Linköping Electronic Articles in
Computer and Information Science
Vol. 3(1998): nr 10

Algorithms for Graph Partitioning: A Survey

Per-Olof Fjällström

Department of Computer and Information Science
Linköping University
Linköping, Sweden

Linköping University Electronic Press
Linköping, Sweden

http://www.ep.liu.se/ea/cis/1998/010/

Published on September 10, 1998 by
Linköping University Electronic Press
581 83 Linköping, Sweden

Linköping Electronic Articles in
Computer and Information Science
ISSN 1401-9841
Series editor: Erik Sandewall

Typeset by the author using LaTeX
Formatted using étendu *style*

Recommended citation:
<Author>. <Title>. Linköping Electronic Articles in Computer and Information Science, Vol. 3(1998): nr 10.
http://www.ep.liu.se/ea/cis/1998/010/. *September 10, 1998.*

This URL will also contain a link to the author's home page.

Abstract

The *graph partitioning* problem is as follows.

> Given a graph $G = (N, E)$ (where N is a set of weighted nodes and E is a set of weighted edges) and a positive integer p, find p subsets $N_1, N_2, \ldots, N_p$ of N such that
>
> 1. $\cup_{i=1}^{p} N_i = N$ and $N_i \cap N_j = \emptyset$ for $i \neq j$,
> 2. $W(i) \approx W/p$, $i = 1, 2, \ldots, p$, where $W(i)$ and W are the sums of the node weights in N_i and N, respectively,
> 3. the *cut size*, i.e., the sum of weights of edges crossing between subsets is minimized.

This problem is of interest in areas such as VLSI placement and routing, and efficient parallel implementations of finite element methods. In this survey we summarize the state-of-the-art of sequential and parallel graph partitioning algorithms.

Keywords: Graph partitioning, sequential algorithms, parallel algorithms.

The work presented here is funded by CENIIT (the Center for Industrial Information Technology) at Linköping University.

1 Introduction

The *graph partitioning* problem is as follows.

> Given a graph $G = (N, E)$ (where N is a set of weighted nodes and E is a set of weighted edges) and a positive integer p, find p subsets $N_1, N_2, \ldots, N_p$ of N such that
>
> 1. $\cup_{i=1}^{p} N_i = N$ and $N_i \cap N_j = \emptyset$ for $i \neq j$,
> 2. $W(i) \approx W/p$, $i = 1, 2, \ldots, p$, where $W(i)$ and W are the sums of the node weights in N_i and N, respectively,
> 3. the *cut size*, i.e., the sum of weights of edges crossing between subsets is minimized.

Any set $\{N_i \subseteq N : 1 \leq i \leq p\}$ is called a *p-way partition* of N if it satisfies condition (1). (Each N_i is then a *part* of the partition.) A *bisection* is a 2-way partition. A partition that satisfies condition (2) is a *balanced* partition. See Figure 1.

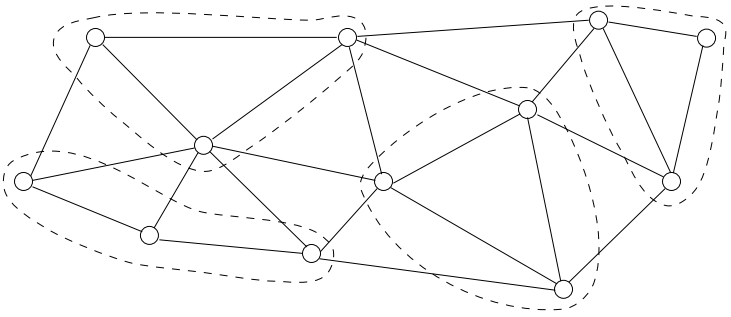

Figure 1: Example of a graph partitioned into four parts.

The graph partitioning problem is of interest in areas such as VLSI placement and routing [AK95], and efficient parallel implementations of finite element methods [KGGK94]. Since the latter application is the main motivation for this survey, let us illustrate the need for graph partitioning by an example from this area.

Suppose that we want to solve a heat conduction problem on a two-dimensional domain. To solve this problem numerically, the original problem is replaced by a discrete approximation. Finite element methods do this by partitioning the domain into *finite elements*, that is, simple convex regions such as triangles (or tetrahedra for three-dimensional domains). The boundary of a finite element is composed of *nodes, edges* and *faces*. The domain partition can be represented by a graph. The *finite element graph* has one node for each node in the partition, and one edge for each edge in the partition.

