refinement on the original graph. Barnard and Simon find that MRSB tends to result in an order of magnitude speedup over RSB, yielding partitions that are comparable in quality.

Hendrickson and Leland improve upon this work by simplifying the approach in a fundamental way [22]. Rather than deal with the numerical overhead of propagating a Fielder vector down the hierarchy of graphs, they opt instead to interpolate the partition itself down the hierarchy, yielding a conceptually simpler approach. Their algorithm uses the notion of *maximal matchings* to coarsen graphs. A *matching* on a graph G is simply a subset of E in which no two edges have a vertex in common. A maximal matching is one in which no other edge from E can be added to the matching without some vertex being shared (Note that this differs from a *maximum matching* which is the largest possible matching for a given graph. See Figure 10). Maximal matchings can be computed using an simple $O(E)$ greedy algorithm.

Hendrickson and Leland construct their hierarchy of graphs by generating maximal matchings and then merging the matched vertices into new *multinodes*. They use vertex and edge weights to encode characteristics of the original graph in the contracted versions. For instance, whenever two vertices are

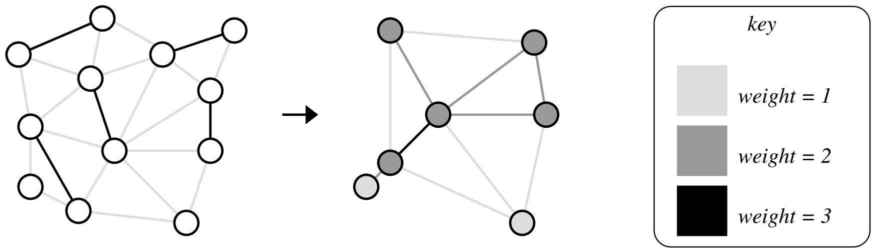


Figure 11: A graph contraction based on the matching of Figure 10(b). This contraction uses the vertex and edge weighting scheme of Hendrickson and Leland [22] to maintain information about the original graph (note the coloring of the contracted vertices and edges).

combined, the sum of their weights is assigned to the resulting multinode. Similarly, any edges that joined the vertices to a common node are combined into a single edge whose weight equals the sum of the original ones. Figure 11 shows an example. In this way, each coarse graph maintains global information about the original.

The benefits of this approach should be clear. A partition for the coarse graph can be computed much more quickly than for the original graph. Yet, this coarse partition maps directly to a partition on the original graph since each vertex of G is uniquely represented by a multinode in G_m. Moreover, the partition's edge cut and subset weights are identical for both graphs. Thus, since coarsening is cheap, the problem has been vastly simplified. Karypis and Kumar argue that a good bisection of a coarse graph constructed using this method can only be worse than a good bisection of the finer graph by a small factor [27].

To try and minimize this factor, Hendrickson and Leland use a modified version of KL/FM to refine the partition at every third step of the interpolation. Their experiments demonstrate that the multilevel technique generally produces partitions that are better than the corresponding single-level spectral algorithm and requires only a fraction of the time. Similar results were independently obtained by Bui and Jones, who implemented multilevel versions of the KL/FM and steepest descent algorithms [4].

6.2 Further Refinements

Karypis and Kumar build on the multilevel work of Hendrickson and Leland with the goal of finding a "best" algorithmic choice for each of the coarsening, partitioning, and refinement stages [25]. Although an exhaustive cross-product of all techniques would be infeasible, their study does an excellent job of examining each stage independently and measuring its impact on the overall partitioning time. Their goal is to find algorithms that represent a good tradeoff between running time and quality. For instance, rather than using a randomized technique for generating maximal matchings as the previous approaches had [22, 4], they suggest a strategy called *heavy edge matching* in which edges with higher weight are given priority for inclusion in the matching. The intuition behind this heuristic is that heavier edges will typically be disadvantageous to cut, and therefore collapsing them into a multinode will remove them from consideration in coarser graphs.

