# *Graph Partitioning for High-Performance Scientific Simulations*

Kirk Schloegel, George Karypis, and Vipin Kumar
Army HPC Research Center
Dept. of Computer Science and Engineering,
University of Minnesota
Minneapolis, Minnesota

Technical Report: TR 00-018

To be included in *CRPC Parallel Computing Handbook*
J. Dongarra, I. Foster, G. Fox, K. Kennedy, L. Torczon, and A. White, editors.
Morgan Kaufmann, 2000.

***Early Draft***

# *Contents*

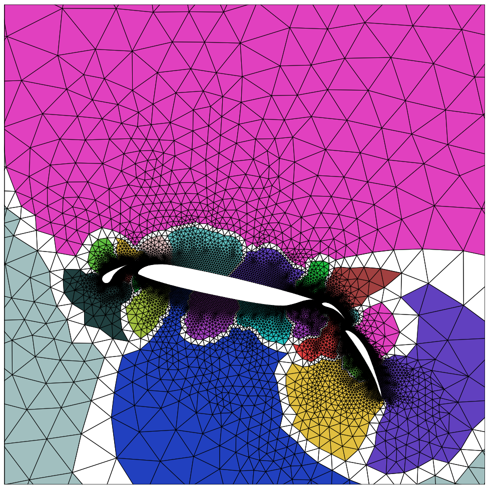


Figure 1: A partitioned 2D irregular mesh of an airfoil. The shading of a mesh element indicates the processor to which it is mapped.

## 0.1 Introduction

Algorithms that find good partitionings of unstructured and irregular graphs are critical for the efficient execution of scientific simulations on high-performance parallel computers. In these simulations, computation is performed iteratively on each element (and/or node) of a physical two- or three-dimensional mesh. Information is then exchanged between adjacent mesh elements. For example, computation is performed on each triangle of the two-dimensional irregular mesh shown in Figure 1. Then information is exchanged for every face between adjacent triangles. Efficient execution of such simulations on distributed-memory machines requires a mapping of the computational mesh onto the processors that equalizes the number of mesh elements assigned to each processor and minimizes the interprocessor communication required to perform the information exchange between adjacent elements. Such a mapping is commonly found by solving a graph partitioning problem. For example, a graph partitioning algorithm was used to decompose the mesh in Figure 1. Here, the mesh elements have been shaded to indicate the processor to which they have been mapped. Simulations performed on shared-memory multiprocessors also benefit from partitioning, as this increases data locality, and so, leads to better cache performance.

In many scientific simulations, the structure of the computation evolves from time step to time step. These simulations require decompositions of the mesh prior to the start of the simulation (as described above) and periodic load balancing during the course of the simulation. Other classes of simulations (i.e., multiphase simulations) consist of a number of computational phases separated by synchronization steps. These require that each of the phases be individually load balanced. Still other scientific simulations model multiple physical phenomenon (i.e., multiphysics simulations) or employ multiple meshes simultaneously (i.e., multimesh simulations). These impose additional requirements that the partitioning algorithm must take into account. Traditional graph partitioning algorithms are not adequate to ensure the efficient execution of these classes of simulations on high-performance parallel computers. Instead, generalized graph partitioning algorithms have been developed for such simulations.

This chapter presents an overview of graph partitioning algorithms used for scientific simulations on high-performance parallel computers. Recent developments in graph partitioning for adaptive and dynamic simulations, as well as partitioning algorithms for sophisticated simulations such as multiphase, multiphysics, and multimesh computations are also discussed. Specifically, Section 0.2 presents the graph partitioning formulation used to model the problem of mapping computational meshes onto processors. Section 0.3 describes numerous static graph partitioning algorithms. Section 0.4 discusses the adaptive graph partitioning problem and describes a number of repartitioning schemes. Section 0.5 discusses the issues involved with the parallelization of static and adaptive graph partitioning schemes. Section 0.6 describes a number of

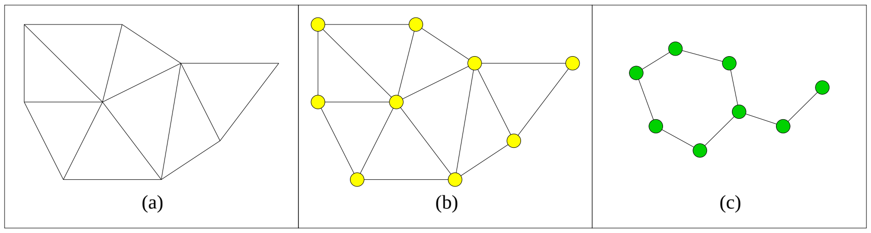


Figure 2: A 2D irregular mesh (a) and corresponding graphs (b) and (c). The graph in (b) models the connectivity between the mesh nodes. The graph in (c) models the adjacency of the mesh elements.

important types of applications for which the traditional graph partitioning problem is inadequate. This section also describes generalizations of the graph partitioning problem that are able to effectively model these applications as well as algorithms for computing partitionings based on these new formulations. Finally, Section 0.7 presents concluding remarks, discusses areas of future research, and charts the functionality of a number of publicly available graph partitioning software packages.

## 0.2 Modeling Mesh-based Computations as Graphs

In order to compute a mapping of a mesh onto a set of processors via graph partitioning, it is first necessary to construct the graph that models the structure of the computation. In general, computation of a scientific simulation can be performed on the mesh nodes, the mesh elements, or both of these. If the computation is mainly performed on the mesh nodes, then this graph is straightforward to construct. A vertex exists for each mesh node, and an edge exists on the graph for each edge between the nodes. We refer to this as the *node* graph. However, if the computation is performed on the mesh elements, then the graph is such that each mesh element is modeled by a vertex, and an edge exists between two vertices whenever the corresponding elements share an edge (in two dimensions) or a face (in three dimensions). We refer to this as the *dual* graph. Figure 2 illustrates a 2D example. Figure 2(a) shows a finite-element mesh. Figure 2(b) shows the corresponding node graph. Figure 2(c) shows the dual graph that models the adjacencies of the mesh elements. Partitioning the vertices of these graphs into $k$ disjoint subdomains provides a mapping of either the mesh nodes or the mesh elements onto $k$ processors. If the partitioning is computed such that each subdomain has the same number of vertices, then each processor will have an equal amount of work during parallel processing. The total volume of communications incurred during this parallel processing can be estimated by counting the number of edges that connect vertices in different subdomains. Therefore, a partitioning should be computed that minimizes this metric (which is referred to as the *edge-cut*).

The objective of the graph partitioning problem is to compute just such a partitioning (i.e., one that balances the subdomains and minimizes the edge-cut). More formally, the graph partitioning problem is as follows. Given a weighted, undirected graph $G = (V, E)$, for which each vertex and edge has an associated weight, the $k$-way graph partitioning problem is to split the vertices of $V$ into $k$ disjoint subsets (or *subdomains*) such that each subdomain has roughly an equal amount of vertex weight (referred to as the *balance constraint*), while minimizing the sum of the weights of the edges whose incident vertices belong to different subdomains (i.e., the edge-cut).

In some cases, it is beneficial to compute partitionings that assign each subdomain a specified amount of vertex weight. This may be necessary for scientific simulations performed on a cluster of heterogeneous workstations. The subdomain weights should result in more work being assigned to the faster machines and less work to the slower machines. Subdomain weights can be specified by using a vector of size $k$ in which each element of the vector indicates the fraction of the total vertex weight that the corresponding subdomain should contain. In this case, the graph partitioning problem is to compute a partitioning that splits the vertices into $k$ disjoint subdomains such that each subdomain has the specified fraction of total vertex weight and such that the edge-cut is minimized.

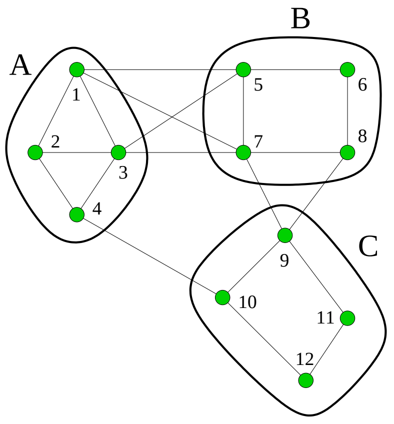


Figure 3: A partitioned graph with an edge-cut of seven. Here, nine communications are incurred during parallel processing.

It is important to note that the edge-cut metric is only an approximation of the total communications volume incurred by parallel processing [33]. It is not a precise model of this quantity. Consider the example in Figure 3. Here, three subdomains, $A$, $B$, and $C$ are shown. The edge-cut of the (three-way) partitioning is seven. During parallel computation, the processor corresponding to subdomain $A$ will need to send the data for vertices 1 and 3 to the processor corresponding to subdomain $B$ and the data for vertex 4 to the processor corresponding to subdomain $C$. Similarly, $B$ needs to send the data for 5 and 7 to $A$ and the data for 7 and 8 to $C$. Finally, $C$ needs to send the data for 9 to $B$ and the data for 10 to $A$. This equals nine units of data to be sent, while the edge-cut is seven. Edge-cut and total communication volume are not the same because the edge-cut counts every edge cut, while data is required to be sent only one time if two or more edges of a single vertex are cut by the same subdomain. (This is the case, for example, for vertex 3 and subdomain $B$ in Figure 3.) It should also be noted that total communication volume alone cannot accurately predict interprocessor communication overhead. A more precise measure is the maximum time required by any of the processors to perform communication (assuming that computation and communication occur in alternating phases). This depends on a number of factors, including the amount of data to be sent out of any one processor, as well as the number of processors with which a processor must communicate. In particular, on message-passing architectures, minimizing the maximum number of message start-ups that any one processor must perform can sometimes be more important than minimizing the communications volume [35]. Nevertheless, there still tends to be a strong correlation between edge-cuts and interprocessor communication costs for graphs of uniform degree (i.e., graphs in which most vertices have about the same number of edges). This is a typical characteristic of graphs derived from scientific simulations. Therefore, the min-cut partitioning problem is a reasonable model for minimizing the interprocessor communications of parallel scientific simulations.

**Computing a $k$-way Partitioning via Recursive Bisection** Graphs are frequently partitioned into $k$ subdomains by recursively computing two-way partitionings (i.e., *bisections*) of the graph [6]. This method requires the computation of $k-1$ bisections. If $k$ is not a power of two, then for each bisection, the appropriate subdomain weights need to be specified in order to ensure that the resulting $k$-way partitioning is balanced.

For a large class of graphs derived from scientific simulations, recursive bisection algorithms are able to compute $k$-way partitionings that are within a constant factor of the optimal solution [86]. Furthermore, if the balance constraint is sufficiently relaxed, then recursive bisection methods can be used to compute $k$-way partitionings that are within $\log p$ of the optimal for all graphs [86]. Since the direct computation of a good $k$-way partitioning is harder in general than the computation of a good bisection (although both problems are NP-complete), recursive bisection has become a widely used technique.

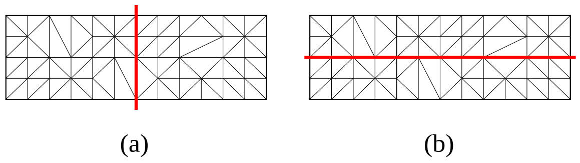


Figure 4: Two mesh bisections normal to the coordinate axes. In (a) the mesh is bisected normal to the $x$-axis. In (b) the mesh is bisected normal to the $y$-axis. The subdomain boundary in (a) is smaller than that in (b).

## 0.3 Static Graph Partitioning Techniques

The graph partitioning problem is known to be NP-complete. Therefore, in general it is not possible to compute optimal partitionings for graphs of interesting size in a reasonable amount of time. This fact, combined with the importance of the problem, has led to the development of several heuristic approaches [1, 2, 5, 6, 9, 13, 16, 20, 22, 23, 25, 26, 27, 28, 30, 31, 37, 38, 45, 49, 55, 59, 61, 62, 67, 69, 74, 75, 77, 85, 96, 106]. These can be classified as either geometric techniques [6, 23, 31, 59, 62, 67, 69, 77], combinatorial techniques [1, 2, 16, 20, 22, 25, 28, 55], spectral techniques [37, 38, 74, 75, 85], combinatorial optimization techniques [5, 27, 106], or multilevel methods [9, 13, 26, 30, 45, 49, 61, 96]. In this section, we discuss several of these classes and describe the important schemes from them.

### 0.3.1 Geometric Techniques

Geometric techniques [6, 23, 31, 59, 62, 67, 69, 77] compute partitionings based solely on the coordinate information of the mesh nodes. Since these techniques do not consider the connectivity between the mesh elements, there is no concept of edge-cut here. In order to minimize the interprocessor communications incurred due to parallel processing, geometric schemes are usually designed to minimize a related metric, such as the number of mesh elements that are adjacent to nonlocal elements (i.e., the size of the subdomain boundary). Usually, these techniques partition the mesh elements directly, rather than the graphs that model the structures of the computations. Because of this distinction, they are often referred to as *mesh-partitioning* schemes.

Geometric techniques are applicable only if coordinate information exists for the mesh nodes. This is usually true for meshes used in scientific simulations. Even if the mesh is not embedded in a $k$-dimensional space, there are techniques that are able to compute node coordinates automatically, based on the connectivity of the mesh elements [29]. Typically, geometric partitioners are extremely fast. However, they tend to compute partitionings of lower quality than schemes that take the connectivity of the mesh elements into account. For this reason, multiple trials are usually performed, with the best partitioning of these being selected.

**Coordinate Nested Dissection** Coordinate Nested Dissection (CND) (also referred to as Recursive Coordinate Bisection) is a recursive bisection scheme that attempts to minimize the boundary between the subdomains (and therefore, the interprocessor communications) by splitting the mesh in half normal to its longest dimension. Figure 4 illustrates how this works. Figure 4(a) gives a mesh bisected normal to the $x$-axis. Figure 4(b) gives the same mesh bisected normal to the $y$-axis. The subdomain boundary in Figure 4(a) is much smaller than that in Figure 4(b). This is because the mesh is longer in the direction of the $x$-axis than in the direction of $y$-axis.

The CND algorithm works as follows. The centers of mass of the mesh elements are computed, and these are projected onto the coordinate axis that corresponds to the longest dimension of the mesh. This gives an ordering of the mesh elements. (Note that this scheme can result in multiple mesh elements being projected onto the same point along the selected dimension. Such "ties" in ordering can be broken arbitrarily.) The ordered list is then split in half to produce a bisection of the mesh elements[1]. Each subdomain can then be

[1]Alternatively, the mesh *nodes* can be ordered and split in half instead of the mesh elements.

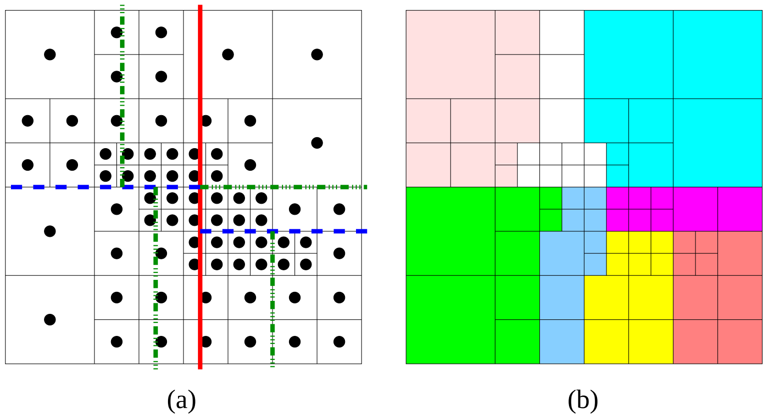


Figure 5: An eight-way partitioning of a mesh computed by a CND scheme. First, the solid bisection was computed. Then the dashed bisections were computed for each of the subdomains. Finally, the dashed-and-dotted bisections were computed. The centers of mass of the mesh elements are shown in (a). The mesh elements are shaded in (b) to indicated their subdomains.

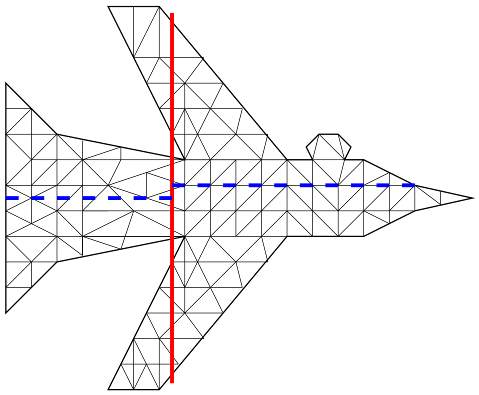


Figure 6: A four-way partitioning computed by a CND scheme. First, the solid bisection was computed. Then the dashed bisections were computed for each of the subdomains. The upper- and lower-left subdomains are disconnected.

recursively subdivided by the same technique [6]. Figure 5 illustrates an eight-way partitioning computed by this method. Figure 5(a) shows the centers of mass of the mesh elements and the computed recursive bisections. First, the solid line bisected the entire mesh. Then, the dashed lines bisected the two subdomains. Finally, the dashed-and-dotted lines bisected the resulting subdomains. Figure 5(b) shows the mesh elements shaded according to their subdomains.

The CND scheme is extremely fast, requires little memory, and is easy to parallelize. In addition, partitionings obtained by this scheme can be described quite compactly (just by the splitters used at each node of the recursive bisection tree). However, partitionings computed via CND tend to be of low quality. Furthermore, for complicated geometries CND tends to produce partitionings that contain disconnected subdomains. Figure 6 gives an example of this. Here, the upper- and lower-left subdomains are both disconnected. Several variations of CND have been developed that attempt to address its disadvantages [31]. However, even the most sophisticated variants tend to produce worse quality partitionings than more sophisticated schemes.