The temperature at the nodes of the partition can be found by solving a system of linear equations of the form

$$Ku = f,$$

where u and f are vectors with one element for each node, and K is a matrix such that element $k_{i,j}$ is nonzero only if nodes i and j share an edge in the domain partition. An iterative method solves such a system by repeatedly computing a vector y, such that $y = Kx$. We note that to compute the value of y at node i, we only need to know the value of x at the nodes that are neighbors to node i in the finite element graph.

To parallelize the matrix-vector multiplication, we can assign each node to a processor. That is, the value of x and y at a node i, and the nonzero elements of the i-th row of K are stored in a specific processor. This mapping of nodes to processors should be done such that all processors have the same *computational load*, that is, they do about the same number of arithmetic operations. We see that the number of arithmetic operations to compute value of y at node i is proportional to the degree of node i.

Besides balancing the computational load, we must also try to reduce communication time. To compute the value of y at node i, a processor needs the values of x at the neighbors of node i. If we have assigned a neighbor to another processor, the corresponding value must be transferred over the processor interconnection network. How much time this takes depends on many factors. However, the cut size, i.e., the number of edges in the finite element graph whose end nodes have been mapped to different processors, will clearly influence the communication time. (The communication time also depends on factors such as *dilation*, the maximum number of edges of the processor interconnection network that separates two adjacent nodes in the finite element graph, and *congestion*, the maximum number of values routed over any edge of the interconnection network.)

As the above example shows, the graph partitioning problem is an approximation of the problem that we need to solve to parallelize finite element methods efficiently. For this and other reasons, researchers have developed many algorithms for graph partitioning. It should be observed that this problem is NP-hard [GJS76]. Thus, it is unlikely that there is a polynomial-time algorithm that always finds an optimal partition. Algorithms that are guaranteed to find an optimal solution (e.g., the algorithm of by Karisch et al. [KRC97]) can be used on graphs with less than hundred nodes but are too slow on larger graphs. Therefore, all practical algorithms are heuristics that differ with respect to *cost* (time and memory space required to run the algorithm) and *partition quality*, i.e., cut size.

In the above example the graph does not change during the computations. That is, neither N, E nor the weights assigned to nodes and edges change, and consequently there is no need to partition the graph more than once (for a fixed value of p). We call this the *static* case. However, in some applications the graph changes incrementally from one phase of the computation to another. For example, in adaptive finite element methods, nodes and edges may be added or removed during the computations. In this so-called *dynamic* case,

graph partitioning needs to be done repeatedly. Although doing this using the same algorithms as for the static case would be possible, we can probably do better with algorithms that consider the existing partition. Thus, for this situation we are interested in algorithms that, given a partition of a graph G, compute a partition of a graph G' that is "almost" identical to G. We call such algorithms *repartitioning* algorithms.

Recently, researchers have begun to develop parallel algorithms for graph partitioning. There are several reasons for this. Some sequential algorithms give high-quality partitions but are quite slow. A parallelization of such an algorithm would make it possible to speed up the computation. Moreover, if the graph is very large or has been generated by a parallel algorithm, using a sequential partitioning algorithm would be inefficient and perhaps even impossible. Another reason is the need for dynamic graph partitioning. As already mentioned, this involves repeated graph partitioning. Again, to do this sequentially would be inefficient, so we require a parallel algorithm.

The goal of this survey is to summarize the state-of-the-art of graph partitioning algorithms. In particular, we are interested in parallel algorithms. The organization of this report is as follows. In Section 2, we describe some sequential algorithms. In Section 3, we consider parallel partitioning and repartitioning algorithms. Section 4, finally, offers some conclusions.

2 Sequential algorithms for graph partitioning

Researchers have developed many sequential algorithms for graph partitioning; in this section we give brief descriptions of some of these algorithms. We divide the algorithms into different categories depending on what kind of input they require. The section ends with a summary of experimental comparisons between algorithms. We also give references to some available graph partitioning software packages.