Another interesting result of the Karypis-Kumar study is that using spectral partitioning on the coarsest graph proves not only to be slower than greedier algorithms (as expected), but also results in partitions of significantly worse quality. They suggest that this surprising result stems from the Lanczos algorithm's failure to converge in its allotted iterations. This indicates that while spectral methods work well on large graphs whose complexity may foil greedier algorithms, their use on smaller graphs may be overkill.

The multilevel suite of choice according to Karypis and Kumar consists of: heavy edge matching for the contraction phase; the greedy graph growing algorithm for the partitioning phase; and a variation of Kernighan-Lin called *BKL(*,1)* for the refinement phase. BKL(*,1) only considers moving the boundary vertices and uses just a single iteration when the graph becomes sufficiently large (contains more than 2% of the vertices in the original graph). Further work by Karypis and Kumar accelerates the running time of multilevel recursive bisection by developing a multilevel p-way partitioning algorithm in which coarsening and refinement are performed a single time rather than at every step of the bisection [26].

7 Parallel Techniques

Since the graph partitioning algorithms described in this paper are used to distribute computations across a processor set, it seems only natural to take advantage of that parallel computing power to generate partitions more quickly. However, the multilevel techniques of the previous section can generate high-quality partitions for graphs with millions of edges in mere seconds. Thus, one has to wonder whether the effort of parallelizing these algorithms is worth the trouble. Even assuming that perfect speedups are achievable, would the benefits be worth the headaches of a parallel implementation? It seems unlikely.

Unfortunately, there are other more compelling reasons that motivate parallel solutions to the graph partitioning problem. One of these is that parallel computers have sufficient memory to handle large-scale computations whose workload graphs exceed the memory capacity of sequential machines. In addition, the workload for many graph-based applications changes dynamically: some algorithms refine the initial graph in areas of interest, generating more computation in that region. Other computations involve dynamic shifts in the amount of work at each vertex. For example, in particle simulations, the number of particles per vertex can fluctuate during the course of a run [3]. These runtime changes in the computation's workload result in the need for dynamic load balancing. Given the choice, it would be preferable to compute a repartitioning in-place rather than to ship the entire graph to a single processor and generate a new partition from scratch. Thus, parallel solutions must be considered.

7.1 Opportunities for Parallelism

Computing a p-way partition for a graph results in two opportunities for parallelism: first, the natural parallelism that results from the divide-and-conquer nature of recursive bisection; second, the opportunity to compute each bisection in parallel. For example, to compute a p-way partition using p processors, first all p processors would cooperate in generating a bisection. Then two groups of $p/2$ processors would cooperate on each subgraph to compute the quadrisection. This would continue until pairs of processors were computing their respective subsets. In his efforts to parallelize MRSB, Barnard describes this general algorithmic structure using *recursive asynchronous task teams* which recursively use smaller numbers of processors to solve smaller versions of the same problem [1].

In order to compute a partition in parallel, the graph must first be distributed across the processor set. Since this is an instance of graph partitioning in itself, the graph vertices are generally divided amongst the processors in an arbitrary manner. This results in a challenge common to all parallel partitioning algorithms: namely, the fact that referring to a vertex's neighbors is likely to require communication with other processors.

As a result, geometric algorithms tend to make ideal candidates for parallel implementations since they ignore a graph's edges, operating instead on the vertex coordinates. These algorithms therefore tend to be amenable to embarrassingly parallel implementations in which each processor computes over a fraction of V, using occasional global communications to compare notes. For example, Diniz *et al.* found the inertial bisection algorithm to be fast and trivially implementable in parallel [7]. Similarly, Miller *et al.* predict that their circle bisection algorithm will be reasonably efficient in parallel [16]. Unfortunately, the only published results that describe a parallel implementation of circle bisection give little indication of the algorithm's parallel performance [23].

7.2 Challenges to Parallelism

By way of contrast, edge-based algorithms tend to be difficult to parallelize. For example, KL/FM and the coarsening algorithms of multilevel techniques require vertices to compare information with their neighbors to update gains and compute matchings. Naive parallel implementations would tend to result in far too much interprocessor communication. Moreover, if multiple processors were to refine a partition or compute a matching in parallel, decisions which seem locally advantageous may inadvertently clash with one another, resulting in a poor overall solution.