**Recursive Inertial Bisection** The CND scheme can only compute bisections that are normal to one of the coordinate axes. In many cases, this restriction can limit the quality of the partitioning. Figure 7 gives

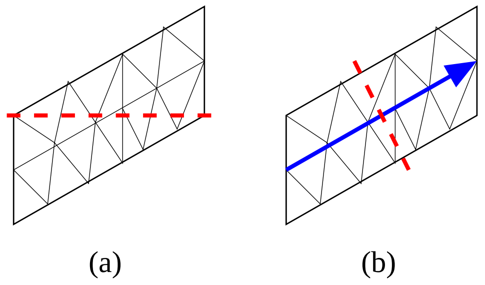


Figure 7: Bisection for a mesh are computed by the CND and RIB schemes. In (a) the mesh is bisected by the CND scheme. In (b) the mesh is bisected by the RIB scheme. This scheme results in a significantly smaller subdomain boundary.

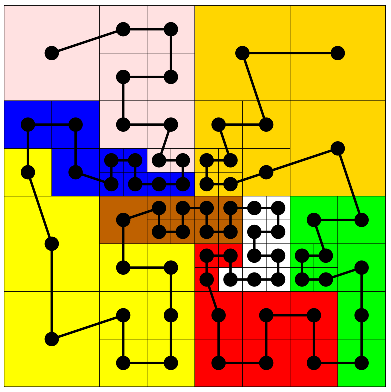


Figure 8: A Peano-Hilbert space-filling curve is used to order the mesh elements. The eight-way partitioning that is produced by this ordering is shown.

an example. The mesh in Figure 7(a) is bisected normal to the longest dimension of the mesh. However, the subdomain boundary is still quite long. This is because the mesh is oriented at an angle to the coordinate axes. Taking this angle into account when orienting the bisection can result in a smaller subdomain boundary. One way to do so is to treat the mesh elements as point masses and to compute the principal inertial axis of the mass distribution. If the mesh is convex, then this axis will align with the overall orientation of the mesh. A bisection line that is orthogonal to this will often result in a small subdomain boundary, as the mesh will tend to be thinnest in this direction [73].

The Recursive Inertial Bisection (or RIB) algorithm improves upon the CND scheme by making use of this idea as follows. The inertial axis of the mesh is computed, and an ordering of the elements is produced by projecting their centers of mass onto this axis. The ordered list is then split in half to produce a bisection. The scheme can be applied recursively to produce a $k$-way partitioning [62]. As an example, the bisection of the mesh in Figure 7(b) is computed using the RIB algorithm. The solid arrow indicates the inertial axis of the mesh. The dashed line is the bisection. Here, the subdomain boundary is much smaller than that produced by the CND scheme in Figure 7(a).

**Space-filling Curve Techniques** The CND and RIB algorithms find orderings of the mesh elements and then split the ordered list in half to produce a bisection. In these schemes, orderings are computed by projecting the elements onto either the coordinate or inertial axes. A disadvantage of these techniques is that orderings are computed based on a single dimension at a time. A scheme that considers more than one dimension may produce better partitionings.

One such method orders the mesh elements according to the positions of their centers of mass along a space-filling curve [65, 67, 69, 103] (or a related self-avoiding walk [32]). Space-filling curves are continuous curves

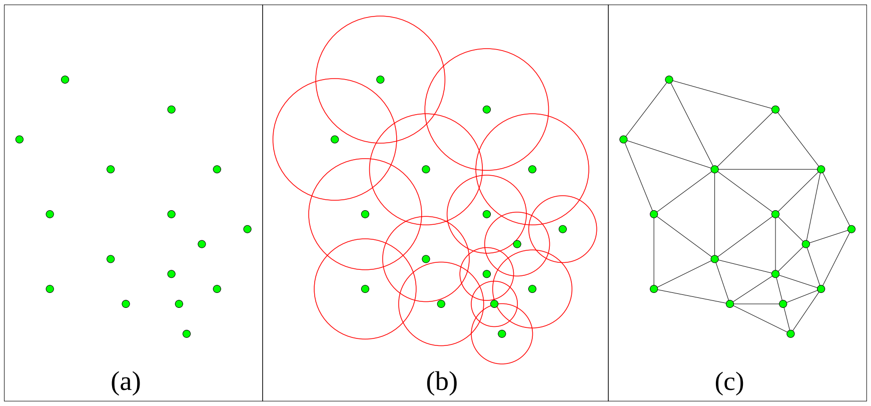


Figure 9: The nodes of a finite-element mesh (a). A three-ply neighborhood systems for the nodes (b). The (1, 3)-overlap graph for the mesh (c).

that completely fill higher-dimensional spaces such as squares or cubes. A number of such curves have been defined that fill space in a locality-preserving way (e.g., Peano-Hilbert curves [39]). These produce orderings of mesh elements that have the desirable characteristic that elements that are near to each other in space are likely to be ordered near to each other as well. After the ordering is computed, the ordered list of mesh elements is split into $k$ parts resulting in $k$ subdomains. Figure 8 illustrates a space-filling curve method for computing an eight-way partitioning of a quad-tree mesh.

Space-filling curve partitioners are fast and generally produce partitionings of somewhat better quality than either the CND or RIB schemes. They tend to work particularly well for classes of simulations in which the dependencies between the computational nodes are governed by their spatial proximity to one another as in $n$-body computations using hierarchical methods [103].

**Sphere-cutting Approach** Miller, Teng, Thurston, and Vavasis [59] proposed a new class of graphs, called *overlap* graphs, that contains all *well-shaped* meshes, as well as all planar graphs. Meshes are considered well shaped if the angles and/or aspect ratios of their elements are bounded within some values. Most of the meshes that are used in scientific simulations are well shaped according to this definition. Miller, et al., proved that overlap graphs have $O(n^{(d-1)/d})$ vertex separators. In doing so, they extended results by Lipton and Tarjan [57] and others [60]. Note that a vertex separator is a set of vertices that, if removed, splits the graph into two roughly equal-sized subgraphs, such that no edge connects the two subgraphs. That is, instead of partitioning the graph between the vertices (and so cutting edges), the graph is partitioned along the vertices. For this formulation, the sum weight of the separator vertices should be minimized.

Miller, et al., used the concept of a *neighborhood system* to define an overlap graph. A $k$-ply neighborhood system is a set of $n$ spheres in a $d$-dimensional space such that no point in space is encircled by more than $k$ of the spheres. An ($\alpha$, $k$)-overlap graph contains a vertex for each sphere, with an edge existing between two vertices if the corresponding spheres intersect when the smaller of them is expanded by a factor of $\alpha$. Figure 9 illustrates these concepts. Figure 9(a) shows a set of points in a two-dimensional space. Figure 9(b) shows a three-ply neighborhood system for these points. Figure 9(c) shows the (1, 3)-overlap graph constructed from this neighborhood system.

Gilbert, Miller, and Teng [23] describe an implementation of a geometric bisection scheme based on these results. This scheme projects each vertex of a $d$-dimensional ($\alpha$, $k$)-overlap graph onto the unit ($d$+1)-dimensional sphere that encircles it. A random great circle of the sphere has a high probability of splitting the vertices into three sets $A$, $B$, and $C$, such that no edge joins $A$ and $B$, $A$ and $B$ each have at most $\frac{d+1}{d+2}n$ vertices, and $C$ has only $O(\alpha k^{1/d} n^{(d-1)/d})$ vertices [59]. Therefore, by selecting a few great circles at random and picking the best separator from these, the algorithm can compute a vertex separator of guaranteed quality (in asymptotic terms) with high probability.

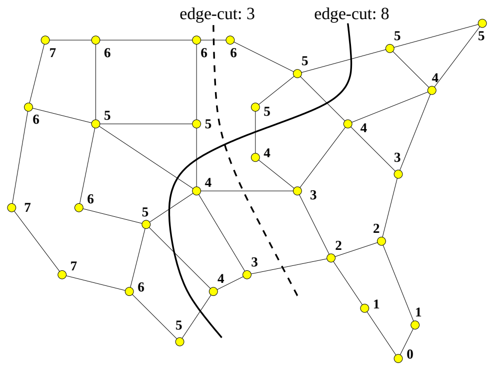


Figure 10: A graph partitioned by the LND algorithm. The vertex in the extreme bottom-right was selected and labeled zero. Then, the vertices were labeled in a breadth-first manner according to how far they are from the zero vertex. After half of the vertices had been labeled, a bisection (solid line) was constructed such that the labeled vertices are in one subdomain and the unlabeled vertices are in another subdomain. This figure also shows a higher-quality bisection (dashed line) for the same graph.

The sphere-cutting scheme is unique among those described in this chapter in that it guarantees the quality of the computed bisection for well-shaped meshes. But it is not guaranteed to compute perfectly balanced bisections. It is proven in [59] that the larger subdomain will contain no more than $\frac{d+1}{d+2}n$ vertices. However, experiments cited in [23] on a small number of test graphs indicate that, in three dimensions, splits as bad as $2:1$ are rare, and most are within twenty percent. The authors of [23] suggest a modification of the scheme that will result in balanced bisections by shifting the separating plane normal to its orientation.

### 0.3.2 Combinatorial Techniques

When computing a partitioning, geometric techniques attempt to group together vertices that are spatially near to each other, whether or not these vertices are highly connected. Combinatorial partitioners, on the other hand, attempt to group together highly connected vertices, whether or not these are near to each other in space. That is, combinatorial partitioning schemes compute a partitioning based only on the adjacency information of the graph; they do not consider the coordinates of the vertices. For this reason, the partitionings produced typically have lower edge-cuts and are less likely to contain disconnected subdomains than partitionings produced by geometric schemes. However, combinatorial techniques tend to be slower than geometric partitioning techniques and are not as amenable to parallelization.

**Levelized Nested Dissection** A partitioning will have a low edge-cut if adjacent vertices are usually in the same subdomain. The Levelized Nested Dissection (LND) algorithm attempts to put connected vertices together. It starts with a subdomain containing a single vertex and incrementally adds adjacent vertices [22].

More precisely, the LND algorithm works as follows. An initial vertex is selected and assigned the number zero. Then all of the vertices that are adjacent to the selected vertex are assigned the number one. Next, all of the vertices that are not assigned a number and are adjacent to any vertex that has been assigned a number are assigned that number plus one. This process continues until half of the vertices have been assigned a number. At this point, the algorithm terminates. The vertices that have been assigned numbers are in one subdomain, and the vertices that have not been assigned numbers are in the other subdomain. Figure 10 illustrates the LND algorithm. It shows the numbering starting with the extreme lower-right vertex. Here, the solid line shows a bisection with an edge-cut of eight computed by the LND algorithm.

This scheme tends to perform better when the initial seed is a pseudo-peripheral vertex (i.e., one of the pairs of vertices that are approximately the greatest distance from each other in the graph) as in Figure 10. Such a vertex can be found by a process that is similar to the LND algorithm. A random vertex is initially

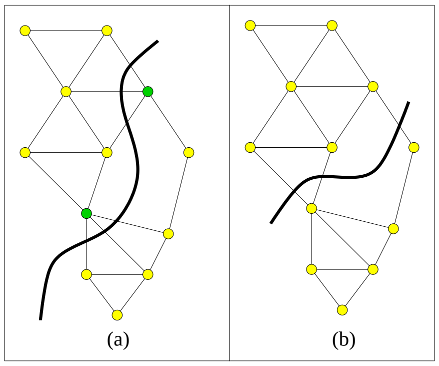


Figure 11: A bisection of a graph refined by the KL algorithm. The two shaded vertices will be swapped by the KL algorithm in order to improve the quality of the bisection (a). The resulting bisection is shown in (b).

selected to start the numbering of vertices. Here, all of the vertices are numbered. The vertex (or one of the vertices) with the highest number is likely to be in a corner of the graph. This vertex can be used either as an input to find another corner vertex (at the other end of the graph) or as the seed vertex for the LND scheme.

The LND algorithm ensures that at least one of the computed subdomains is connected (as long as the input graph is fully connected). It tends to produce partitionings of comparable or better quality than geometric schemes. However, even with a good seed vertex, the LND algorithm can sometimes produce poor quality partitionings. For example, the graph in Figure 10 contains a natural bisector shown by the dashed line. However, the LND algorithm was unable to find this bisection. For this reason, multiple trials of the LND algorithm are often performed, and the best partitioning from these is selected. Several variations and improvements of levelized nested dissection schemes are studied in [12, 25, 79].

**Kernighan-Lin / Fiduccia-Mattheyses Algorithm (KL/FM)** Closely related to the graph partitioning problem is that of partition refinement. Given a graph with a suboptimal partitioning, the problem is to improve the partition quality while maintaining the balance constraint. This differs from the graph partitioning problem only in that it requires an initial partitioning of the graph. Indeed, a refinement scheme can be used as a partitioning scheme simply by using a random partitioning as its input.

Given a bisection of a graph that separates the vertices into sets $A$ and $B$, a powerful means of refining the bisection is to find two equal-sized subsets, $X$ from $A$ and $Y$ from $B$, such that swapping $X$ to $B$ and $Y$ to $A$ yields the greatest possible reduction in the edge-cut. This type of swapping can be repeated until no further improvement is possible [55]. However, the problem of finding optimal sets $X$ and $Y$ is intractable itself (just like the graph partitioning problem). For this reason, Kernighan and Lin [55] developed a greedy method of finding and swapping near-optimal sets $X$ and $Y$ (referred to as Kernighan-Lin or KL refinement).

The KL algorithm consists of a small number of passes through the vertices. During each pass, the algorithm repeatedly finds a pair of vertices, one from each of the subdomains, and swaps their subdomains. The pairs are selected so as to give the maximum improvement in the quality of the bisection (even if this improvement is negative). Once a pair of vertices has been moved, neither is considered for movement in the rest of the pass. When all of the vertices have been moved, the pass ends. At this point, the state of the bisection at which the minimum edge-cut was achieved is restored. (That is, all vertices that were moved after this point are moved back to their original subdomains.) Another pass of the algorithm can then be performed by using the resulting bisection as the input. The KL algorithm usually takes a small number of such passes to converge. Each pass of the KL algorithm takes $O(|V|^2)$. Figure 11 illustrates a single swap made by the KL algorithm. In Figure 11(a), the two dark grey vertices are selected to switch subdomains. Figure 11(b) shows the bisection after this swap is made.

Fiduccia and Mattheyses present a modification of the KL algorithm [20] (called Fiduccia-Mattheyses or FM refinement) that improves its runtime without significantly decreasing its effectiveness (at least with respect to graphs arising in scientific computing applications). This scheme differs from the KL algorithm in that it moves *only a single vertex* at a time between subdomains instead of swapping *pairs* of vertices. The FM algorithm makes use of two priority queues (one for each subdomain) to determine the order in which vertices are examined and moved. As in KL, the FM algorithm consists of a number of passes through the vertices. Prior to each pass, the *gain* of every vertex is computed (i.e., the amount by which the edge-cut will decrease if the vertex changes subdomains). Then it is placed into the priority queue that corresponds to its current subdomain and ordered according to its gain. During a pass, the vertices at the top of each of the two priority queues are examined. If the top vertex in only one of the priority queues is able to switch subdomains while still maintaining the balance constraint, then that vertex is moved to the other subdomain. If the top vertices of both of the priority queues can be moved while maintaining the balance, then the vertex that has the highest gain among these is moved. Ties are broken by selecting the vertex that will most improve the balance. When a vertex is moved, it is removed from the priority queue and the gains of its adjacent vertices are updated. (Therefore, these vertices may change their positions in the priority queue.) The pass ends when neither priority queue has a vertex that can be moved. At this point, the highest quality bisection that was found during the pass is restored. With the use of appropriate data structures, the complexity of each pass of the FM algorithm is $O(|E|)$.

KL/FM-type algorithms are able to escape from some types of local minima because they explore moves that temporarily increase the edge-cut. Figure 12 illustrates this process. Figure 12(a) shows a bisection of a graph with an edge-cut of six. Here, the weights of the vertices and edges are one. There are twenty vertices in the graph. Therefore, a perfectly balanced bisection will have subdomain weights of ten. However, in this case we allow the subdomains to be up to ten percent imbalanced. Therefore, subdomains of weight eleven are acceptable[2]. Figure 12(b) shows the gain of each vertex. Since all of the gains are negative, moving any vertex will result in the edge-cut increasing. Therefore, the bisection is in a local minimum. However, the algorithm will still select one of the vertices with the highest gain and move it. The white vertex is selected. Figure 12(c) shows the new bisection, as well as the updated vertex gains. There are now two positive gain vertices. However, neither of these can be moved at this time. The black vertex has just moved, and so it is ineligible to move again until the end of the pass. The other vertex with +1 gain is unable to move, as this will violate the balance constraint. Instead, one of the highest negative-gain vertices (shown in white) from the left subdomain is selected. Figure 12(d) shows the result of this move. Now there are two positive gain vertices that are able to move and two that are ineligible to move. The white vertex is selected. Figure 13 shows the results of continued refinement. By Figure 13(d), the bisection has reached another minimum with an edge-cut of two. The refinement algorithm has succeeded in climbing out of the original local minimum and reducing the edge-cut from six to two.