2.1 Local improvement methods

A local improvement algorithm takes as input a partition (usually a bisection) of a graph G, and tries to decrease the cut size, e.g., by some local search method. Thus, to solve the partitioning problem such an algorithm must be combined with some method (e.g., one of the methods described in Section 2.2) that creates a good initial partition. Another possibility is to generate several random initial partitions, apply the algorithm to each of them, and use the partition with best cut size. If a local improvement method takes a bisection as input, we must apply it recursively until a p-way partitioning is obtained.

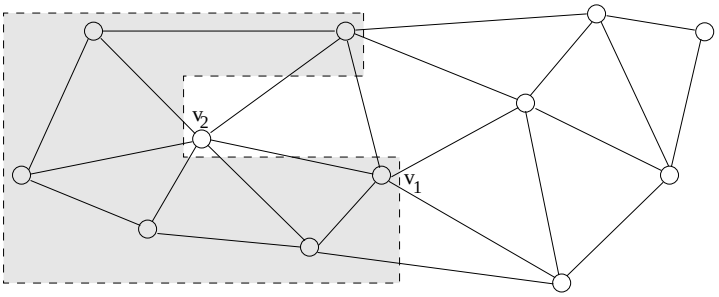

Figure 2: Example of a bisected graph. Nodes in shaded area are in N_1; the others are in N_2. We have $int(v_1) = 2$, $ext(v_1) = 3$, $g(v_1) = 1$, $int(v_2) = 0$, $ext(v_2) = 6$, $g(v_2) = 6$, and $g(v_1, v_2) = 5$. The cut size is 11. (Node and edge weights are assumed to be equal to 1.)

Kernighan and Lin [KL70] proposed one of the earliest methods for graph partitioning, and more recent local improvement methods are often variations on their method. Given an initial bisection, the Kernighan-Lin (KL) method tries to find a sequence of node pair exchanges that leads to an improvement of the cut size.

Let $\{N_1, N_2\}$ be a bisection of the graph $G = (N, E)$. (The node weights are here assumed to be equal to 1.) For each $v \in N$, we define

$$\begin{aligned} int(v) &= \sum_{(v,u)\in E \;\&\; P(v)=P(u)} w(v,u), \\ ext(v) &= \sum_{(v,u)\in E \;\&\; P(v)\neq P(u)} w(v,u), \end{aligned}$$

where $P(v)$ is the index of the part to which node v belongs, and $w(v, u)$ is the weight of edge (v, u). (The total cut size is thus $0.5 \sum_{v\in N} ext(v)$.) The *gain* of moving a node v from the part to which it currently belongs to the other part is

$$g(v) = ext(v) - int(v).$$

Thus, when $g(v) > 0$, we can decrease the cut size by $g(v)$ by moving v. For $v_1 \in N_1$ and $v_2 \in N_2$, let $g(v_1, v_2)$ denote the gain of exchanging v_1 and v_2 between N_1 and N_2. That is,

$$g(v_1, v_2) = \begin{cases} g(v_1) + g(v_2) - 2\, w(v_1, v_2) & \text{if } (v_1, v_2) \in E \\ g(v_1) + g(v_2) & \text{otherwise.} \end{cases}$$

See Figure 2.

One iteration of the KL algorithm is as follows. The input to the iteration is a balanced bisection $\{N_1, N_2\}$. First, we unmark all nodes. Then, we repeat the following procedure n times ($n = \min(|N_1|, |N_2|)$). Find an unmarked pair $v_1 \in N_1$ and $v_2 \in N_2$ for which $g(v_1, v_2)$ is maximum (but not necessarily positive). Mark v_1 and v_2, and update g-values of all the remaining unmarked nodes as

if we had exchanged v_1 and v_2. (Only g-values of neighbors of v_1 and v_2 need to be updated.)

We have now an ordered list of node pairs, (v_1^i, v_2^i), $i = 1, 2, \ldots n$. Next, we find the index j such that $\sum_{i=1}^{j} g(v_1^i, v_2^i)$ is maximum. If this sum is positive, we exchange the first j node pairs, and begin another iteration of the KL algorithm. Otherwise, we terminate the algorithm.

A single iteration of the KL method requires $O(|N|^3)$ time. Dutt [Dut93] has shown how to improve this to $O(|E| \max\{\log |N|, deg_{max}\})$ time, where deg_{max} is the maximum node degree.