As a result, the challenge to many partitioning algorithms is to utilize the available concurrency while minimizing the opportunities for processors to clash. One approach used by Diniz *et al.* to parallelize KL/FM only allows vertex swaps between paired sets of processors at any given point [7]. This results in a reasonably parallel solution. However, its partitions are significantly worse than those generated by sequential KL/FM, which benefits from its ability to access to the entire graph and to swap arbitrary vertices.

A more successful approach was used by Barnard and Karypis/Kumar to parallelize the coarsening stages of their respective multilevel algorithms [1, 28]. Both approaches use a clever parallel algorithm for maximal independent set creation developed by Luby [32]. Luby's algorithm assigns a random number to every vertex in the graph. Each vertex then checks its value against those of its neighbors, and if it has the smallest value, it includes itself in the maximal independent set I. This process is then repeated for all vertices that are neither in, nor adjacent to I until no more vertices can be added.

Barnard uses this approach to construct the maximal independent sets used in MRSB, forming a parallel version of the algorithm called PMRSB [1]. The remainder of the algorithm is relatively straightforward to parallelize: the Lanczos algorithm is run on a single processor, broadcasting the result to the others. BLAS-1 and BLAS-2 routines implement RQI for refining the Fielder vector. Barnard found that PMRSB generates partitions that are comparable to MRSB, yet runs about 140 times faster using 256 processors.

Karypis and Kumar use Luby's algorithm to construct a global coloring of the graph, such that no two connected vertices are the same color [28]. This coloring is then used to prevent neighboring vertices from interfering with one another by restricting operations to run on a single color at a time. For example, to compute a maximal matching of the graph, the colors are iterated through one at a time to prevent two adjacent vertices from adding themselves to the matching simultaneously. They implement a parallel version of KL/FM using the same strategy. Comparing their parallel multilevel p-way algorithm to the sequential version, the authors find that it produces partitions that are only slightly worse with speedups of about 14 to 35 on 128 processors.

One final parallel partitioning result that deserves mention is that of Walshaw, Cross, and Everett [46]. In computing partitions in parallel, their focus is on creating partitions that are not only fast to compute and of high quality, but which also require a minimal number of vertices to move from one processor to another. The motivation is obvious: if a computation's dynamic changes require a graph to be repartitioned, it would be preferable to adjust the existing partition rather than to compute a new partition from scratch so that data movement is minimized. Their experiments show that their technique does an excellent job of meeting these goals as compared to other partitioning algorithms.

8 Summary and Discussion

Globally-Oriented Algorithms In the last decade, the three major innovations in graph partitioning seem to have been the circle-based algorithm of Miller *et al.* [33, 16], the spectral algorithm of Pothen *et al.* [40], and the multilevel technique of Hendrickson and Leland [22]. All of these algorithms produce partitions of high quality, especially when combined with a refinement algorithm like KL/FM. To its disadvantage, the circle-based algorithm relies on geometric coordinates and therefore is not fully general. Spectral methods suffer from long runtimes without the benefit of multilevel or parallel techniques. In contrast, the multilevel strategy seems to have no disadvantages other than the challenges presented by an efficient parallelization.

One common theme in all of these innovations is their use of a global perspective on the graph when computing a partition. The circle-based algorithm uses the vertices' coordinates to compute a global centerpoint useful for partitioning. Spectral partitioning methods utilize properties of the Laplacian matrix that reflect a graph's global connectivity. Hendrickson and Leland use a graph contraction policy that preserves global information while eliminating edges and vertices from a graph. In contrast, most of the algorithms that produce lower-quality partitions (*e.g.*, variations on coordinate bisection and graph-walking) look at the graph in a far more restricted and localized manner.

One final observation on the global nature of these algorithms is that as good as their partitions are, they can always benefit from the use of a post-processing local refinement step using KL/FM or one of its many variations. This seems to reflect the fact that any tractable solution to an NP-hard problem can only consider a small subset of the exponentially large set of solutions. Using global information can result in a high-quality choice from this subset, but local adjustments can bring it even closer to the optimal solution.