While KL/FM schemes are able to escape from certain types of local minima, this ability is still limited. Therefore, the quality of the final bisection obtained by KL/FM schemes is highly dependent on the quality of the input bisection. Several techniques have been developed that try to improve these algorithms by allowing the movement of larger sets of vertices together (i.e., more than just a single vertex or vertex pair) [2, 16, 28]. These schemes improve the effectiveness of KL/FM refinement at the cost of increased algorithm complexity. KL/FM-type refinement algorithms tend to be more effective when the average degree of the graph is large [9]. Furthermore, they perform much better when the balance constraints are relaxed. When perfect balance is desired, these schemes are quite constrained as to the refinement moves that can be made at any one time. As the imbalance tolerance increases, they are allowed greater freedom in making vertex moves and can provide higher-quality bisections.

### 0.3.3 Spectral Methods

Another method of solving the bisection problem is to formulate it as the optimization of a discrete quadratic function. However, even with this new formulation, the problem is still intractable. For this reason, a class of

[2]It is common for KL/FM-type algorithms to tolerate a slight amount of imbalance in the partitioning in an attempt to minimize the edge-cut.

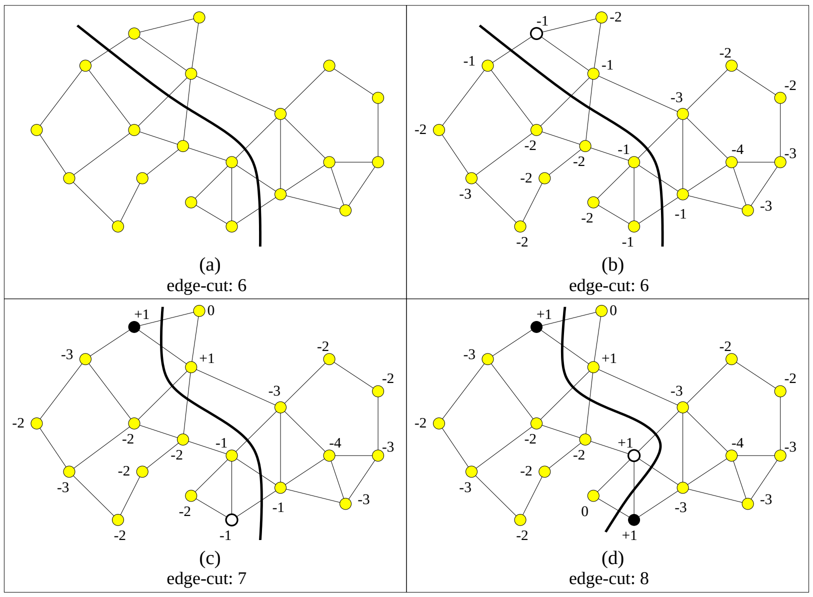


Figure 12: A bisection of a graph refined by a KL/FM algorithm. The white vertices indicate those selected to be moved. In (a) the partitioning is in a local minimum. In (b) the algorithm explores moves that increase the edge-cut. In (c) and (d) the edge-cut is increased, but now there are edge-cut reducing moves to be made.

graph partitioning methods, called *spectral* methods, relax this *discrete* optimization problem by transforming it into a *continuous* one. The minimization of the relaxed problem is then solved by computing the second eigenvector of the discrete Laplacian of the graph.

More precisely, spectral methods work as follows. Given a graph $G$, its discrete Laplacian matrix $L_G$ is defined as

$$(L_G)_{qr} = \begin{cases} 1, & \text{if } q \neq r,\ q \text{ and } r \text{ are neighbors,} \\ -deg(q), & \text{if } q = r, \\ 0, & \text{otherwise.} \end{cases}$$

$L_G$ is equal to $A - D$, where $A$ is the adjacency matrix of the graph and $D$ is a diagonal matrix in which $D[i,i]$ is equal to the degree of vertex $i$. The discrete Laplacian $L_G$ is a negative semidefinite matrix. Furthermore, its largest eigenvalue is zero and the corresponding eigenvector consists of all ones. Assuming that the graph is connected, the magnitude of the second largest eigenvalue gives a measure of the connectivity of the graph. The eigenvector corresponding to this eigenvalue (referred to as the *Fiedler* vector), when associated with the vertices of the graph, gives a measure of the distance (based on connectivity) between the vertices. Once this measure of distance is computed for each vertex, these can be sorted by this value, and the ordered list can be split into two parts to produce a bisection [38, 74, 75]. A $k$-way partitioning can be computed by recursive bisection. Figure 14 illustrates the spectral bisection technique. It shows a graph along with its adjacency matrix $A$, degree matrix $D$, Laplacian $L_G$, and the resulting bisected graph.

While the recursive spectral bisection algorithm typically produces higher-quality partitionings than geometric schemes, calculating the Fiedler vector is computationally intensive. This process dominates the runtime of the scheme and results in overall times that are several orders of magnitude higher than geometric techniques. For this reason, great attention has been focused on speeding up the algorithm. First, the improvement of methods, such as the Lanczos algorithm [66], for approximating eigenvectors has made the computation of eigenvectors practical. Multilevel methods have also been employed to speed up the computation of eigenvectors [4][3]. Finally, spectral partitioning schemes that use multiple eigenvectors in

[3]Note that multilevel eigensolver methods are not the same as the multilevel graph partitioning techniques discussed in Section 0.3.4.

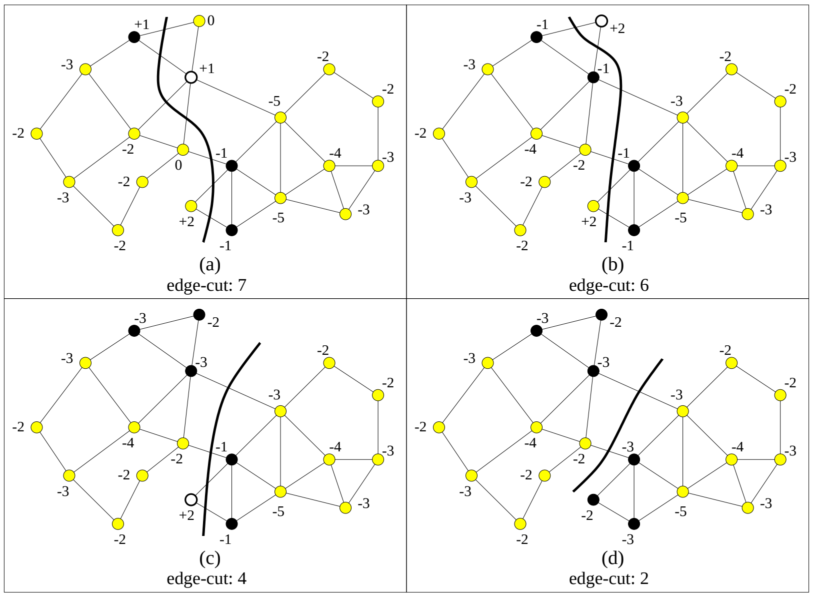


Figure 13: The KL/FM algorithm from Figure 12 is continued here. Edge-cut reducing moves are shown from (a) through (d). By (d), the refinement algorithm has reached a local minimum.

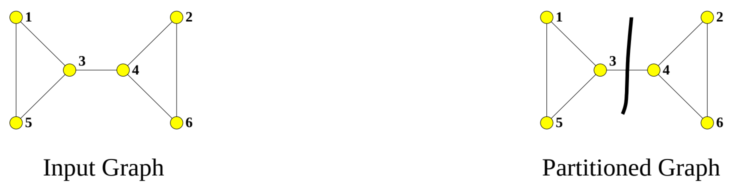
order to divide the computation into four and eight parts at each step of the recursive decomposition have been investigated [38]. Since the computation of the additional eigenvectors is relatively inexpensive, this scheme has a smaller net cost, while producing partitionings of comparable or better quality, compared to bisecting the graph at each recursive step.

### 0.3.4 Multilevel Schemes

Recently, a new class of partitioning algorithms has been developed [9, 13, 26, 30, 37, 45, 49, 61, 97] that is based on the multilevel paradigm. This paradigm consists of three phases: graph coarsening, initial partitioning, and multilevel refinement. In the graph coarsening phase, a series of graphs is constructed by collapsing together selected vertices of the input graph in order to form a related coarser graph. This newly constructed graph then acts as the input graph for another round of graph coarsening, and so on, until a sufficiently small graph is obtained. Computation of the initial bisection is performed on the coarsest (hence smallest) of these graphs and so is very fast. Finally, partition refinement is performed on each level graph, from the coarsest to the finest (i.e., original graph) using a KL/FM-type algorithm. Figure 15 illustrates the multilevel paradigm.

A common method for graph coarsening is collapsing together the pairs of vertices that form a matching. A matching of the graph is a set of edges, no two of which are incident on the same vertex. Vertex matchings can be computed by a number of methods. Widely used schemes include random matching, heavy-edge matching [45], maximum weighted matching [21], and approximated maximum weighted matching [61]. As an example, Figure 16(a) shows a random matching along with the coarsened graph that results from collapsing together vertices incident on every matched edge. Figure 16(b) shows a heavy-edge matching that tends to select edges with higher weights [45].

The multilevel paradigm works well for two reasons. First, a good coarsening scheme can hide a large number of edges on the coarsest graph. Figure 16 illustrates this point. The original graphs in Figures 16(a) and (b) have total edge weights of thirty-seven. After coarsening is performed on each, their total edge weights are reduced. Figures 16(a) and (b) show two possible coarsening heuristics, random and heavy-edge. In both cases, the total weight of the visible edges in the coarsened graph is less than that on the original graph. Note that by reducing the exposed edge weight, the task of computing a good quality partitioning becomes



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & & & 1 & & 1 & \\ 2 & & & & 1 & & 1 \\ 3 & 1 & & & 1 & 1 & \\ 4 & & 1 & 1 & & & 1 \\ 5 & 1 & & 1 & & & \\ 6 & & 1 & & 1 & & \end{array}$$

A

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 2 & & & & & \\ 2 & & 2 & & & & \\ 3 & & & 3 & & & \\ 4 & & & & 3 & & \\ 5 & & & & & 2 & \\ 6 & & & & & & 2 \end{array}$$

D

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & -2 & & 1 & & 1 & \\ 2 & & -2 & & 1 & & 1 \\ 3 & 1 & & -3 & 1 & 1 & \\ 4 & & 1 & 1 & -3 & & 1 \\ 5 & 1 & & 1 & & -2 & \\ 6 & & 1 & & 1 & & -2 \end{array}$$

$L_G$ = A - D

Figure 14: A graph, along with its adjacency matrix $A$, degree matrix $D$, and Laplacian $L_G$. The Fiedler vector of $L_G$ associates a value with each vertex. The vertices are then sorted according to this value. A bisection is obtained by splitting the sorted list in half.

easier. For example, a worst case partitioning (i.e., one that cuts every edge) of the coarsest graph will be of higher quality than the worst case partitioning of the original graph. Also, a random bisection of the coarsest graph will tend to be better than a random bisection of the original graph.

The second reason that the multilevel paradigm works well is that incremental refinement schemes such as KL/FM become much more powerful in the multilevel context. Here, the movement of a single vertex across the subdomain boundary in one of the coarse graphs is equivalent to the movement of a large number of highly connected vertices in the original graph, but is much faster. The ability of a refinement algorithm to move groups of highly connected vertices all at once allows the algorithm to escape from some types of local minima. Figure 17 illustrates this phenomenon. It shows a partitioned graph (included are the edge weights for the graph) both before and after coarsening. The partitioning for the uncoarsened graph (on the left-hand side) is in a local minimum. However, the partitioning for the coarsened graph (on the right side) is not. That is, edge-cut reducing moves can be made here. As discussed in Section 0.3.2, modifications of KL/FM schemes have been developed that attempt to move sets of vertices at once in order to improve the effectiveness of refinement [2, 16, 28]. However, computing good sets to move is computationally intensive. Multilevel schemes benefit from moving multiple vertices at the same time without having to compute these sets.

Preliminary theoretical work that explains the effectiveness of the multilevel paradigm has been done in [44].

**Multilevel Recursive Bisection** The multilevel paradigm was developed independently by Bui and Jones, [9] in the context of computing fill-reducing matrix reorderings; by Hendrickson and Leland [37], in the context of finite-element mesh partitioning; and by Hauck and Borriello [30] (called Optimized KLFM) and by Cong and Smith [13], for hypergraph partitioning[4]. Karypis and Kumar studied this paradigm extensively in [45] by evaluating a variety of coarsening, initial partitioning, and refinement schemes in the context of graphs from many different application domains. Their evaluation showed that the overall paradigm is quite robust and consistently outperformed the spectral partitioning method in both speed and quality of partitioning. The evaluation also showed that the heavy-edge matching heuristic is very effective in hiding edges in the coarsest graph. Figure 16 gives an example of this. The random matching in Figure 16(a) results in a total exposed edge weight of thirty, while the heavy-edge matching in Figure 16(b) results in a total exposed edge weight of only twenty-one. When heavy-edge matching is used, the initial

[4]A hypergraph is a generalization of a graph in which edges can connect not just two, but an arbitrary number of vertices.

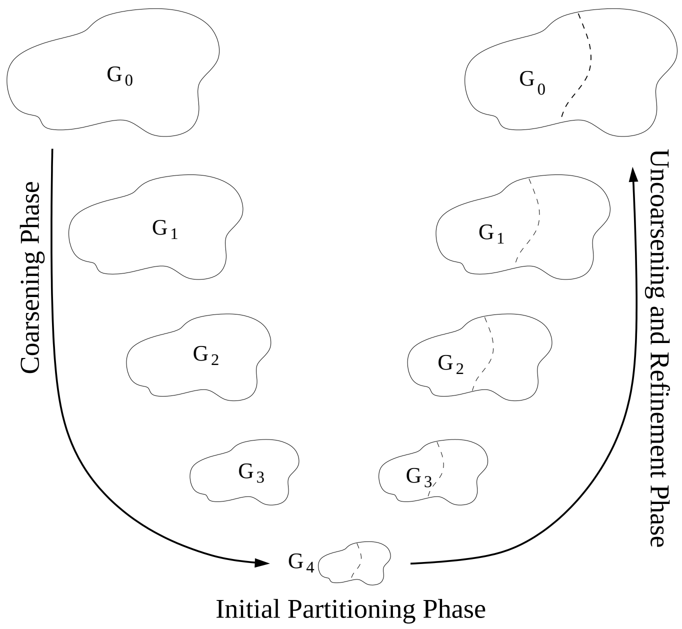


Figure 15: The three phases of the multilevel graph partitioning paradigm. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a bisection is computed. During the uncoarsening and refinement phase, the bisection is successively refined as it is projected to the larger graphs. $G_0$ is the input graph, which is the finest graph. $G_{i+1}$ is the next level coarser graph of $G_i$. $G_4$ is the coarsest graph.

partitioning that is computed on the coarsest graph is often not too different from the final partitioning obtained after multilevel refinement. This allows the use of greatly simplified (and therefore fast) variants of KL/FM schemes during the uncoarsening phase. These simplified schemes significantly speed up refinement without compromising the quality of the partitioning. Furthermore, these simplified variants are much more amenable to parallelization than the original KL/FM heuristic that is inherently serial [24]. Karypis and Kumar also showed that as long as a good matching scheme is used and KL/FM refinement is performed on each level graph, the method of computing the initial partitioning on the coarsest graph does not have much impact on the final solution quality.

Multilevel recursive bisection partitioning algorithms are available in several public domain libraries, such as Chaco [36], METIS [47], and SCOTCH [68], and are used extensively for graph partitioning in a variety of domains. Additional variations of the heavy-edge heuristic are presented in [51] in the context of hypergraph partitioning. These variations are implemented in the hMetis [46] library for partitioning hypergraphs.

**Multilevel $k$-way Partitioning** Karypis and Kumar [49] present a scheme for refining a $k$-way partitioning that is a generalization of simplified variants of the KL/FM bisection refinement algorithm. Using this $k$-way refinement scheme, Karypis and Kumar present a $k$-way multilevel partitioning algorithm in [49] whose runtime is linear in the number of edges (i.e., $O(|E|)$); whereas the runtime of multilevel recursive bisection schemes is $O(|E| \log k)$. Experiments on a large number of graphs arising in various domains (including finite-element methods, linear programming, VLSI, and transportation) show that this scheme produces partitionings that are of comparable or better quality than those produced by multilevel recursive bisection, while requiring substantially less time. For example, partitionings of graphs containing millions of vertices can be computed in only a few minutes on a typical workstation. For many of these graphs, the process of graph partitioning takes less time than the time to read the graph from disk into memory. Compared with multilevel spectral bisection [37, 74, 75], multilevel $k$-way partitioning is usually two orders of magnitude faster and produces partitionings with generally smaller edge-cuts. The runtimes of multilevel $k$-way partitioning algorithms are usually comparable to the runtimes of small numbers (2-4) of runs of

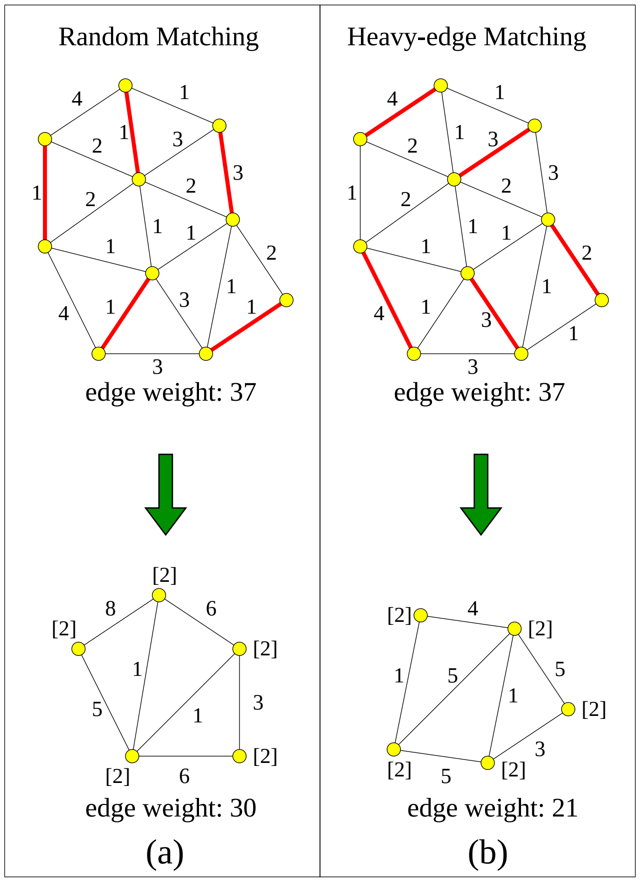


Figure 16: A random matching of a graph along with the coarsened graph (a). The same graph is matched (and coarsened) with the heavy-edge heuristic in (b). The heavy-edge matching minimizes the exposed edge weight.

geometric recursive bisection algorithms [23, 31, 59, 62, 77] but tend to produce higher-quality partitionings for a variety of graphs, including those originating in scientific-simulation applications.