Fiduccia and Mattheyses (FM) [FM82] presented a KL-inspired algorithm in which an iteration can be done in $O(|E|)$ time. Like the KL method, the FM method performs iterations during which each node moves at most once, and the best bisection observed during an iteration (if the corresponding gain is positive) is used as input to the next iteration. However, instead of selecting pairs of nodes, the FM method selects single nodes. That is, at each step of an iteration, an unmarked node with maximum g-value is alternatingly selected from N_1 and N_2.

The FM method has been extended to deal with an arbitrary number of parts, and with weighted nodes [HL95c].

Another local improvement method is given by Diekmann, Monien and Preis [DMP94]. This method is based on the notion of *k-helpful sets*: given a bisection $\{N_1, N_2\}$, a set $S \subset N_1$ ($S \subset N_2$) is k-helpful if a move of S to N_2 (N_1) would reduce the cut size by k. Suppose that $S \subset N_1$ is k-helpful, then a set $\bar{S} \subset N_2 \cup S$ is a *balancing set* of S if $|\bar{S}| = |S|$ and $\bar{S}$ is at least $(-k+1)$-helpful. Thus, by moving S to N_2 and then moving $\bar{S}$ to N_1, the cut size is reduced by at least 1.

One iteration of this algorithm is as follows. The input consists of a balanced bisection $\{N_1, N_2\}$ and a positive integer l. First, we search for a k-helpful set S such that $k \geq l$. If no such set exists, we look for the set S with highest helpfulness. If there is no set with positive helpfulness, we set $S = \emptyset$ and $l = 0$. Next, if $S \neq \emptyset$, we search for a balancing set $\bar{S}$ of S. If such a set is found, we move S and $\bar{S}$ between N_1 and N_2, and set $l = 2l$. Otherwise, we set $l = \lfloor l/2 \rfloor$. Finally, unless $l = 0$ we start a new iteration. For further details about how helpful and balancing sets are found, see [DMP94].

Simulated annealing (SA) [KGV83] is a general purpose local search method based on statistical mechanics. Unlike the KL and FM methods, the SA method is not greedy. That is, it is not as easily trapped in some local minima. When using SA to solve an optimization problem, we first select an initial *solution* S, and an initial *temperature* T, $T > 0$. The following step is then performed L times. (L is the so-called *temperature length*.) A random *neighbor* S' to the current

solution S is selected. Let $\delta = q(S) - q(S')$, where $q(S)$ is the quality of a solution S. If $\delta \leq 0$, then S' becomes the new current solution. However, even if $\delta > 0$, S' replaces S with probability $e^{-\delta/T}$. If, after these L iterations, a certain stopping criterion is satisfied, we terminate the algorithm. Otherwise, we set $T = rT$, where r, $0 < r < 1$ is the *cooling ratio*, and another round of L steps is performed.

Johnson et al. [JAMS89] have adapted SA to graph bisectioning and compared it experimentally with the KL algorithm. Although SA performs better than KL for some types of graphs, they conclude that SA is not the best algorithm for graphs that are sparse or has some local structure. This conclusion is supported by the results obtained by Williams [Wil91].

Tabu search [Glo89, Glo90] is another general combinatorial optimization technique. Rolland et al. [RPG96] have successfully used this method for graph bisectioning. Starting with a randomly selected balanced bisection, they iteratively search for better bisections. During each iteration a node is selected and moved from the part to which it currently belongs to the other part. After the move, the node is kept in a so-called *tabu list* for a certain number of iterations. If a move results in a balanced bisection with smaller cut size than any previously encountered balanced bisection, the bisection is recorded as the currently best bisection. If a better bisection has not been found after a fixed number of consecutive moves, the algorithm increases an *imbalance factor*. This factor limits the cardinality difference between the two parts in the bisection. Initially, this factor is set to zero, and it is reset to zero at certain intervals. The selection of which node to move depends on several factors. Moves that would result in a larger cardinality difference than is allowed by the current imbalance factor are forbidden. Moreover, moves involving nodes in the tabu list are allowed only if that would lead to an improvement over the currently best bisection. Of all the allowed moves, the algorithm does the move that leads to the greatest reduction in cut size.