Challenges to Evaluating Partitioning Algorithms In terms of selecting a "best" sequential partitioning algorithm, evidence suggests that the multilevel algorithm suggested by Karypis and Kumar is an excellent choice [25]. It uses the multilevel approach, benefiting from its speed and quality; it uses a fast greedy algorithm for the coarse partitioning, which has been shown to be effective; and it uses an intelligent local refinement technique during the interpolation phase.

One thing that is a bit unsatisfactory about this choice is that it provides little explanation for *why* the greedy graph partitioning algorithm is the right choice as the root partitioner. In fact, this is a problem that seems to be common to the majority of the work discussed in this paper: an algorithm is developed, usually based on theoretical principles or intuition; it is run against the other leading competitors; the authors find that it produces the best partition for 9 of 12 benchmark graphs considered; therefore it is classified as a "good" algorithm. It is rare to find any discussion characterizing the graphs on which the algorithm performed particularly well or particularly badly. Ideally, one would like some analysis (or even intuition) indicating whether a given algorithm prefers graphs with a specific edge density, a certain spatial density, some type of structure, etc. For the most part, no such characterizations are made.

One reason for this could certainly be that there simply are no discernible patterns. If true, then further analysis may simply be too difficult to seem worthwhile. Although the circle bisection and spectral algorithms are based on theory that places lower bounds on their separator sizes, these bounds are typically loose by nature (*e.g.*, the $d:1$ guarantee stemming from centerpoint separators). Furthermore, practical implementations are forced to cut corners that cause them to deviate from the theory (*e.g.*, the finite-precision computation of eigenvectors; or the nudging of geometric separators to ensure a bisection). Add to this the magnitude of the graphs being considered and the lack of an optimal solution for comparison, and suddenly, trying to understand why an algorithm works well or not seems like an extremely hard problem. Better to just run a couple dozen more trials.

As an illustration of this problem, consider the partitions shown in Color Plate 1, computed by Simon using recursive coordinate bisection, level-structure bisection, and spectral bisection [44]. Simon makes a valiant effort to characterize the partitions found by each algorithm for the four-element wing graph in column 1, which serves as a running example:

> *[Coordinate bisection] creates long, narrow, and disconnected domains. [Level-structure bisection] creates more compact domains, but their boundaries are "fuzzy", and sometimes they are disconnected. RSB creates well balanced, connected domains, which yield a visually most pleasing partitioning.*

Applied to the graph in question, these descriptions are quite reasonable and give the reader a sense that there *is* an intuitive aspect to the algorithms that relates their behavior to the partition sizes. A table listing edge cuts confirms Simon's qualitative reasoning.

However, when the same algorithms are applied to the more complex graph of a shuttle solid rocket motor (column 2 of Color Plate 1), the intuition falls apart—the spectral algorithm produces a partition that appears quite similar to that of the coordinate bisection. In addition, it results in disconnected subsets. In spite of these qualitative problems, the table of edge cuts confirms that the spectral algorithm is still much better than the other approaches (4k edges cut as compared to the 14k and 6k of the coordinate and level-structure algorithms). Simon is forced to admit that "the considerable quantitative differences are *not* obviously visible from the figures," and that "these pictures indicate that our visual perception may be inadequate." Understanding why an algorithm generates a good or bad partition is clearly a complex task.

Missing Data Points There are a few obvious avenues of research in graph partitioning that appear to have been neglected. Chief among these is whether or not recursive circle bisection can compete with today's top algorithms. It would appear that no work has been done to evaluate the use of recursive circle bisection as the coarse partitioning strategy in a multilevel algorithm. In fact, even refining the basic circle bisection algorithm using KL/FM seems like a promising approach that has been unexplored.

Another lingering question concerns the poor quality of the spectral algorithm in Karypis and Kumar's study, as compared to the greedy graph growing algorithm. Greedy graph growing has apparently never been evaluated in the single level domain of partitioning. Perhaps it ought to be since it outpartitions spectral methods in the multilevel domain. Otherwise, a more satisfactory explanation for why the spectral algorithm performs so poorly in the multilevel context is required.