Multilevel $k$-way graph partitioning algorithms are available in the JOSTLE [94] and METIS [47] software packages.

### 0.3.5 Combined Schemes

All of the graph partitioning techniques discussed in this section have individual advantages and disadvantages. Combining different types of schemes intelligently can maximize the advantages without suffering all of the disadvantages. In this section, we briefly describe a few commonly-used combinations.

KL/FM-type algorithms are often used to improve the quality of partitionings that are computed by other methods. For example, an initial partitioning can be computed by a fast geometric method, and then the relatively low quality partitioning can be refined by a KL/FM algorithm. Multilevel schemes use this technique, as well, by performing KL/FM refinement on each coarsened version of the graph after an initial partitioning is computed (by either LND [45], spectral [37], or other methods [26]). As another example, spectral methods can be used to compute coordinate information for vertices [29]. These coordinates can then be used by a geometric scheme to partition the graph [85].

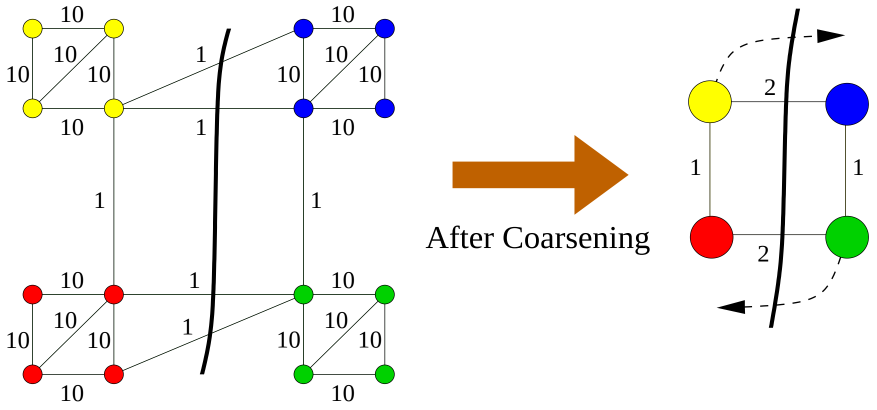


Figure 17: An example of a partitioned graph (with edge weights) before and after coarsening. The partitioning for the uncoarsened graph is in a local minima, while the partitioning for the coarsened graph is not.

### 0.3.6 Qualitative Comparison of Graph Partitioning Schemes

The large number of graph partitioning schemes reviewed in this section differ widely in the edge-cut quality produced, runtime, degree of parallelism, and applicability to certain kinds of graphs. Often, it is not clear as to which scheme is better under different scenarios. In this section, we categorize properties of graph partitioning algorithms commonly used in scientific-simulation applications. This task is quite difficult, as it is not possible to precisely model the properties of the graph partitioning algorithms. Furthermore, for most of the schemes, sufficient data on the edge-cut quality and runtime for a common pool of benchmark graphs is not available. The relative comparison of different schemes draws upon the experimental results in [23, 31, 38, 45]. We try to make reasonable assumptions whenever enough data is not available. For the sake of simplicity, we have chosen to represent each property in terms of a small discrete scale. In the absence of extensive data, it is not possible to do much better than this in any case.

Figure 18 compares three variations of spectral partitioners [4, 37, 74, 75], a multilevel algorithm [49], an LND algorithm [22], a Kernighan-Lin algorithm (with random initial partitionings) [55], a CND algorithm [31], two variations of the RIB algorithm [36, 62], and two variations of the geometric sphere-cutting algorithm [23, 59].

For each graph partitioning algorithm, Figure 18 shows a number of characteristics. The first column shows the number of trials that are performed for each partitioning algorithm. For example, for the KL algorithm, different trials can be performed each starting with a different random partitioning of the graph. Each trial is a different run of the partitioning algorithm, and the best of these is selected. As we can see from this table, some algorithms require only a single trial either because multiple trials will give the same partitioning or a single trial gives very good results (as in the case of multilevel graph partitioning). However, for some schemes (e.g., KL and geometric partitioning), different trials yield significantly different edge-cuts. Hence, these schemes usually require multiple trials in order to produce good quality partitionings. For multiple trials, we only show the case of ten and fifty trials, as often the quality saturates beyond fifty trials or the runtime becomes too large.

The second column shows whether the partitioning algorithm requires coordinates for the vertices of the graph. Some algorithms such as CND and RIB are applicable only if coordinate information is available. Others (e.g., combinatorial schemes) only require the sets of vertices and edges.

The third column of Figure 18 shows the relative quality of the partitionings produced by the various schemes. Each additional circle corresponds to roughly ten percent improvement in the edge-cut. The edge-cut quality for CND serves as the base, and it is shown with one circle. Schemes with two circles for quality should find partitionings that are roughly ten percent better than CND. This column shows that the quality of the

| | Number of Trials | Needs Coordinates | Quality | Local View | Global View | Run Time | Degree of Parallelism |
|---|---|---|---|---|---|---|---|
| Recursive Spectral Bisection | 1 | no | ●●●● | ○ | ●●●● | ■■■■ | ▲▲ |
| Multilevel Spectral Bisection | 1 | no | ●●●● | ○ | ●●●● | ■■■ | ▲▲ |
| Mulitlevel Spectral Bisection-KL | 1 | no | ●●●●●● | ●● | ●●●● | ■■■ | ▲▲ |
| Multilevel Partitioning | 1 | no | ●●●●●● | ●● | ●●●● | ■■ | ▲▲ |
| Levelized Nested Dissection | 1 | no | ●● | ○ | ●● | ■■ | ▲▲ |
| | 1 | no | ●● | ●● | ○ | ■■ | ▲ |
| Kernighan-Lin | 10 | no | ●●●◐ | ●● | ●◐ | ■■■ | ▲▲ |
| | 50 | no | ●●●● | ●● | ●● | ■■■◧ | ▲▲ |
| Coordinate Nested Dissection | 1 | yes | ● | ○ | ● | ■ | ▲▲▲ |
| Recursive Inertial Bisection | 1 | yes | ●● | ○ | ●● | ■ | ▲▲▲ |
| Recursive Inertial Bisection-KL | 1 | yes | ●●●● | ●● | ●● | ■■ | ▲ |
| | 1 | yes | ●● | ○ | ●● | ■ | ▲▲▲ |
| Geometric Sphere-cutting | 10 | yes | ●●●◐ | ○ | ●●●◐ | ■■ | ▲▲▲ |
| | 50 | yes | ●●●● | ○ | ●●●● | ■■◧ | ▲▲▲ |
| | 1 | yes | ●●●● | ●● | ●● | ■■ | ▲ |
| Geometric Sphere-cutting-KL | 10 | yes | ●●●●●◐ | ●● | ●●●◐ | ■■■ | ▲▲ |
| | 50 | yes | ●●●●●● | ●● | ●●●● | ■■■◧ | ▲▲ |

Figure 18: Graph partitioning schemes rated with respect to quality, runtime, degree of parallelism and related characteristics.

partitionings produced by the multilevel graph partitioning algorithm and the multilevel spectral bisection with KL is very good. The quality of geometric partitioning with KL refinement is equally good when fifty or more trials are performed. The quality of the other schemes is worse than the above three by various degrees. Note that for both KL partitioning and geometric partitioning, the quality improves as the number of trials increases.

The reason for the differences in the quality of the various schemes can be understood if we consider the degree of quality as a sum of two quantities that we refer to as local view and global view. A graph partitioning algorithm has a local view of the graph if it is able to do localized refinement. According to this definition, all the graph partitioning algorithms that perform KL/FM-type refinement possess this local view, whereas the others do not. Global view refers to the extent that the graph partitioning algorithm takes into account the structure of the graph. For instance, spectral bisection algorithms take into account only global information of the graph by minimizing the edge-cut in the continuous approximation of the discrete problem. On the other hand, a single trial of the KL algorithm does not utilize information about the overall structure of the graph, since it starts from a random bisection. For schemes that require multiple random trials, the degree of the global view increases as the number of trials increases. The global view of multilevel graph partitioning is among the highest. This is because multilevel graph partitioning captures global graph structure in two ways. First, it captures global structure through the process of coarsening; second, it captures global structure during initial graph partitioning by performing multiple trials.

The sixth column of Figure 18 shows the relative time required by different graph partitioning schemes. CND, RIB, and geometric sphere-cutting (with a single trial) require relatively small amounts of time. We show the runtime of these schemes by one square. Each additional square corresponds to roughly a factor of ten increase in the runtime. As we can see, spectral graph partitioning schemes require several orders of magnitude more time than the faster schemes. However, the quality of the partitionings produced by the

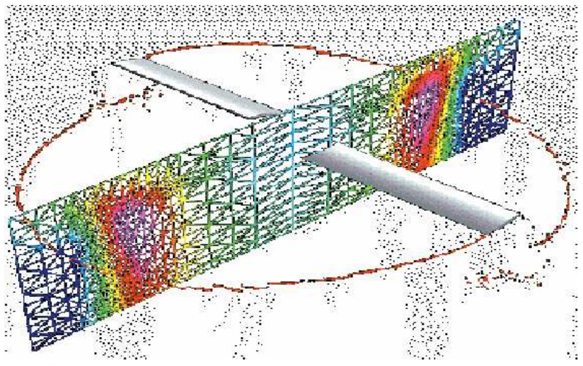


Figure 19: A helicopter blade rotating through a mesh. As the blade spins, the mesh is adapted by refining it in the regions that the blade has entered and de-refining it in the regions that are no longer of interest. (Figure provided by Rupak Biswas, NASA Ames Research Center.)

faster schemes is relatively poor. The quality of these schemes can be improved by increasing the number of trials and/or by using the KL/FM refinement, both of which increase the runtime of the partitioner. On the other hand, multilevel graph partitioning requires a moderate amount of time and produces partitionings of very high quality.

The degree of parallelizability of different schemes differs significantly and is depicted by a number of triangles in the seventh column of Figure 18. One triangle means that the scheme is largely sequential, two triangles means that the scheme can exploit a moderate amount of parallelism, and three triangles means that the scheme can be parallelized quite effectively. Schemes that require multiple trials are inherently parallel, as different trials can be done on different (groups of) processors. In contrast, a single trial of KL is very difficult to parallelize and appears inherently serial [24]. Multilevel schemes that utilize relaxed variations of KL/FM refinement and the spectral bisection scheme are moderately parallel in nature.

## 0.4 Load Balancing of Adaptive Computations

For large-scale scientific simulations, the computational requirements of techniques relying on globally refined meshes become very high, especially as the complexity and size of the problems increase. By locally refining and de-refining the mesh either to capture flow-field phenomena of interest [7] or to account for variations in errors [67], adaptive methods make standard computational methods more cost effective. One such example is numerical simulations for improving the design of helicopter blades [7]. (See Figure 19.) In order to capture flow-field phenomena of interest accurately, the finite-element mesh must be extremely fine both around the helicopter blade and in the vicinity of the sound vortex created by the blade. It should be coarser in other regions of the mesh for maximum efficiency. As the simulation progresses, neither the blade nor the sound vortex remains stationary. Therefore, the new regions of the mesh that these enter need to be refined, while those regions that are no longer of key interest should be de-refined. These dynamic adjustments to the mesh result in some processors having significantly more (or less) work than others and thus cause load imbalance. Similar issues exist for problems in which the amount of computation associated with each mesh element changes over time [18]. For example, in particles-in-cells methods that advect particles through a mesh, large temporal and spatial variations in particle density can introduce substantial load imbalance.

In both of these types of applications, it is necessary to dynamically load balance the computations as the simulation progresses. This dynamic load balancing can be achieved by using a graph partitioning algorithm. In the case of adaptive finite-element methods, the graph either corresponds to the mesh obtained after adaptation or else corresponds to the original mesh with the vertex weights adjusted to reflect error estimates [67]. In the case of particles-in-cells, the graph corresponds to the original mesh with the vertex weights adjusted to reflect the particle density. We will refer to this problem as *adaptive graph partitioning* to differentiate it from the static graph partitioning problem that arises when the computations remain fixed.

Adaptive graph partitioning shares most of the requirements and characteristics of static graph partitioning but also adds an additional objective. That is, the amount of data that needs to be redistributed among the processors in order to balance the load should be minimized. In order to accurately measure this cost, we need to consider not only the weight of a vertex, but also its *size* [63]. Vertex weight is the computational cost of the work represented by the vertex, while size reflects its redistribution cost. Thus, the repartitioner should attempt to balance the partitioning with respect to vertex weight while minimizing vertex migration with respect to vertex size. Depending on the representation and storage policy of the data, size and weight may not necessarily be equal [63].

Oliker and Biswas studied various metrics for measuring data redistribution costs in [63]. They presented the metrics TotalV and MaxV. TotalV is defined as the sum of the sizes of vertices that change subdomains as the result of repartitioning. TotalV reflects the overall volume of communications needed to balance the partitioning. MaxV is defined as the maximum of the sums of the sizes of those vertices that migrate into or out of any one subdomain as a result of repartitioning. MaxV reflects the maximum time needed by any one processor to send or receive data. Results in [63] show that measuring the MaxV can sometimes be a better indicator of data redistribution overhead than measuring the TotalV. However, many repartitioning schemes [63, 80, 81, 100] attempt to minimize TotalV instead of MaxV for the following reasons: (i) TotalV can be minimized during refinement by the use of relatively simple heuristics; minimizing MaxV tends to be more difficult. (ii) The MaxV is lower bounded by the amount of vertex weight that needs to be moved out of the most overweight subdomain (or into the most underweight subdomain). For many problems, this lower bound can dominate the MaxV and so no improvement is possible. (iii) Minimizing TotalV often tends to do a fairly good job of minimizing MaxV.

**Repartitioning Approaches** A repartitioning of a graph can be computed by simply partitioning the new graph from scratch. Since no consideration is given to the existing partitioning, it is unlikely that vertices will be assigned to their original subdomains with this method. Therefore, this approach will tend to require much more data redistribution than is necessary in order to balance the load.

An alternate strategy is to attempt to perturb the input partitioning just enough so as to balance it. This can be accomplished trivially by the following *cut-and-paste repartitioning* method: Excess vertices in overweight subdomains are simply swapped into one or more underweight subdomains (regardless of whether these subdomains are adjacent) in order to balance the partitioning. While this method will optimally minimize data redistribution, it can result in significantly higher edge-cuts compared with more sophisticated approaches and will typically result in disconnected subdomains. For these reasons, it is usually not considered a viable repartitioning scheme for most applications. A better approach is to use a diffusion-based repartitioning scheme. These schemes attempt to minimize the data redistribution costs while significantly decreasing the possibility that subdomains become disconnected.

Figure 20 illustrates these methods for a graph whose vertices and edges have weights of one. The shading of a vertex indicates the original subdomain to which it belongs. In Figure 20(a), the original partitioning is imbalanced because subdomain 3 has a weight of six, while the average subdomain weight is only four. The edge-cut of the original partitioning is twelve. In Figure 20(b), the original partitioning is ignored and the graph is partitioned from scratch. This partitioning also has an edge-cut of twelve. However, thirteen out of twenty vertices are required to change subdomains. That is, TotalV is thirteen. MaxV is six. In Figure 20(c), cut-and-paste repartitioning was used. Here, only two vertices are required to change subdomains and MaxV is also two. The edge-cut of this partitioning is sixteen, and subdomain 1 is now disconnected. Figure 20(d) gives a diffusive repartitioning that presents a compromise between those in Figure 20(b) and (c). Here, TotalV is four, MaxV is two, and the edge-cut is fourteen.