Rolland et al. experimentally compare their algorithm with the KL and SA algorithms, and find that their algorithm is superior both with respect solution quality and running time.

A *genetic algorithm* (GA) [Gol89] starts with an initial set of solutions (*chromosomes*), called a *population*. This population evolves for several generations until some stopping condition is satisfied. A new generation is obtained by selecting one or more pairs of chromosomes in the current population. The selection is based on some probabilistic selection scheme. Using a *crossover* operation, each pair is combined to produce an offspring. A *mutation* operator is then used to randomly modify the offspring. Finally, a replacement scheme is used to decide which offspring will replace which members

of the current population.

Several researchers have developed GAs for graph partitioning [SR90, Las91, BM96]. We briefly describe the bisection algorithm of Bui and Moon [BM96]. They first reorder the nodes: the new order is the order in which the nodes are visited by breadth-first search starting in a random node. Then an initial population consisting of balanced bisections is generated. To form a new generation they select one pair of bisections. Each bisection is selected with a probability that depends on its cut size: the smaller the cut size, the greater the chance of being selected. Once a pair of parents is selected, an offspring (a balanced bisection) is created. (We refer to [BM96] for a description of how this is done.) Next, they try to decrease the cut size of the offspring by applying a variation of the KL algorithm. Finally, a solution in the current population is selected to be replaced by the offspring. (Again, we refer to [BM96] for a description of how this is done.)

Bui and Moon experimentally compare their algorithm with the KL and SA algorithms. They conclude that their algorithm produces partitions of comparable or better quality than the other algorithms.

2.2 Global methods

A global method takes as input a graph G and an integer p, and generates a p-way partition. Most of these methods are *recursive*, that is, they first bisect G (some recursive methods quadrisect or octasect the graph). The bisection step is then applied recursively until we have p subsets of nodes. Global methods are often used in combination with some local improvement method.

2.2.1 Geometric methods

Often, each node of a graph has (or can be associated with) a geometric location. Algorithms that use this information, we call *geometric* partitioning algorithms. Some geometric algorithms completely ignore the edges of the graph, whereas other methods consider the edges to reduce cut size. Node and edge weights are usually assumed to be equal to 1.

The simplest example of a geometric algorithm is *recursive coordinate bisection* (RCB) [BB87]. (This method is closely related to the *multidimensional binary tree* or *k-D* tree data structure proposed by Bentley [Ben75].) To obtain a bisection, we begin by selecting a coordinate axis. (Usually, the coordinate axis for which the node coordinates have the largest spread in values is selected.) Then, we find a plane, orthogonal to the selected axis, that bisects the nodes of the graph into two equal-sized subsets. This involves finding the *median* of a set of coordinate values.

The *inertial* method [FL93] is an elaboration of RCB; instead of selecting a coordinate axis, we select the axis of *minimum angular momentum* of the set of nodes. In three-dimensional space this axis is equal to the eigenvector associated with the smallest eigenvalue of the matrix

$$I = \begin{pmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{pmatrix}$$

where

$$\begin{aligned}
I_{xx} &= \sum_{v \in N} ((y(v) - y_c)^2 + (z(v) - z_c)^2), \\
I_{yy} &= \sum_{v \in N} ((x(v) - x_c)^2 + (z(v) - z_c)^2), \\
I_{zz} &= \sum_{v \in N} ((x(v) - x_c)^2 + (y(v) - y_c)^2), \\
I_{xy} &= I_{yx} = -\sum_{v \in N} (x(v) - x_c)(y(v) - y_c), \\
I_{yz} &= I_{zy} = -\sum_{v \in N} (y(v) - y_c)(z(v) - z_c), \\
I_{xz} &= I_{zx} = -\sum_{v \in N} (x(v) - x_c)(z(v) - z_c), \text{ and} \\
(x_c, y_c, z_c) &= \frac{\sum_{v \in N} (x(v), y(v), z(v))}{|N|},
\end{aligned}$$

where $(x(v), y(v), z(v))$ denotes the coordinates of node v.

Then, we continue exactly as in the RCB method, that is, we find a plane, orthogonal to the axis of minimum angular momentum, that bisects the nodes of the graph into two equal-sized subsets.

The inertial method has been successfully combined with the KL method. That is, at each recursive step, the bisection computed by the inertial method is improved by the KL method [LH94].