Finally, the study of good parallel algorithms seems relatively young. Studies comparing the most recent techniques are scant. Furthermore, there seems to be a default assumption that parallelizing the best sequential algorithms will result in the best parallel algorithms, even though the poor speedups reported by Karypis and Kumar [28] hint that this may not be the case.

Workload Distribution Recent progress in graph partitioning has been very impressive. Large graphs are being partitioned faster and better than ever before. Graphs with millions of edges can be partitioned well in seconds. So is the problem of workload distribution solved? In the abstract sense, it appears that the answer is "yes." However, it seems that the mapping between the *abstraction* of graph partitioning and the *reality* of parallel computing has been neglected. Although the graph partitioning problem serves as a nice starting point for distributing workloads, it fails to describe many real-world issues that could impact application performance.

For example, minimizing the total amount of interprocessor communication is a valid goal, but neglects to balance that communication among the processors. Nor does it minimize the number of processors that each processor has to communicate with. Nor can it model network distance between processors for clusters of multiprocessors. Although some amount of flexibility is available in mapping a partition's subsets to the set of physical processors, exposing these issues to the partitioning algorithm could only improve its solutions. Hendrickson addresses some of these issues as well as several others in a recent paper that casts doubt on whether traditional graph partitioning is adequate [18]. Future work would be well-served by continuing to think about real-world issues while solving abstract problems. The work of Walshaw *et al.* [46] provides an excellent example of this by repartitioning graphs in a way that seeks to reduce the amount of data that must be shuffled.

Publicly-available Partitioners It should be noted that several partitioning packages are available online, which implement many of the algorithms described in this paper. The leading contenders seem to be the METIS and ParMETIS packages by Karypis and Kumar [29, 30] and the Jostle package by Walshaw *et al.* [47]. METIS and ParMETIS implement Karypis and Kumar's multilevel, p-way multilevel, and parallel p-way multilevel algorithms. Jostle also supports parallel partitioning, with Walshaw's emphasis on minimizing vertex movement. Chaco [20] is Hendrickson and Leland's package that contains implementations of their 4-way/8-way spectral algorithms, multilevel algorithms, and refinements to KL/FM. Other online packages worth investigating are Scotch [38] and Party [42].

Acknowledgements The author would like to thank Jim Fix for strategizing sessions and Wayne Wong for his editorial feedback. Additional thanks to Mike, Susannah, and Mom for providing encouragement in the form of "you're *still* not done with generals?" and to Larry for shunning such crude techniques. Soundtrack provided by the Dirty Three.

References

[1] Stephen T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, pages 602–625. ACM/IEEE, December 1995.

[2] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, April 1994. (also available in a more complete form as NASA Ames Research Center Technical Report RNR-92-033).

[3] Tim Bartel and Steve Plimpton. DSMC simulations of low-density fluid flow on MIMD supercomputers. *Computer Systems in Engineering*, 3(1–4):333–336, 1992.

[4] Thang N. Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[5] Feng Cao, John R. Gilbert, and Shang-Hua Teng. Partitioning meshes with lines and planes. Technical Report CSL-96-01, Xerox Palo Alto Research Center, January 1996.

[6] K. L. Clarkson, D. Eppstein, G. L. Miller, C. Sturtivant, and Shang-Hua Teng. Approximating center points with iterative radon points. *International Journal of Computational Geometry and Applications*, 6(3):357–77, September 1996.

[7] 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. SIAM, February 1995.

[8] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD–4(1):92–98, January 1985.

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

[10] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181. IEEE Computer Society Press, June 1982.

[11] M. Fielder. Algebraic connectivity of graphs. *Czechoslovak Math. J.*, 23:298–305, 1973.

[12] M. Fielder. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czechoslovak Math. J.*, 25:619–633, 1975.

[13] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[14] Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10:345–363, 1973.

[15] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[16] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 418–427. IEEE Computer Society Press, April 1995.