### 0.4.1 Scratch-Remap Repartitioners

The example in Figure 20(b) illustrated how partitioning from scratch resulted in the lowest edge-cut of the three repartitioning methods. This is expected since it is possible to use a state-of-the-art graph partitioner to compute the new partitioning from scratch. However, this repartitioning resulted in the highest data

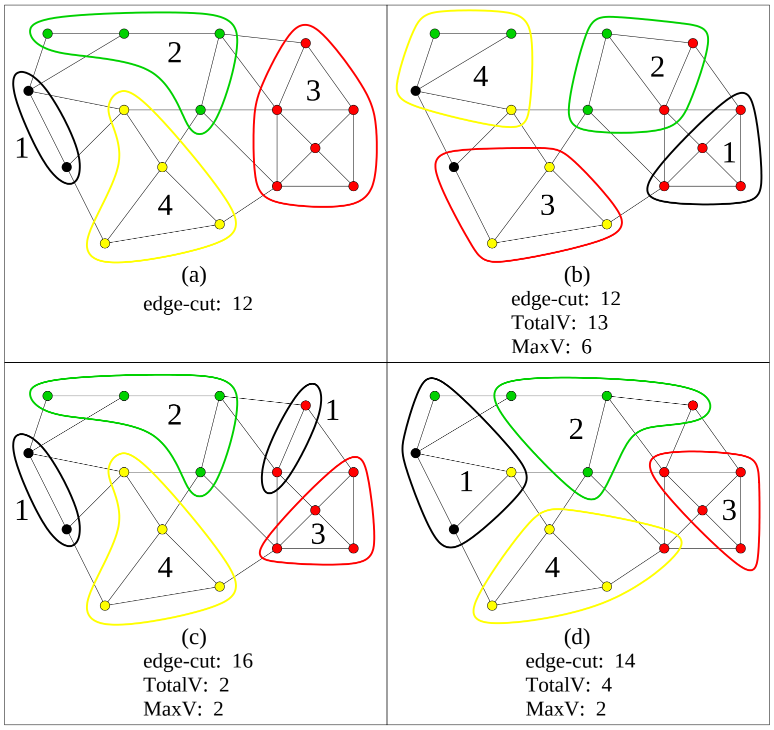


Figure 20: Various repartitioning schemes. An example of an imbalanced partitioning (a). This partitioning is balanced by partitioning the graph from scratch (b), cut-and-pasted repartitioning (c), and diffusive repartitioning (d).

redistribution costs. To understand this, it is necessary to examine the partitionings in Figures 20(a) and (b). Notice that in Figure 20(a), subdomain 1 is on the left, subdomain 3 is on the right, and subdomain 4 is on the bottom. For the partitioning in Figure 20(b), subdomain 1 is on the right, subdomain 3 is on the bottom, and subdomain 4 is on the top left. A large amount of the data redistribution required for the partitioning in Figure 20(b) is brought about because the subdomains are labeled suboptimally. Simply changing the subdomain labels of the new partitioning in accordance with the old partitioning (without otherwise modifying the partitioning) can significantly reduce the data redistribution cost [88].

Oliker and Biswas [63] present a number of repartitioning schemes that compute new partitionings from scratch and then intelligently map the subdomain labels to those of the original partitionings in order to minimize the data redistribution costs. We refer to this method as *scratch-remap* repartitioning. Partition remapping is performed as follows: (i) Construct a similarity matrix, $S$, of size $k \times k$. A similarity matrix is one in which the rows represent the subdomains of the old partitioning, the columns represent the subdomains of the new partitioning, and each element, $S_{qr}$, represents the sum of the sizes of the vertices that are in subdomain $q$ of the old partitioning and in subdomain $r$ of the new partitioning. (ii) Select $k$ elements such that every row and column contains exactly one selected element and such that the sum of the selected elements is maximized. This corresponds to the remapping in which the amount of overlap between the original and the remapped partitionings is maximized, and hence, the total volume of data redistribution required in order to realize the remapped partitioning is minimized. (iii) For each element $S_{qr}$ selected, rename domain $r$ to domain $q$ on the remapped partitioning. Figure 21 illustrates such a remapping process. Here, similarity matrix $S$ has been constructed based on the example in Figure 20. The first row of $S$ indicates that subdomain 1 on the old partitioning (Figure 20(a)) consists of zero vertices from subdomains 1 and 2 on the new partitioning (Figure 20(b)) and one vertex from each of subdomains 3 and 4 on the new partitioning. Likewise, the second row indicates that subdomain 2 on the old partitioning consists of two vertices from each of subdomains 2 and 4 on the new partitioning and zero vertices from the other two subdomains. The third and fourth rows are constructed similarly. In this example, we select underlined elements $S_{14}$, $S_{22}$,

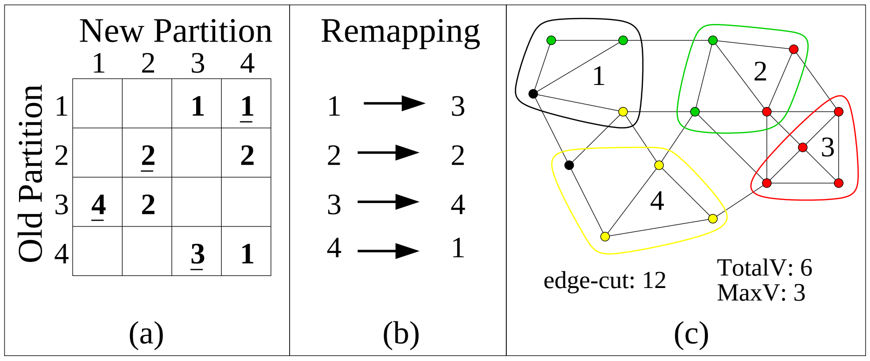


Figure 21: A similarity matrix, the corresponding remapping, and the remapped partitioning from Figure 20(b).

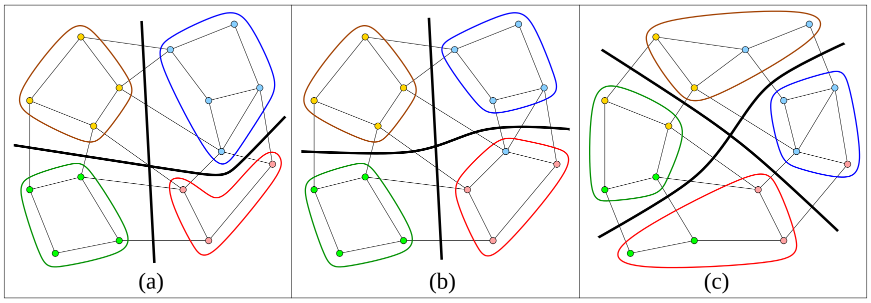


Figure 22: An imbalanced partitioning and two repartitioning techniques. The partitioning in (a) is imbalanced. It is balanced by an incremental method in (b) and by a scratch-remap method in (c).

$S_{31}$, and $S_{43}$. This combination maximizes the sum of the sizes of the selected elements. Running through the selected elements, subdomain 1 on the newly computed partitioning is renamed 3, and subdomains 2, 3, and 4 are renamed 2, 4, and 1, respectively. Figure 21(c) shows the remapped partitioning. Here, TotalV is six and MaxV is three.

Although the remapping phase reduces the data redistribution costs (without affecting the edge-cut), scratch-remap schemes still tend to result in higher redistribution costs than schemes that attempt to balance the input partitioning by minimal perturbation (e.g., cut-and-paste and diffusion-based schemes). For example, if the newly adapted graph is only slightly different from the original graph, then partitioning from scratch could produce a new partitioning that is still substantially different from the original and requires many vertices to be moved even after the remapping phase. On such a graph, the imbalance could easily be corrected by moving only a small number of vertices. Figure 22 illustrates an example of this. The partitioning in Figure 22(a) is slightly imbalanced. The upper-right subdomain has five vertices, while the average subdomain weight is four. In Figure 22(b), the partitioning is balanced by moving a single vertex from the upper-right subdomain to the lower-right subdomain. Therefore, both TotalV and MaxV are one. Figure 22(c) shows a new partitioning that has been computed from scratch and then optimally remapped to the partitioning in Figure 22(a). Despite this optimal remapping, the repartitioning has a TotalV of seven and a MaxV of two. All three of the partitionings have similar edge-cuts.

The reason that the scratch-remap scheme does so poorly here with respect to data redistribution is because the information that is provided by the original partitioning is not utilized until the final remapping process. At this point, it is too late to avoid high data redistribution costs even if we compute an optimal remapping. The problem in our example is that the partitioning in Figure 22(a) is shaped like a '+', while the partitioning in Figure 22(c) forms an 'x'. Both of these are of equal quality, so a static partitioning algorithm could easily compute either of these. However, we would like the partitioning algorithm used in a scratch-remap repartitioner to drive the computation of the partitioning toward that of the original partitioning, whenever possible, without affecting the quality. A scratch-remap algorithm can potentially do this if it is able to extract and use the information implicit in the original partitioning during the computation of the new partitioning. An algorithm called Locally-Matched Multilevel Scratch-Remap (LMSR) that tries to accomplish this is presented in [81]. LMSR decreases the amount of data redistribution required to balance the graph compared to naive scratch-remap schemes, particularly for slightly imbalanced graphs [81].

### 0.4.2 Diffusion-based Repartitioners

Diffusive load balancing schemes attempt to minimize the difference between the original partitioning and the final repartitioning by making incremental changes in the partitioning to restore balance. Subdomains that are overweight in the original partitioning export vertices to adjacent subdomains. These may further export vertices to their neighbors in an effort to reach global balance. By limiting the movement of vertices to neighboring subdomains, these schemes attempt to minimize the edge-cut and maintain connected subdomains. As an example, the repartitioning in Figure 20(d) is obtained by a diffusive process. In this case, subdomain 3 migrates a vertex to each of subdomains 2 and 4. This causes the recipient subdomains to become overweight. Each next migrates a vertex to subdomain 1.

Any diffusion-based repartitioning scheme needs to address two questions: (i) *How much work should be transferred between processors?* and (ii) *Which tasks should be transferred?* The answer to the first question tells us how to balance the partitioning, while the answer to the second tells us how to minimize the edge-cut as we do this. A lot of work has focused on answering the first question in the context of balancing unrelated tasks that are unevenly distributed among processors [8, 14, 15, 40, 41, 42, 105, 107]. These take the machine architecture, but not the interdependencies of the tasks, into consideration when computing the amount of work to transfer between processors. More recently, in the context of adaptive computational simulations, work has focused not only on how much, but also which tasks to transfer [18, 64, 65, 67, 70, 80, 81, 87, 92, 93, 99, 100]. In the rest of this section, we focus on these schemes.

Schemes for determining how much work to transfer between processors can be grouped into two categories. Diffusion schemes that base the exchange of work among the processors only on their respective work loads (and not on the loads of distant processors) [80] are called *local* diffusion algorithms. Other schemes [18, 64, 65, 67, 70, 80, 81, 87, 92, 93, 99, 100], use global views of the processor loads to balance the partitioning. We call these *global* diffusion schemes. Most global diffusion schemes either perform diffusion in a recursive bisection manner [18, 87, 93], utilize space-filling curves [65, 67, 70], or compute *flow solutions* [64, 80, 81, 99, 100] that prescribe the amount of work to be moved between pairs of processors.

Recursive bisection diffusion schemes [18, 87, 93] split the subdomains into two groups and then attempt to balance these groups. Next, both of the (balanced) groups are split in two and the algorithm recurses on these subgroups.

Adaptive space-filling curve partitioners [65, 67, 70] can compute repartitionings by maintaining the original ordering of the mesh elements. Here, the weights associated with the ordered mesh elements are changed to reflect the structural changes in the computation. All that is required to compute a repartitioning is to recompute the $k$-way splitting of the ordered list with respect to the new weights.

Flow solutions are usually computed in order to optimize some objective. Ou and Ranka [64] present a global diffusion scheme that optimally minimizes the one-norm of the flow using linear programming. Such a scheme will minimize TOTALV provided that the weights and sizes of the vertices are equal. Hu and Blake [42] present a method that optimally minimizes the two-norm of the flow. They prove that such a flow solution can be obtained by solving the linear equation $(-L)\lambda = b$, where $b$ is the vector containing the load of each subdomain minus the average subdomain load, $L$ is the Laplacian matrix (as defined in Section 0.3.3) of the graph that models the subdomain connectivity (i.e., the *subdomain connectivity graph*), and $\lambda$, the flow solution, is a vector with $k$ elements. An amount of vertex weight equal to $\lambda_q - \lambda_r$ needs to be moved from subdomain $q$ to subdomain $r$ for every $r$ that is adjacent to $q$ in order to balance the partitioning.

Figure 23 illustrates the difference between one- and two-norm minimization of the flow solution. This figure shows the subdomain connectivity graph for a nine-way partitioning along with the two different

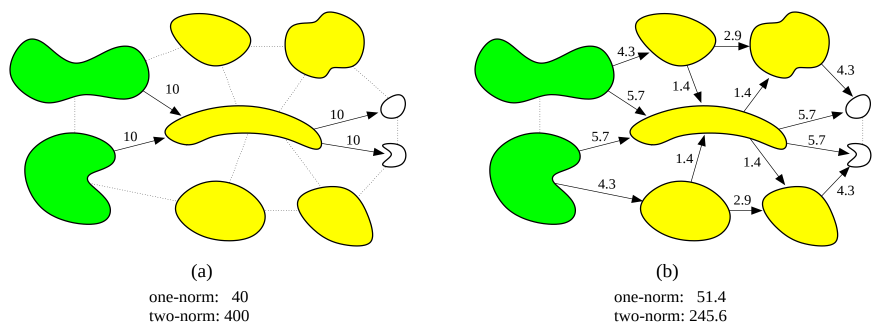


Figure 23: Two different flow solutions for the subdomain graph of an imbalanced partitioning. The one-norm of the data migration is minimized in (a). The two-norm of the data migration is minimized in (b).

flow solutions. Here, the two dark subdomains are overweight by ten, while the two white subdomains are underweight by ten. The weight of the rest of the subdomains equals the average subdomain weight. The flow solution in Figure 23(a) minimizes the one-norm of the data movement. The flow solution in Figure 23(b) minimizes the two-norm of the data movement. The one-norm minimization solution can minimize TotalV, but will not in general minimize MaxV, as most of the flow is sent through a few links. The two-norm minimization solution more evenly distributes the flow through the links (and thus tends to result in lower values for MaxV), but requires greater total flow (and therefore, worse TotalV), compared to the one-norm solution.

The flow solution indicates how much vertex weight needs to be transfered between each pair of adjacent subdomains. The second problem is to determine exactly which vertices to move so as to minimize the edge-cut of the resulting partitioning. One possibility is to repeatedly transfer layers of vertices along the subdomain boundary until the desired amount of vertex weight has been transferred [64, 93]. A more precise scheme is to move one vertex at a time across the subdomain boundary, each time selecting the vertex that will result in the smallest edge-cut [99]. This scheme, like the KL/FM algorithm, utilizes only a local view of the graph, and can make (globally) poor selections. This problem can be corrected if the transfer of vertices is performed in a multilevel context [80, 100]. Such schemes, called *multilevel diffusion* algorithms, perform graph coarsening and then begin diffusion on the coarsest graph. During the uncoarsening phase, vertices are moved to achieve (or maintain) load balance, while also trying to improve the edge-cut. By beginning diffusion on the coarsest graph, these algorithms are able to move large chunks of highly connected vertices in a single step. Thus, the bulk of the work required to balance the partitioning is done quickly. Furthermore, by moving highly connected vertices together, high-quality edge-cuts can often be maintained. Experimental results show that multilevel diffusion can compute partitionings of higher quality than schemes that perform diffusion only on the original graph [80, 100] and is often faster.

Partitionings that are highly imbalanced in localized areas, diffusion-based schemes require vertex flow to propagate over long distances. For this class of problems, it is beneficial to determine not only *how much* and *which* vertices to move, but also *when* vertices should move [15]. A diffusion algorithm, called Wavefront Diffusion, that determines the best time to migrate vertices is presented in [81]. In Wavefront Diffusion, the flow of vertices moves in a wavefront starting from the most overweight subdomains. This method guarantees that all subdomains will contain the largest possible selection of vertices when it is their turn to export vertices. Thus, subdomains are able to select those vertices for migration that will best minimize edge-cut and data redistribution costs. Wavefront Diffusion obtains significantly lower data redistribution costs while maintaining similar or better edge-cut results compared to diffusion schemes that do not determine the best time to migrate vertices, especially for partitionings that are highly imbalanced in localized areas [81].

**Trade-off Between Edge-cut and Data Redistribution Costs** Often, the objective of minimizing the data redistribution cost is at odds with the objective of minimizing the edge-cut. For applications in which the mesh is frequently adapted or the amount of state associated with each element is relatively high, minimizing the data redistribution cost is preferred over minimizing the edge-cut. For applications in which repartitioning occurs infrequently, the key objective of a repartitioning scheme will be obtaining the minimal edge-cut.

While a number of coarsening and refinement heuristics have been developed [80, 95] that can control the trade-offs between these two objectives to some extent, most adaptive partitioners naturally minimize one in preference to the other. For example, Wavefront Diffusion tends to minimize data redistribution costs better than the LMSR algorithm. However, the LMSR algorithm tends to minimize the edge-cut of the repartitioning better than Wavefront Diffusion. As such, the two provide the user with a limited control of the trade-offs among these objectives. A new scheme, called the Unified Repartitioning Algorithm [84], has been developed that gives the user a more fine-tuned control of the trade-offs among the objectives. Experimental results on a variety of problems show that the Unified Repartitioning Algorithm is able to reduce the sum of the inter-processor communication overhead incurred during the iterative mesh-based computation and the data redistribution costs required to balance the load as well as or better than other repartitioning schemes.

## 0.5 Parallel Graph Partitioning

The ability to perform partitioning in parallel is important for many reasons. The amount of memory on serial computers is often not enough to allow the partitioning of graphs corresponding to large problems that can now be solved on massively parallel computers and workstation clusters. A parallel graph partitioning algorithm can take advantage of the significantly higher amount of memory available in parallel computers to partition very large graphs. Also, as heterogeneous systems of parallel machines are integrated into a single system of systems (e.g., the NASA Information Power Grid [43]), the role of graph partitioning will change. Here, the exact number of processors and/or the architectural characteristics of the hardware assigned to a computation will not be known until immediately before the computation is permitted to execute. Parallel graph partitioning is crucial for efficiency in such an environment. In the context of adaptive graph partitioning, the graph is already distributed among processors, but needs to be repartitioned due to the dynamic nature of the underlying computation. In such cases, having to bring the graph to one processor for repartitioning can create a serious bottleneck that could adversely impact the scalability of the overall application.