Miller, Teng, Thurston and Vavasis [MTTV93] have designed an algorithm that bisects a d-dimensional graph (i.e., a graph whose nodes are embedded in d-dimensional space) by first finding a suitable d-dimensional sphere, and then dividing the nodes into those interior and exterior to the sphere. The sphere is found by a randomized algorithm that involves stereographic projection of the nodes onto the surface of a $(d+1)$-dimensional sphere. More specifically, the algorithm for finding the bisecting sphere is as follows:

1. Stereographically project the nodes onto the $(d+1)$-dimensional unit sphere. That is, node v is projected to the point where the line from v to the north pole of the sphere intersects the sphere.

2. Find the *center point* of the projected nodes. (A center point of a set of points S in d-dimensional space is a point c such that

every hyperplane through c divides S fairly evenly, i.e., in the ratio $d:1$ or better. Every set S has a center point, and it can be found by linear programming.)

3. *Conformally map* the points on the sphere. First, rotate them around the origin so that the center point becomes a point $(0,\ldots,0,r)$ on the $(d+1)$-axis. Second, dilate the points by (1) projecting the rotated points back to d-dimensional space, (2) scaling the projected points by multiplying their coordinates by $\sqrt{(1-r)/(1+r)}$, and (3) stereographically projecting the scaled points to the $(d+1)$-dimensional unit sphere.

   The center point of the conformally mapped points now coincides with the center of the $(d+1)$-dimensional unit sphere.

4. Choose a random hyperplane through the center of the $(d+1)$-dimensional unit sphere.

5. The hyperplane from the previous step intersects the $(d+1)$-dimensional unit sphere in a great circle, i.e., a d-dimensional sphere. Transform this sphere by reversing the conformal mapping and stereographic projection. Use the obtained sphere to bisect the nodes.

A practical implementation of this algorithm is given by Gilbert, Miller and Teng [GMT95]. Their implementation includes a heuristic for computing approximate center points, and a method for improving the balance of a bisection. Moreover, they do not apply the above algorithm to all nodes of the graph but to a randomly selected subset of the nodes. Also, to obtain a good partition several randomly selected hyperplanes are tried. That is, for each hyperplane they compute the resulting cut size, and use the hyperplane that gives the lowest cut size.

Bokhari, Crockett and Nicol [BCN93] describe sequential and parallel algorithms for *parametric binary dissection*, a generalization of recursive coordinate bisection that can consider cut size. More specifically, suppose that we want to bisect the graph using a cut plane orthogonal to the x-axis. The position of this plane is chosen such that $\max(n_l + \lambda \cdot e_l, n_r + \lambda \cdot e_r)$ is minimized. Here, n_l (n_r) is the number of nodes in the subset lying to the left (right) of the plane, e_l (e_r) is the number of the edges with exactly one end node in the left (right) subset, and λ is the parameter.

2.2.2 Coordinate-free methods

In some applications, the graphs are not embedded in space, and geometric algorithms cannot be used. Even when the graph is embedded (or embeddable) in space, geometric methods tend to give relatively high cut sizes. In this section, we describe algorithms that only consider the combinatorial structure of the graph.

The *recursive graph bisection* (RGB) method [Sim91] begins by finding a *pseudo peripheral node* in the graph, i.e., one of a pair of nodes that are approximately at the greatest *graph distance* from each other in the graph. (The graph distance between two nodes is the number of edges on the shortest path between the nodes.) Using breadth-first search starting in the selected node, the graph distance from this node to every other node is determined. Finally, the nodes are sorted with respect to these distances, and the sorted set is divided into two equal-sized sets.

The so-called *greedy method* uses breadth-first search (starting in a pseudo peripheral node) to find the parts one after another [Far88, CJL94].

The *recursive spectral bisection* (RSB) method [PSL90, Sim91] uses the eigenvector corresponding to the second lowest eigenvalue of the *Laplacian matrix* of the graph. (We define the Laplacian matrix L of a graph as $L = D - A$, where D is the diagonal matrix expressing node degrees and A is the adjacency matrix.) This eigenvector (called the *Fiedler vector*) contains important information about the graph: the difference between coordinates of the Fiedler vector provides information about the distance between the corresponding nodes. Thus, the RSB method bisects a graph by sorting its nodes with respect to their Fiedler coordinates, and then dividing the sorted set into two halves. The Fiedler vector can be computed using a modified Lanczos algorithm [Lan50]. RSB has been generalized to quadrisection and octasection, and to consider node and edge weights [HL95b]. The RSB method has been combined with the KL method with good results [LH94].