[17] Michael T. Heath and Padma Raghavan. A cartesian parallel nested dissection algorithm. *SIAM Journal on Matrix Analysis and Applications*, 16(1):235–253, January 1995.

[18] Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, LNCS. Springer-Verlag, August 1998.

[19] Bruce Hendrickson and Robert Leland. An improved spectral load balancing method. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 953–961. SIAM, 1993.

[20] Bruce Hendrickson and Robert Leland. The chaco user's guide: Version 2.0. Technical Report SAND95–2344, Sandia National Laboratories, July 1995. (information about obtaining Chaco is available at http://www.cs.sandia.gov/~bahendr/partitioning.html).

[21] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, March 1995.

[22] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, pages 626–657. ACM/IEEE, December 1995.

[23] Yu Charlie Hu, Shang-Hua Teng, and S. Lennart Johnsson. A data-parallel implementation of the geometric partitioning algorithm. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.

[24] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, November–December 1989.

[25] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *to appear in SIAM Journal on Scientific Computing*.

[26] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *to appear in Journal of Parallel and Distributed Computing*.

[27] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, pages 658–677. ACM/IEEE, December 1995. (a more complete version appears at http://www-users.cs.umn.edu/~karypis/metis/publications/main.html).

[28] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96 Conference Proceedings*. ACM/IEEE, November 1996. (a more complete version is available at http://www-users.cs.umn.edu/~karypis/metis/publications/main.html).

[29] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices (version 3.0.3)*. University of Minnesota / Army HPC Research Center, November 1997. (the METIS homepage is at http://www-users.cs.umn.edu/˜karypis/metis/metis/main.shtml).

[30] George Karypis, Kirk Schloegel, and Vipin Kumar. *ParMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library*. University of Minnesota, version 2.0 edition, July 1997. (the ParMETIS homepage is at http://www-users.cs.umn.edu/˜karypis/metis/parmetis/main.shtml).

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

[32] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, November 1986.

[33] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA volumes in mathematics and its applications*, pages 57–84. Springer-Verlag, 1993.

[34] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Geometric separators for finite element meshes. *Siam Journal on Scientific Computing*, 19(2):364–386, March 1998.

[35] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 538–547. IEEE Computer Society Press, October 1991.

[36] B. Mohar. The laplacian spectrum of graphs. In *Sixth International Conference on Theory and Applications of Graphs*, 1988.

[37] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, 1980.

[38] Francois Pellegrini. Scotch 3.1 user's guide. Technical Report 1137-96, University of Bordeaux, June 1997. (the Scotch homepage is at http://www.labri.u-bordeaux.fr/Equipe/ALiENor/membre/pelegrin/scotch/).

[39] Alex Pothen. Graph partitioning algorithms with applications to scientific computing. In David E. Keyes, Ahmed Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms*, volume 4 of *ICASE/LaRC Interdisciplinary Series in Science and Engineering*. Kluwer Academic Press, January 1997.

[40] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, July 1990.

[41] Alex Pothen, Horst D. Simon, Lie Wang, and Stephen T. Barnard. Towards a fast implementation of spectral nested dissection. In *Proceedings of Supercomputing '92*, pages 42–51. ACM/IEEE, November 1992.

[42] Robert Preis and Ralf Dickmann. The party partitioning–library, user guide—version 1.1. Technical Report tr-rsfb-96-024, University of Paderborn, September 1996. (the Party homepage is at http://www.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html).

[43] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. In *Proceedings of the 1997 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 731–737. IEEE Computer Society Press, June 1997.

[44] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2–3):135–148, 1991.

[45] Horst D. Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, September 1997.

[46] C. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In B. H. V. Topping, editor, *Advances in Computational Mechanics with Parallel and Distributed Processing*, pages 97–103, Edinburgh, April/May 1997. Civil-Comp Press.

[47] Chris Walshaw. *The Jostle User Manual: Version 2.0*. University of Greenwich, July 1997. (the Jostle homepage is at http://www.gre.ac.uk/˜c.walshaw/jostle/).

[48] Roy D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, October 1991.