Work in parallel graph partitioning [3, 24, 31, 50, 52, 78, 96] has been focused on geometric [31, 78], spectral [3], and multilevel partitioning schemes [50, 52, 96]. Geometric graph partitioning algorithms tend to be quite easy to parallelize. Typically, these require a parallel sorting algorithm. Spectral and multilevel partitioners are more difficult to parallelize. Their parallel asymptotic runtimes are the same as that of performing a parallel matrix–vector multiplication on a randomly partitioned matrix [52]. This is because the input graph is not well distributed across the processors. If the graph is first partitioned and then distributed across the processors accordingly, the parallel asymptotic runtimes of spectral and multilevel partitioners drop to that of performing a parallel matrix–vector multiplication on a well-partitioned matrix. Thus, performing these partitioning schemes efficiently in parallel requires a good partitioning of the input graph [52, 96]. In the case of static graph partitioning, we cannot expect the input graph to be partitioned already, since this is exactly what we are trying to do. However, for the adaptive graph partitioning problem, we can expect the input partitioning to be of high quality (that is, have a low edge-cut, even though it will be imbalanced). For this reason, parallel adaptive graph partitioners [81, 100] tend to run significantly faster than static partitioners.

Since the runtimes of most parallel geometric partitioning schemes are not affected by the initial distribution of the graph, they can be used to compute a partitioning for multilevel (or spectral) partitioning algorithms. That is, a rough partitioning of the input graph can be computed by a fast geometric approach. This partitioning can be used to redistribute the graph prior to performing parallel multilevel (or spectral) partitioning [53]. Use of this “boot-strapping” approach significantly increases the parallel efficiency of the more accurate partitioning scheme by providing it with data locality.

Parallel multilevel algorithms for graph partitioning are available in the ParMETIS [53] and JOSTLE [94] libraries.

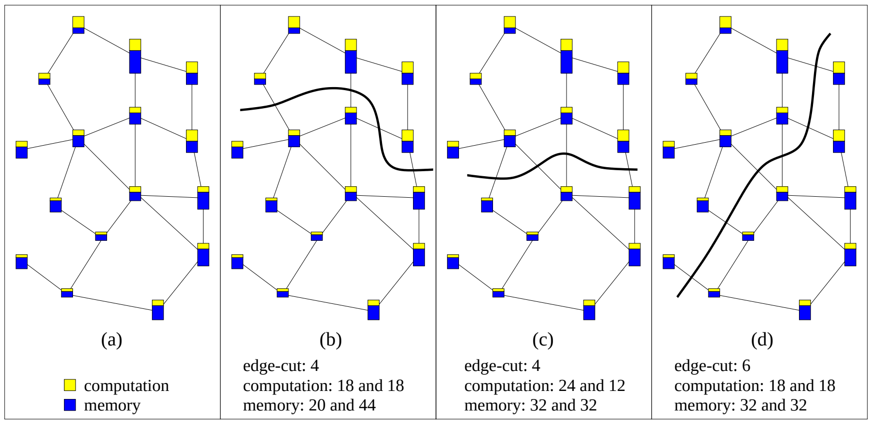


Figure 24: An example of a computation with nonuniform memory requirements. Each vertex in the graph is split into two amounts. The size of the lightly-shaded portion represents the amount of computation associated with the vertex, while the size of the dark portion represents the amount of memory associated with the vertex. The bisection in (b) balances the computation. The bisection in (c) balances the memory, but only the bisection in (d) balances both of these.

## 0.6 Multi-constraint, Multi-objective Graph Partitioning

In recent years, with advances in the state of the art of scientific simulation, sophisticated classes of computations such as multiphase, multiphysics, and multimesh simulations have become commonplace. For many of these, the traditional graph partitioning formulation is not adequate to ensure their efficient execution on high-performance parallel computers. Instead, new graph partitioning formulations and algorithms are required to meet the needs of these. In this section, we describe some important classes of scientific simulations that require more generalized formulations of the graph partitioning problem in order to ensure their efficiency on high-performance machines; we discuss these requirements; and we describe new, generalized formulations of the graph partitioning problem as well as algorithms for solving these problem.

**Multiphysics Simulations** In multiphysics simulations, a variety of materials and/or processes are simulated together. The result is a class of problems in which the computation as well as the memory requirements are not uniform across the mesh. Existing partitioning schemes can be used to divide the mesh among the processors such that either the amount of computation or the amount of memory required is balanced across the processors. However, they cannot be used to compute a partitioning that simultaneously balances both of these quantities. Our inability to do so can either lead to significant computational imbalances, limiting efficiency, or significant memory imbalances, limiting the size of problems that can be solved using parallel computers. Figure 24 illustrates this problem. It shows three possible partitionings of a graph in which the amount of computation and memory associated with a vertex can be different throughout the graph. The partitioning in Figure 24(b) balances the computation among the subdomains, but creates a serious imbalance for memory requirements. The partitioning in Figure 24(c) balances the memory requirement, while leaving the computation imbalanced. The partitioning in Figure 24(d), that balances both of these, is the desired solution. In general, multiphysics simulations require the partitioning to satisfy not just one, but a multiple number of balance constraints. (In this case, the partitioning must balance two constraints, computation and memory).

**Multiphase Simulations** Multiphase simulations consist of $m$ distinct computational phases, each separated by an explicit synchronization step. In general, the amount of computation performed for each element of the mesh is different for different phases. The existence of the synchronization steps between the phases requires that each phase be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a decomposition based on this sum. Doing so may lead

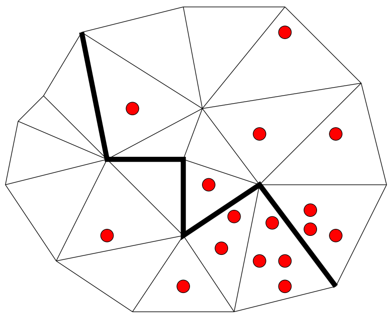


Figure 25: A mesh for a particle-in-cell computation. Here, both the mesh elements and the particles should be balanced across the subdomains.

to some processors having too much work during one phase of the computation (these may still be working after other processors are idle) and not enough work during other phases (these may be idle while other processors are still working). Instead, it is critical that every processor have an equal amount of work from all of the phases of the computation. A traditional partitioning scheme can be used to balance the load across the processors for a single phase of the computation. However, the load may be seriously imbalanced for the other phases. Another method is to use $m$ distinct partitionings, each of which balances the load of a single phase only. This method requires that costly data redistribution be performed after each phase in order to realize the partitioning corresponding to the next phase. A better method is to compute a single partitioning that simultaneously balances the work performed in each of the phases. In this case, no redistribution of the data is necessary, and all of the phases are well balanced.

Figure 25 gives an example. It shows the mesh for a simulation of particles moving through space. This computation is composed of two phases. The first phase is a mesh-based computation. The second phase is a particle-based computation. In order to load balance such an application, each processor must have a roughly equal amount of both the mesh computation and the particle computation. One such bisection is shown. It splits both the mesh elements and the particles in half.

Figure 26 shows another example. This is the mesh associated with the numerical simulation of the ports and the combustion chamber of an internal combustion engine. In this particular problem, the overall computation is performed in six phases. (Each corresponds to a different shade in the figure.) In order to solve such a multiphase computation efficiently on a parallel machine, every processor should contain an equal number of mesh elements of each different shade. Figure 27 shows two subdomains of an eight-way partitioning of the mesh in Figure 26. This partitioning balances all six phases while also minimizing the interprocessor communications. (Note that not all of the shades are visible in Figures 26 and 27.)

**Multimesh Computations** An important class of emerging numerical methods are multimesh computations. Multiple meshes arise in several settings that use grids to discretize partial differential equations. For example, some operations are innately more efficient on structured grids, such as radiation transport sweeps or FFTs. However, complex geometries are better fitted with unstructured meshes. In some simulations, both kinds of grids may be used throughout the computation. Similarly, various codes that solve for multiple physical quantities may use separate grids to solve the appropriate equations for each variable. For example, consider a simulation of the welding of a joint between two parts, a process in which the parts are pressed together and thermally annealed [71]. One grid could be used for the solution of the stress–strain relations that mediate the mechanical deformation of the parts. A second grid could be used to solve the heat equation for thermal conduction in the system. Since the regions of high strain may be distinct from those with high thermal gradients, each grid can be individually tailored to accurately represent the relevant physics.

Now consider the implementation of such a multiphysics example on a distributed-memory parallel machine. A typical time step consists of computing a solution on the first mesh, interpolating the result to the second mesh, computing a solution on the second mesh, interpolating it back to the first mesh, and so on. One way of performing this type of computation in parallel is to partition the meshes separately so that every processor

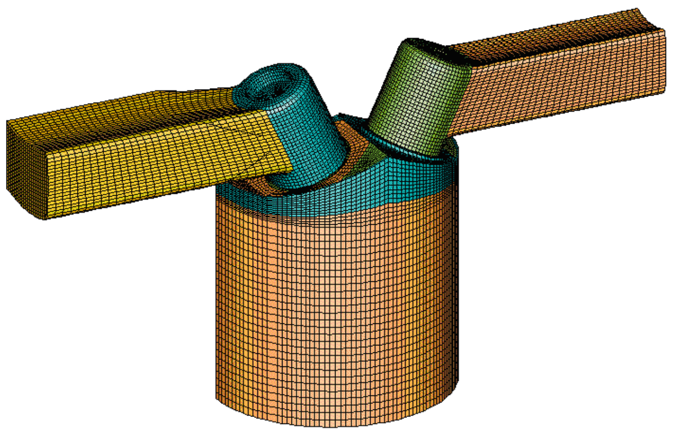


Figure 26: An internal combustion engine simulation is an example application whose computation is performed in multiple phases. Each shade represents elements active during a different phase. (Figure provided by Analysis and Design Application Company Limited.)

has a portion of each mesh. This approach will balance the computations and minimize the communications during each of the solution phases. However, because the different meshes are partitioned independently, there is no assurance that an individual processor will own portions of the meshes that spatially overlap. Therefore, the amount of communication performed during the interpolation and transfer of the solution data can be quite high, even if an efficient approach is used to manage this communication [71]. Ideally, we would like to partition the different meshes such that each processor performs an equal amount of work for every mesh. At the same time, we would like to minimize the amount of interprocessor communications required during the computations of the solutions, as well as that required during the interpolation and transfer of the solutions.

**Domain Decomposition-based Preconditioners** The two keys to the efficient solution of systems of sparse linear equations via iterative methods are (i) the ability to perform the matrix–vector multiplication efficiently in parallel, and (ii) minimizing the number of iterations required for the method to converge. The matrix–vector multiplication is typically implemented by first reordering the sparse matrix to minimize the number of nonzero elements that are off of the block diagonal. Then a striped partitioning of the matrix and the vector is used. Here, an interprocessor communication is required for every nonzero element off of the block diagonal. A high-quality partitioning of the graph corresponding to the sparse matrix provides a reordering such that the number of interprocessor communications is minimized. Use of various preconditioners can minimize the number of iterations required for the solution to converge. There are a number of preconditioning schemes that construct a preconditioner of each block of the block diagonal separately. These are combined to form a preconditioner for the entire matrix. Examples are block-diagonal preconditioners and local ILU preconditioners. These preconditioners ignore the intra-subdomain interactions that are represented by the nonzero elements off of the block diagonal.

Since the matrix reordering is commonly obtained by a graph partitioner, this ensures that the *number* of nonzeros that are ignored in the preconditioner is relatively small. Therefore, the matrix–vector multiplications will be computed efficiently. However, this ordering does not attempt to minimize the *magnitude* of these ignored nonzeros. Therefore, it could be the case that while the number of nonzero elements is small, the sum of the ignored nonzeros is quite large. The consequence of this is that the preconditioner may not be as effective as it could be if the sum of the ignored elements was minimized [56]. That is, the number of iterations for the method to converge may not be minimized. The magnitude of the ignored elements could be minimized directly by a partitioning that is computed using the magnitude of the elements as the edge weights of the graph. However, such an approach will not minimize the communication overhead incurred by the matrix–vector multiplication. This is because an ordering computed in this way would not minimize the number of ignored elements. Ideally, we would like to obtain an ordering that minimizes both the number

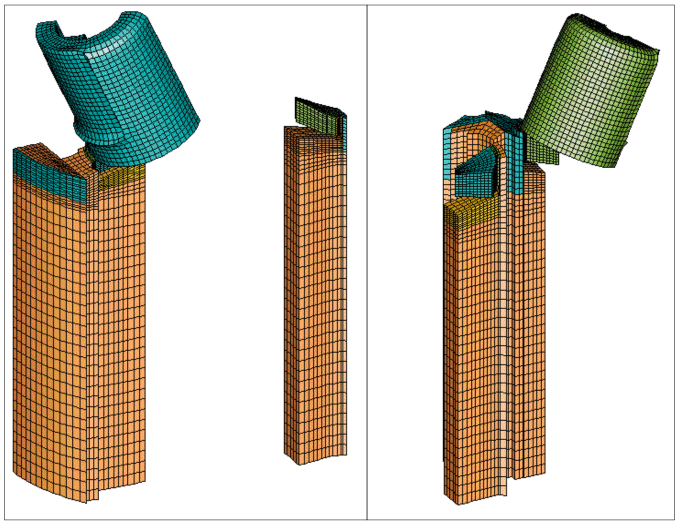


Figure 27: Two subdomains of an eight-way partitioning computed by the multi-constraint graph partitioner implemented in METIS 4.0 are shown. Note, that all of the subdomains have an equal number of elements of each shade (although they are not all visible). (Figure provided by Analysis and Design Application Company Limited.)

of intra-domain interactions (reducing the communication overhead) and the numerical magnitude of these interactions (potentially leading to a better preconditioner).

Figures 28 through 30 illustrate this problem. Figure 28 shows a partitioning of a graph that minimizes the edge-cut and the corresponding matrix ordered with respect to this partitioning. Here, there are only a small number of ignored nonzero entries off of the diagonal. However, their magnitudes are high compared to the other elements. Figure 29 shows a partitioning of a graph that minimizes the magnitude of the ignored entries and the matrix ordered accordingly. Here, there are quite a bit more ignored entries compared to the ordering shown in Figure 28. However, the magnitudes of these entries are small. Figure 30 shows the partitioning that attempts to minimize both the number and the magnitude of the ignored entries as well as the corresponding matrix.

### 0.6.1 A Generalized Formulation for Graph Partitioning

The common characteristic of these problems is that they require the computation of partitionings that satisfy an arbitrary number of balance constraints and/or an arbitrary number of optimization objectives. Traditional graph partitioning techniques have been designed to balance only a single constraint (i.e., the vertex weight) and to minimize only a single objective (i.e., the edge-cut). An extension of the graph partitioning formulation that can model these problems is to assign a weight vector of size $m$ to each vertex and a weight vector of size $l$ to each edge. The problem becomes that of finding a partitioning that minimizes the edge-cuts with respect to all $l$ weights, subject to the constraints that each of the $m$ weights is balanced across the subdomains. This *multi-constraint, multi-objective* graph partitioning problem is able to model all of the problems described above effectively.

For example, the problem of balancing computation and memory can be modeled by associating a vector of size two with each vertex (i.e., a two-constraint problem), where the elements of the vector represent the computation and memory requirements associated with the vertex. Similarly, the problem of computing an ordering for a system of sparse linear systems preconditioned by a block-diagonal method can be modeled by assigning a vector of size two to each edge (i.e., a two-objective problem), where the elements of the vector represent the number of nonzero entries (all ones in this case) and the magnitude of these entries.

Computing decompositions for multimesh computations is a multi-constraint, multi-objective problem. Fig-

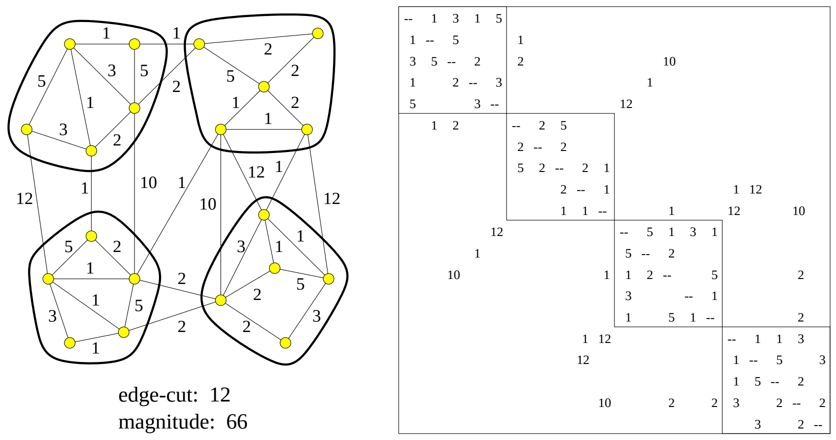


Figure 28: A partitioning of a graph that minimizes the number of edges cut by the partitioning along with the associated sparse matrix ordered accordingly.