The *multilevel recursive spectral bisection* (multilevel-RSB) method, proposed by Barnard and Simon [BS93], uses a multilevel approach to speed up the computation of the Fiedler vector. (Experiments show that multilevel-RSB is an order of magnitude faster than RSB, and produces partitions of the same quality.) This algorithm consists of three phases: *coarsening, partitioning,* and *uncoarsening*.

During the coarsening phase a sequence of graphs, $G^i = (N^i, E^i)$, $i = 1, 2, \ldots, m$, is constructed from the original graph $G^0 = (N, E)$. More specifically, given a graph $G^i = (N^i, E^i)$, an approximation G^{i+1} is obtained by first computing I_i, a *maximal independent subset* of N^i. An independent subset of N^i is a subset such that no two nodes in the subset share an edge. An independent subset is maximal if no node can be added to the set. N^{i+1} is set equal to I_i, and E^{i+1} is constructed as follows.

With each node $v \in I_i$ is associated a domain D_v that initially contains only v itself. All edges in E^i are unmarked. Then, as long as there is an unmarked edge $(u, v) \in E^i$ do as follows. If u and v belong to the same domain, mark (u, v) and add it to the domain. If

only one node, say u, belongs to a domain, mark (u, v), and add v and (u, v) to that domain. If u and v are in different domains, say D_x and D_y, then mark (u, v), and add the edge (x, y) to E^{i+1}. Finally, if neither u nor v belongs to a domain, then process the edge at a later stage.

At some point we obtain a graph G^m that is small enough for the Lanczos algorithm to compute the corresponding Fiedler vector (denoted f^m) in a small amount of time. To obtain (an approximation of) f^0, the Fiedler vector of the initial graph, we reverse the process, i.e., we begin the uncoarsening phase. Given the vector f^{i+1}, we obtain the vector f^i by *interpolation* and *improvement*. The interpolation step is as follows. For each $v \in N^i$, if $v \in N^{i+1}$, then $f^i(v) = f^{i+1}(v)$; otherwise $f^i(v)$ is set equal to the average value of the components of f^{i+1} corresponding to neighbors of v in N^i. (We use $f^i(v)$ to denote the component of the vector corresponding to node v.) Next, the vector f^i is improved. This is done by Rayleigh quotient iteration; see [BS93] for further details.

The *multilevel-KL* algorithm [HL95c, BJ93, KK95b, KK95c, Gup97] is another example of how a multilevel approach can be used to obtain a fast algorithm. During the coarsening phase the algorithm creates a sequence of increasingly coarser approximations of the initial graph. When a sufficiently coarse graph has been found, we enter the partitioning phase during which the coarsest graph either is bisected [KK95b] or partitioned into p parts [HL95c, KK95c]. During the uncoarsening phase this partition is propagated back through the hierarchy of graphs. A KL-type algorithm is invoked periodically to improve the partition. In the following, we give a more detailed description of each phase; our descriptions are based on the work of Karypis and Kumar [KK95b, KK95c]. (For an analysis of the multilevel-KL methods, see [KK95a].)

During the coarsening phase, an approximation G^{i+1} of a graph $G^i = (N^i, E^i)$ is obtained by first computing a *maximal matching*, M_i. Recall that a matching is a subset of E^i such that no two edges in the subset share a node. A matching is maximal if no more edges can be added to the matching. Given a maximal matching M_i, G^{i+1} is obtained by "collapsing" all matched nodes. That is, if $(u, v) \in M_i$ then nodes u and v are replaced by a node v' whose weight is the sum of the weights of u and v. Moreover, the edges incident on v' are the union of the edges incident on v and u minus the edge (u, v). Unmatched nodes are copied over to G^{i+1}. See Figure 3.

A maximal matching can be found in several ways. However, experiments indicate that so-called *heavy edge matching* gives best results (see [KK95b, KK95c]). It works as follows. Nodes are visited in random order. If a visited node u is unmatched, we match u with an unmatched neighbor v such that no edge between u and an unmatched neighbor is heavier than the edge (u, v).