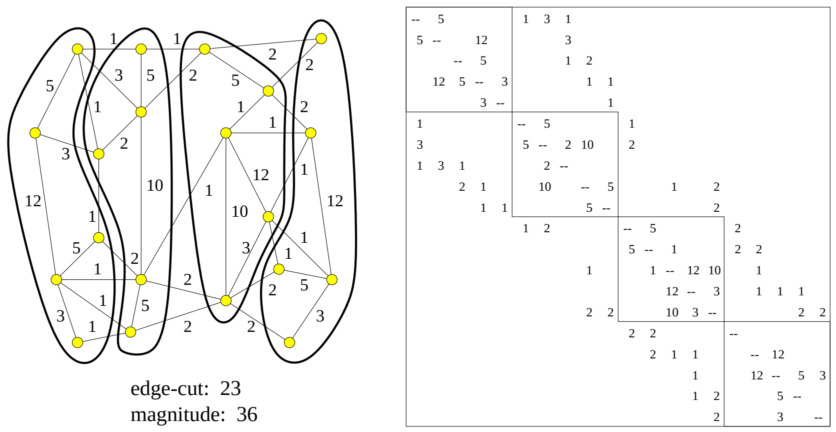


Figure 29: A partitioning of a graph that minimizes the sum magnitude of the edges cut by the partitioning along with the associated sparse matrix ordered accordingly.

ure 31 illustrates an example for a simple case with two meshes. Figure 31(a) shows a pair of overlapping graphs (one with light, circular vertices and dotted edges and the other with dark, square vertices and solid edges). Additionally, dashed lines are included that show the interactions required in order to facilitate the interpolation and transfer process. Figure 31(b) shows the graph that models this problem. Here, the two graphs (and additional edges) from Figure 31(a) are combined. Every square vertex is given a weight of (1, 0) and every circular vertex is given a weight of (0, 1). Solid edges are weighted (1, 0, 0). Dotted edges are weighted (0, 1, 0). Dashed edges are weighted (0, 0, 1). (Note that not all of the vertices and edges are labeled here.) Figure 32 gives a four-way partitioning of this graph. Here, both types of vertices are balanced and their edge-cuts are minimized. At the same time, minimizing the number of dashed edges cut has helped to ensure that regions from the two graphs that spatially overlap tend to be in the same subdomain. This minimizes the communications incurred by the interpolation and transfer process.

**Multi-constraint Graph Partitioning**

Theoretical work relating to multi-constraint graph partitioning includes the Ham-sandwich Theorem and its generalization [89]. This theorem states that a single plane can divide three bounded and connected regions in half in a three-dimensional space. If two of the regions are interpreted as slices of bread and one as a slice of ham, then the conclusion is that a single stroke of a knife can evenly divide the sandwich in two

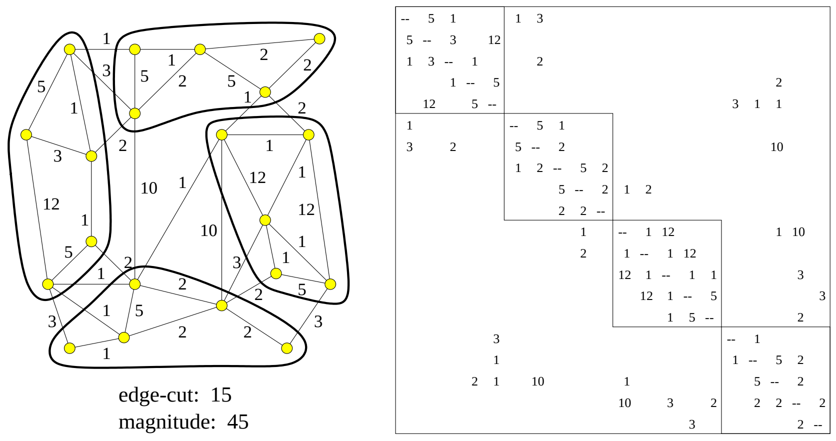


Figure 30: A partitioning that minimizes both the number and the magnitude of the edges cut by the partitioning along with the associated sparse matrix ordered accordingly.

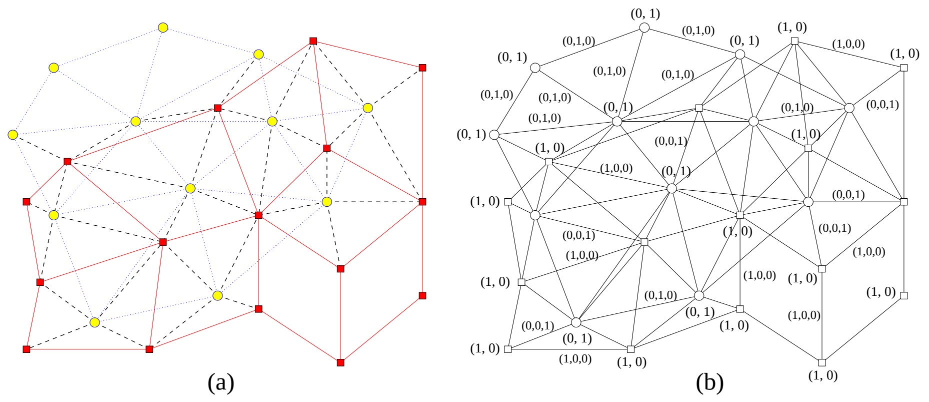


Figure 31: An example of two overlapping meshes (light circles and dark squares) along with dashed interpolation edges (a) and the corresponding multi-constraint multi-objective formulation (b).

so that all three slices are cut exactly in half. Also, Djidjev and Gilbert [19] proved that if a vertex separator theorem holds for a class of graphs (for example, Lipton and Tarjan's planar separator theorem [57]), then the theorem also holds for graphs in which the vertices have an arbitrary number of distinct weights.

Multi-constraint graph partitioning algorithms have recently been developed by a number of researchers [48, 72, 83, 102, 104]. These vary in their generality and complexity. A method is presented in [102] that utilizes a slight modification of a traditional graph partitioner as a black box in order to compute partitionings for multiphase computations. This method partitions disjoint subsets of vertices sequentially. Vertices are grouped together depending on the first phase of the multiphase computation in which they are active. After a set of vertices is partitioned, their subdomains are locked. Subsequent partitioning of other sets of vertices are influenced by the locked vertices. In this way, free vertices that are highly connected to locked vertices are likely to be assigned to the same subdomains as their neighbors. This scheme is sufficient for partitioning the multiphase mesh shown in Figure 26.

A more complex and more general algorithm is presented in [48]. This is a multilevel scheme that extends the coarsening and refinement phases to handle multiple balance constraints. A key component of this algorithm is the initial partitioning algorithm. Here, a partitioning needs to be computed that balances multiple constraints. The authors present a lemma that proves that a set of two-weight objects can be partitioned into two disjoint subsets such that the difference between either of the weights of the two sets is bounded by twice the maximum weight of any object. They further show that this bound can be generalized

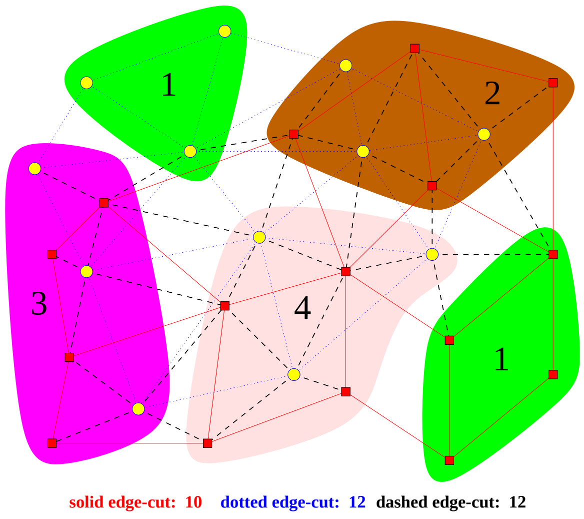


Figure 32: The partitioned meshes from Figure 31.

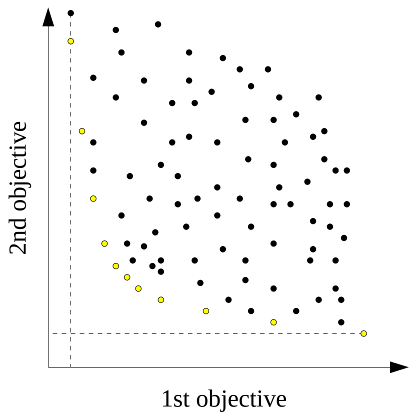


Figure 33: A number of solution points for a two-objective optimization problem. The lightly-shaded points are Pareto optimal.

to $m$ weights. However, maintaining the weight bound depends on the presence of sufficiently many objects with certain weight characteristics (an assumption that usually holds for medium- to large-size graphs). The lemma leads to an algorithm for computing such a bisection. This scheme is sufficient for a wide range of multiphase, multiphysics, and multimesh simulations (including all of the examples described in this section).

A parallel formulation of the multi-constraint partitioner [48] is described in [83]. Experimental results show that this formulation can efficiently compute partitionings of similar quality to the serial algorithm and scales to very large graphs. For example, the parallel multi-constraint graph partitioner is able to compute a three-constraint 128-way partitioning of a 7 million vertex graph in about 7 seconds on 128 processors of a Cray T3E.

### Multi-objective Graph Partitioning

For any single-objective optimization problem (such as the traditional graph partitioning problem), an optimal solution exists in the feasible solution space. In multi-objective optimization, there is no single overall optimal solution, although there is an optimal solution for each one of the objectives. Consider the set of solution points for the two-objective optimization problem shown in Figure 33. The optimally minimal

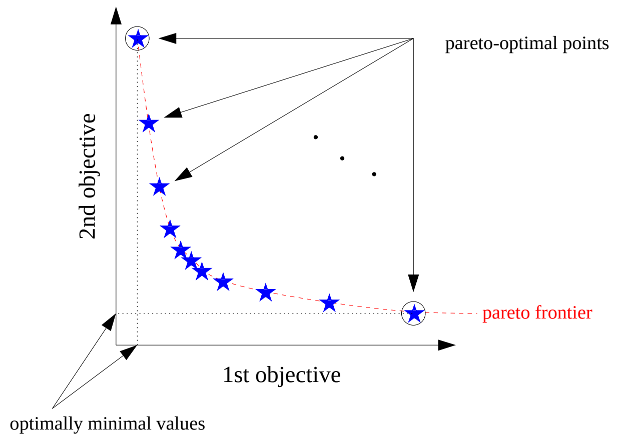


Figure 34: The Pareto frontier for a two-objective optimization problem. The optimally minimal values of each objective are also shown.

values for the two objectives are shown by the dashed lines. In this set, two unique points have the (same) optimal value for the first objective. However, their values for the second objective differ. Clearly, we would prefer the lightly-shaded point over the black point, as this one is equal with respect to the first objective and has a better (smaller) value for the second objective. In this set of solution points, we can quickly determine that most of the points are not of interest. The solutions that are of interest are those that are not dominated by any other solution, regardless of whether they have optimal values for any of the objectives. These are called the *Pareto-optimal* points. A solution is Pareto-optimal if there is no feasible solution for which one can improve the value of any objective without worsening the value of at least one other objective [58]. In Figure 33, the lightly-shaded points (and only these points) are Pareto-optimal. The set of all of Pareto-optimal points is called the *Pareto frontier* [58]. (See Figure 34.) In general, multi-objective optimization problems have many Pareto-optimal solutions. One of the implications of multiple Pareto-optimal solutions is that the definition of the *desired solution* becomes ambiguous. Every multi-objective optimization scheme requires that some method be used in order to disambiguate the definition of a desired solution. In the context of multi-objective graph partitioning, the user should specify the area along the Pareto frontier in which they are interested, and by doing so, control the trade-offs among the objectives.

The key challenge in solving the multi-objective partitioning problem is to allow the user to control the trade-offs among the different objectives. This is particularly difficult when the objectives are dissimilar in nature, as such objectives cannot readily be combined. A new method of reformulating the multi-objective graph partitioning problem so that it can be solved using a traditional (i.e., single-objective) partitioner is presented in [82]. This method provides the user with a fine-tuned control of the trade-offs among the objectives, results in predictable partitionings, and is able to handle dissimilar objectives. Specifically, the algorithm computes a multi-objective partitioning based on a user-specified preference vector. This vector describes how the trade-offs among the objectives should be enforced. For example, if there are two objectives and the user supplies a preference vector of (1, 1), then the algorithm will allow one objective to move away from its optimal value by some amount only if the other objective moves toward its optimal value by more than that amount. For the case of three objectives with a preference vector of (6, 2, 1), the algorithm will prefer a new solution only if $6x + 2y + z > 0$, where $x$ is the gain with respect to the first objective, $y$ is the gain with respect to the second objective, and $z$ is the gain with respect to the third objective.

A number of multi-constraint and multi-objective graph partitioning algorithms, as well as some of their parallel formulations, have been implemented in the METIS [47] and PARMETIS [53] libraries. Serial and parallel multiphase partitioning algorithms [102] are implemented in the JOSTLE [94] library.

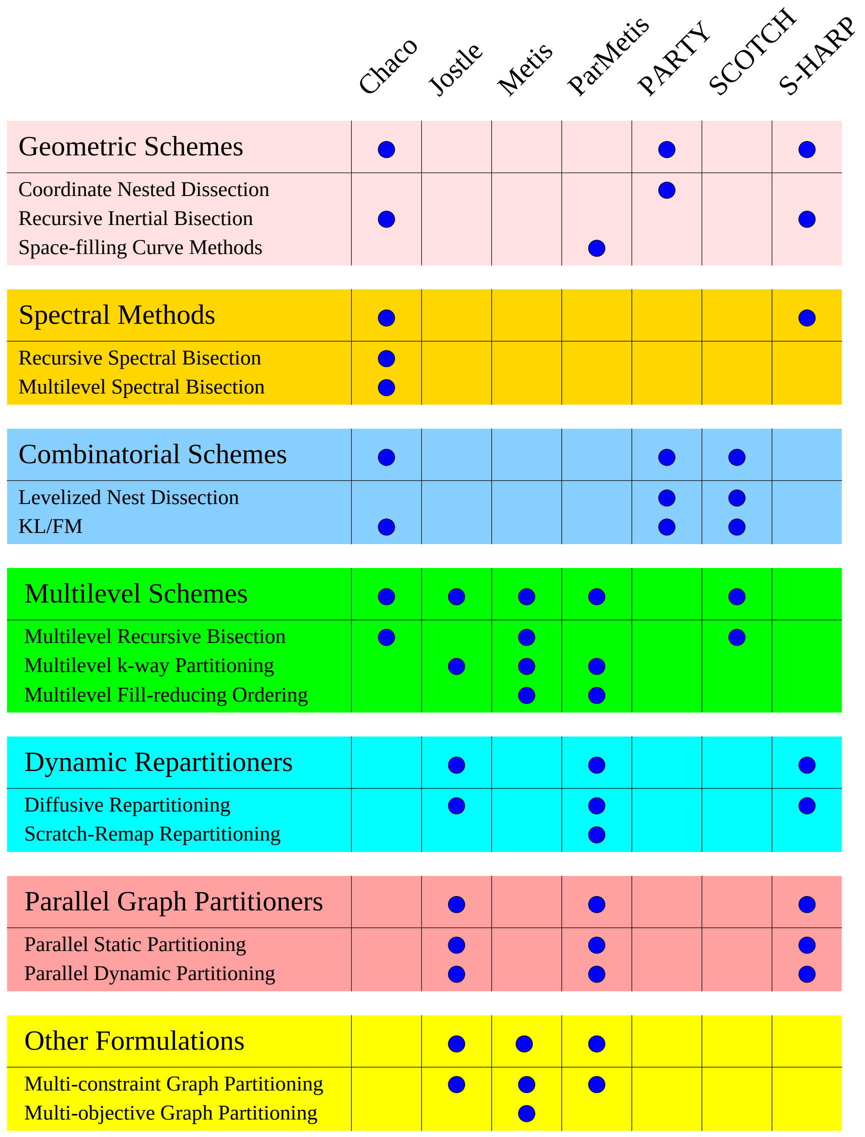
## 0.7 Conclusions

The state of the art in graph partitioning for high-performance scientific simulations has improved dramatically over the past decade. Improvements in the speed, accuracy, generality, and scalability of graph partitioners have led to significant milestones. For example, extremely large graphs (over 0.5 billion vertices) have been partitioned on machines consisting of thousands of processors in only a couple of minutes [54]. However, despite impressive achievements, there is still work to be done in the field. In this section, we discuss some of the limitations of current graph partitioning problem formulations (many of which were highlighted by Hendrickson and Kolda [35]), as well as areas of future work. We end this chapter by charting the functionality of some of the publicly available graph partitioning software packages.

**Limitations of the Graph Partitioning Problem Formulation** As discussed in Section 0.1, the edge-cut metric is not a precise model of the interprocessor communication costs incurred by parallel processing. Nor is it even a precise model of the total communications volume [33]. While the min-cut formulation has proved effective for the well-shaped meshes that are common to scientific simulations, alternative formulations are still needed for more general cases. As an example of recent work in this area, Catalyurek and Aykanat [10] have developed a hypergraph partitioning formulation that precisely models total communication volume. Experimental results comparing the hypergraph partitioning model to the traditional graph partitioning model show that for graphs of nonuniform degree, using the hypergraph model can significantly decrease the interprocessor communication costs compared to using the graph model. However, for graphs of uniform degree, the hypergraph model provides only a modest improvement and requires more runtime compared to state-of-the-art graph partitioners [10]. While the hypergraph partitioning formulation allows us to precisely minimize communications volume, it does not allow us to minimize other important components of interprocessor communication cost such as the message start-up time or the time required for the processor with the most communication (i.e., minimize the maximum processor communication time). Developing new formulations and algorithms that do so is an open area of research in the field.

**Other Application Modeling Limitations** In addition to being imprecise, the traditional partitioning formulation is inadequate for many important classes of scientific simulation. For example, the standard graph partitioning formulation can effectively model only square, symmetric sparse matrices. However, general rectangular and unsymmetric matrices are required for solving linear systems, least squares problems, and linear programs [34]. Bipartite graph partitioning [34] and multi-constraint graph partitioning [48, 83] can be effective for these types of applications. Also, minimizing the edge-cut of a partitioning does not ensure the *numerical scalability* of iterative methods. Numerical scalability means that as the number of processors increases, the convergence rate of the iterative solver remains constant. Vanderstraeten, Keunings, and Farhat [91] have shown that the numerical scalability of a class of iterative solvers can be maintained if partitionings are computed such that their subdomains have low average aspect ratios. The traditional partitioning formulation does not optimize subdomain aspect ratios. Walshaw, Cross, Diekmann, and Schlimbach developed graph partitioning schemes that attempt to minimize the average aspect ratio of the subdomains [98]. Experimental results show that these schemes are able to compute partitionings with significantly better subdomain aspect ratios than traditional partitioners. However, they often result in worse edge-cuts. While these results are promising, it is desirable for a partitioning to minimize both of these objectives (edge-cut and aspect ratio) simultaneously. Recent work in multi-objective graph partitioning [82] may also be relevant here to control the trade-off between these two objectives.

**Architecture Modeling Limitations** When traditional graph partitioners are used for mapping computations onto parallel machines, there is an assumption that the target architecture is flat and homogeneous [17, 35]. While it is true that many current architectures display similar computing powers, bandwidths, and latencies regardless of the processors involved, heterogeneous and hierarchical architectures are becoming increasingly commonplace. For example, consider the problem of decomposing a mesh for parallel processing on an architecture that consists of a cluster of heterogeneous workstations connected by a high-speed, high-latency network to a distributed-memory multiprocessor in which each node consists of a four-processor shared-memory machine. Here, both the computational and communicational speeds depend on the specific



| | Chaco | Jostle | Metis | ParMetis | PARTY | SCOTCH | S-HARP |
|---|---|---|---|---|---|---|---|
| Geometric Schemes | ● | | | | ● | | ● |
| Coordinate Nested Dissection | | | | | ● | | |
| Recursive Inertial Bisection | ● | | | | | | ● |
| Space-filling Curve Methods | | | | ● | | | |
| Spectral Methods | ● | | | | | | ● |
| Recursive Spectral Bisection | ● | | | | | | |
| Multilevel Spectral Bisection | ● | | | | | | |
| Combinatorial Schemes | ● | | | | ● | ● | |
| Levelized Nest Dissection | | | | | ● | ● | |
| KL/FM | ● | | | | ● | ● | |
| Multilevel Schemes | ● | ● | ● | ● | | ● | |
| Multilevel Recursive Bisection | ● | | ● | | | ● | |
| Multilevel k-way Partitioning | | ● | ● | ● | | | |
| Multilevel Fill-reducing Ordering | | | ● | ● | | | |
| Dynamic Repartitioners | | ● | | ● | | | ● |
| Diffusive Repartitioning | | ● | | ● | | | ● |
| Scratch-Remap Repartitioning | | | | ● | | | |
| Parallel Graph Partitioners | | ● | | ● | | | ● |
| Parallel Static Partitioning | | ● | | ● | | | ● |
| Parallel Dynamic Partitioning | | ● | | ● | | | ● |
| Other Formulations | | ● | ● | ● | | | |
| Multi-constraint Graph Partitioning | | ● | ● | ● | | | |
| Multi-objective Graph Partitioning | | | ● | | | | |

Figure 35: A chart illustrating the functionality of a number of publicly available software packages.

processors involved. Standard graph partitioners do not take such considerations into account when computing a partitioning. Partitioning for heterogeneous and hierarchical architectures is especially important in meta-computing environments [43]. In such an environment, it may be impossible to predict the type (or types) of machines or even the exact number of processors that a simulation will be executed on until immediately prior to execution. In this case, both computational speeds and communication costs can fluctuate widely, even between repeated executions of the same simulation. Alternative (e.g., hierarchical and other [11, 90, 101]) partitioning methods are starting to be applied to such problems, but more work still needs to be done.

**Functionality of Available Graph Partitioning Packages** Many of the graph partitioning schemes described in this chapter have been implemented in publicly available software packages. Figure 35 charts the functionality of some the more widely used packages. These are Chaco [36], JOSTLE [94], MeTiS [47], ParMeTiS [53], PARTY [76], SCOTCH [68], and S-HARP [87].

**Acknowledgments**

The authors would like to thank Rupak Biswas, Bruce Hendrickson, Abani Patra, Robert Preis, and Chris Walshaw as well as the CRPC book reviewers for their insightful comments on earlier drafts of this chapter. This work was supported by DOE contract number LLNL B347881, by NSF grants CCR-9972519, EIA-9986042, and ACI-9982274, by Army Research Office contracts DA/DAAG55-98-1-0441, by Army High Performance Computing Research Center cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Additional support was provided by the IBM Partnership Award, and by the IBM SUR equipment grant. Access to computing facilities was provided by AHPCRC and the Minnesota Supercomputer Institute.

# *Bibliography*

[1] C. Ashcraft and J. Liu. A partition improvement algorithm for generalized nested dissection. Technical Report BCSTECH-94-020, York University, North York, Ontario, Canada, 1994.

[2] C. Ashcraft and J. Liu. Using domain decomposition to find graph bisectors. Technical report, York University, North York, Ontario, Canada, 1995.

[3] S. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proc. Supercomputing '95*, 1995.

[4] S. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[5] E. Barnes, A. Vannelli, and J. Walker. A new heuristic for partitioning the nodes of a graph. *SIAM Journal on Discrete Mathematics*, 1:299–305, 1988.

[6] M. Berger and S. Bokhari. Partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[7] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

[8] J. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2:289–313, 1990.

[9] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[10] U. Catalyurek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[11] J. Chen and V. Taylor. ParaPART: Parallel mesh partitioning tool for distributed systems. In *Proc. IRREGULAR'99*, 1999.

[12] Y. Chung and S. Ranka. Mapping finite element graphs on hypercubes. *Journal of Supercomputing*, 6:257–282, 1992.

[13] J. Cong and M. Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in vlsi design. In *Proc. ACM/IEEE Design Automation Conference*, pages 755–760, 1993.

[14] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[15] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25:789–812, 1999.

[16] R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. In D. Hsu, A. Rosenberg, and D. Sotteau, editors, *Interconnection Networks and Mapping and Scheduling Parallel Computations*, volume 21, pages 57–73. AMS Publications, DIMACS Volume Series, 1995.

[17] R. Diekmann, B. Monien, and R. Preis. Load balancing strategies for distributed memory machines. *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools*, 1998.

[18] P. Diniz, S. Plimpton, B. Hendrickson, and R Leland. Parallel algorithms for dynamically partitioning unstructured grids. *Proc. 7th SIAM Conf. Parallel Proc.*, pages 615–620, 1995.

[19] H. Djidjev and J. Gilbert. Separators in graphs with negative and multiple vertex weights. *Algorithmica*, 23:57–71, 1999.

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

[21] H. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

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

[23] J. Gilbert, G. Miller, and S. Teng. Geometric mesh partitioning: Implementation and experiments. In *Proceedings of International Parallel Processing Symposium*, 1995.

[24] J. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, pages 498–513, 1987.

[25] T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. Technical Report UMSI-94-29, University of Minnesota Supercomputing Institute, 1994.

[26] A. Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. *IBM Journal of Research and Development*, 41(1/2):171–183, 1996.

[27] W. Hager and Y. Krylyuk. Graph partitioning and continuous quadratic programming. *SIAM Journal on Discrete Mathematics*, To appear, 1999.

[28] W. Hager, S. Park, and T. Davis. Block exchange in graph partitioning. In P. Pardalos, editor, *Approximation and Complexity in Numerical Optimization: Continuous and Discrete Problems*. Kluwer Academic Publishers, 1999.

[29] K. Hall. An $r$-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, 1970.

[30] S. Hauck and G. Borriello. An evaluation of bipartitioning technique. In *Proc. Chapel Hill Conference on Advanced Research in VLSI*, 1995.

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

[32] G. Heber, R. Biswas, and G. Gao. Self-avoiding walks over adaptive unstructured grids. *Concurrency: Practice and Experience*, to appear 2000.

[33] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Proc. Irregular'98*, pages 218–225, 1998.

[34] B. Hendrickson and T. Kolda. Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput. (to appear)*, 1999.

[35] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing. *Parallel Computing (to appear)*, 2000.

[36] B. Hendrickson and R. Leland. The chaco user's guide, version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.

[37] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proceedings of Supercomputing '95*. ACM Press, December 1995.

[38] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16:452–469, 1995.

[39] D. Hilbert. Uber die stetige abbildung einer linie auf ein flachenstuck. *Math Annalen*, 38, 1891.

[40] G. Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 9:209–218, 1993.

[41] Y. Hu and R. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25:417–444, 1999.

[42] Y. Hu, R. Blake, and D. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience*, 10:467–483, 1998.

[43] W. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of nasa's information power grid. In *Proc. Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.

[44] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proceedings of Supercomputing '95*, 1995.

[45] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[46] G. Karypis and V. Kumar. h MeTiS 1.5: A hypergraph partitioning package. Technical report, Dept. of Computer Science and Engineering, Univ. of Minnesota, 1998.

[47] G. Karypis and V. Kumar. MeTiS 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Dept. of Computer Science and Engineering, Univ. of Minnesota, 1998.

[48] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of Supercomputing '98*, 1998.

[49] G. Karypis and V. Kumar. Multilevel $k$-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[50] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[51] G. Karypis and V. Kumar. Multilevel $k$-way hypergraph partitioning. In *Proceedings of the Design and Automation Conference*, 1999.

[52] G. Karypis and V. Kumar. Parallel multilevel $k$-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[53] G. Karypis, K. Schloegel, and V. Kumar. ParMeTiS: Parallel graph partitioning and sparse matrix ordering library. Technical report, Dept. of Computer Science and Engineering, Univ. of Minnesota, 1997.

[54] J. Keasler. Partitioning challenges in ale3d, 1999. Talk presented at the Workshop on Graph Partitioning and Applications: Current and Future Directions, AHPCRC, MN.

[55] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[56] D. Keyes, 1998. Personal communications.

[57] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[58] M. Makowski. Methodology and a modular tool for multiple criteria analysis of lp models. Technical Report WP-94-102, IIASA, 1994.

[59] G. Miller, S. Teng, W. Thurston, and S. Vavasis. Automatic mesh partitioning. In A. George, John R. Gilbert, and J. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms.* IMA Volumes in Mathematics and its Applications. Springer-Verlag, 1993.

[60] G. Miller and S. Vavasis. Density graphs and separators. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–336, 1991.

[61] B. Monien, R. Preis, and R. Diekmann. Quality matching and local improvement for multilevel graph-partitioning. Technical report, University of Paderborn, 1999.

[62] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In A. K. Noor, editor, *American Soc. Mech. Eng*, pages 291–307, 1986.

[63] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[64] C. Ou and S. Ranka. Parallel incremental graph partitioning using linear programming. *Proceedings Supercomputing '94*, pages 458–467, 1994.

[65] C. Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular and adaptive problems. *Journal of Supercomputing*, 10:119–140, 1996.

[66] B. Parlett, H. Simon, and L. Stringer. On estimating the largest eigenvalue with the lanczos algorithm. *Mathematics of Computation*, 38(137):153–165, 1982.

[67] A. Patra and D. Kim. Efficient mesh partitioning for adaptive *hp* finite element methods. In *International Conference on Domain Decomposition Methods*, 1998.

[68] F. Pellegrini and J. Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *HPCN-Europe, Springer LNCS 1067*, pages 493–498, 1996.

[69] J. Pilkington and S. Baden. Partitioning with spacefilling curves. Technical Report CS94-349, Dept. of Computer Science and Engineering, Univ. of California, 1994.

[70] J. Pilkington and S. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. Technical report, Dept. of Computer Science and Engineering, Univ. of California, 1995.

[71] S. Plimpton, B. Hendrickson, and J. Stewart. A parallel rendezvous algorithm for interpolation between multiple grids. In *Proc. Supercomputing '99*, 1999.

[72] A. Poe and Q. Stout. Load balancing 2-phased geometrically based problems. In *Proc. 9th SIAM Conf. Parallel Processing for Scientific Computing*, 1999.

[73] A. Pothen. Graph partitioning algorithms with applications to scientific computing. In D. Keyes, A. Sameh, and V. Venkatakrishnan, editors, *Parallel Numerical Algorithms.* Kluwer Academic Press, 1996.

[74] A. Pothen, H. Simon, and K. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, 1990.

[75] A. Pothen, H. Simon, L. Wang, and S. Barnard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, pages 42–51, 1992.

[76] R. Preis and R. Diekmann. PARTY - a software library for graph partitioning. Technical report, University of Paderborn, 1997.

[77] P. Raghavan. Line and plane separators. Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

[78] P. Raghavan. Parallel ordering using edge contraction. Technical Report CS-95-293, Department of Computer Science, University of Tennessee, 1995.

[79] P. Sadayappan and F. Ercal. Mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, C-36:1408–1424, 1987.

[80] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.

[81] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. Technical Report TR 98-034, Dept. of Computer Science and Engineering, Univ. of Minnesota, 1998.

[82] K. Schloegel, G. Karypis, and V. Kumar. A new algorithm for multi-objective graph partitioning. In *Proc. EuroPar '99*, pages 322–331, 1999.

[83] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. Technical Report TR 99-031, Dept. of Computer Science and Engineering, Univ. of Minnesota, 1999.

[84] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing 2000*, 2000.

[85] H. Simon, A. Sohn, and R. Biswas. HARP: A fast spectral partitioner. In *Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 43–52, 1997.

[86] H. Simon and S. Teng. How good is recursive bisection? *SIAM J. Scientific Computing*, 18(5):1463–1445, 1997.

[87] A. Sohn. S-HARP: A parallel dynamic spectral partitioner. Technical report, Dept. of Computer and Information Science, New Jersey Institute of Technology, 1997.

[88] A. Sohn and H. Simon. JOVE: A dynamic load balancing framework for adaptive computations on an SP-2 distributed-memory multiprocessor. Technical Report 94-60, Dept. of Computer and Information Science, New Jersey Institute of Technology, 1994.

[89] A. Stone and J. Tukey. Generalized "sandwich" theorems. In *The Collected Works of John W. Tukey.* Wadsworth, Inc., 1990.

[90] J. Teresco, M. Beall, J. Flaherty, and M. Shephard. Hierarchical partition model for adaptive finite element computation. Technical report, Dept. of Computer Science, Rensselaer Polytechnic Institute, 1998.

[91] D. Vanderstraeten, R. Keunings, and C. Farhat. Beyond conventional mesh partitioning algorithms and minimum edge cut criterion: Impact on realistic applications. *SIAM: Parallel Processing for Scientific Computing*, pages 611–614, 1995.

[92] R. VanDriessche and D. Roose. Dynamic load balancing of iteratively refined grids by an enhanced spectral bisection algorithm. Technical report, Dept. of Computer Science, K. U. Leuven, 1995.

[93] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan. Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids. *AIAA Journal*, 32:497–505, 1994.

[94] C. Walshaw. Parallel JOSTLE userguide. Technical Report Userguide Version 1.2.9, University of Greenwich, London, UK, 1998.

[95] C. Walshaw and M. Cross. Load-balancing for parallel adaptive unstructured meshes. In M. Cross *et al.*, editor, *Proc. Numerical Grid Generation in Computational Field Simulations*, pages 781–790. ISGG, Mississippi, 1998.

[96] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. Technical Report 99/IM/44, University of Greenwich, London, UK, 1999.

[97] C. Walshaw and M. Cross. Mesh partitioning: a multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.*, (to appear).

[98] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel mesh partitioning for optimising domain shape. Technical Report 98/IM/38, School of Computing and Mathematical Sciences, University of Greenwich, London, UK, 1998.

[99] C. Walshaw, M. Cross, and M. Everett. Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm. Technical Report 95/IM/06, Centre for Numerical Modelling and Process Analysis, University of Greenwich, London, UK, 1995.

[100] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47(2):102–108, 1997.

[101] C. Walshaw, M. Cross, M. Everett, S. Johnson, and K. McManus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In A. Ferreira and J. Rolim, editors, *Proc. Irregular '95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.

[102] C. Walshaw, M. Cross, and K. McManus. Multiphase mesh partitioning. Technical Report 99/IM/51, University of Greenwich, London, UK, 1999.

[103] M. Warren and J. Salmon. A parallel hashed oct-tree n-body algorithm. *Proceedings of Supercomputing '93*, pages 12–21, 1993.

[104] J. Watts, M. Rieffel, and S. Taylor. A load balancing technique for multi-phase computations. *Proc. of High Performance Computing '97*, pages 15–20, 1997.

[105] J. Watts and S. Taylor. A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed Systems*, (to appear).

[106] H. Wolkowicz and Q. Zhao. Semidefinite programming relaxations for the graph partitioning problem. Technical Report CORR Report 96-17, Department of Combinatorics, University of Waterloo, 1996.

[107] C. Xu and F. Lau. The generalized dimension exchange method for load balancing in k-ary ncubes and variants. *Journal of Parallel and Distributed Computing*, 24:72–85, 1995.