During the partitioning phase the coarsest graph G^m either is

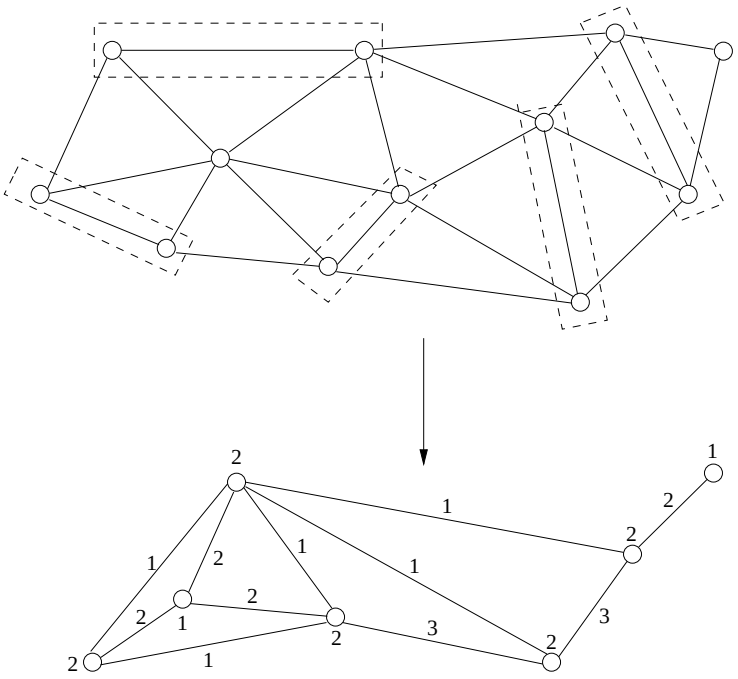

Figure 3: Example of graph coarsening. Node and edge weights on the upper graph (G^i) are assumed to be equal to 1. The numbers close to nodes and edges of the lower graph (G^{i+1}) are node and edge weights.

bisected [KK95b] or partitioned directly into p parts [HL95c, KK95c]. In the former case, we have a multilevel bisection algorithm that needs to be applied recursively to obtain a p-way partition. In the latter case, the graph needs to be coarsened only once.

In [KK95b] several methods for bisecting the coarsest graph were tested, and the best results were obtained for a variant of the greedy algorithm (see p. 10). More specifically, starting in a randomly selected node they grow a part by adding *fringe* nodes, i.e., nodes that currently are neighbors to the part. The fringe node whose addition to the part would result in the largest decrease in cut size is added. This algorithm is executed repeatedly, and the bisection with smallest cut size is used.

During the uncoarsening phase, the partition of G^m is successively transformed into a partition of the original graph G. More specifically, for each node $v \in N^{i+1}$, let $P^{i+1}(v)$ be the index of the part (in the partition of the graph G^{i+1}) to which v belongs. Given P^{i+1}, P^i is obtained as follows. First, an initial partition is constructed using projection: if $v' \in N^{i+1}$ corresponds to a matched pair (u, v) of nodes in N^i, then $P^i(u) = P^i(v) = P^{i+1}(v')$; otherwise $P^i(v') = P^{i+1}(v')$. Next, this initial partition is improved using some variant of the KL method. We describe the so-called *greedy refinement* method from [KK95c].

First, for each $v \in N^i$ and part index $k \neq P^i(v)$ we define

$$\begin{aligned} A^i(v) &= \{P^i(u) : (v,u) \in E^i \,\&\, P^i(u) \neq P^i(v)\}, \\ ext^i(v,k) &= \sum_{(v,u)\in E^i \,\&\, P^i(u)=k} w(v,u), \\ int^i(v) &= \sum_{(v,u)\in E^i \,\&\, P^i(v)=P^i(u)} w(v,u), \\ g^i(v,k) &= ext^i(v,k) - int^i(v). \end{aligned}$$

Moreover, for each part index k we define the corresponding part weight as

$$W^i(k) = \sum_{v\in N^i \,\&\, P^i(v)=k} w(v), \tag{1}$$

where $w(v)$ is the weight of v. A move of a node $v \in N^i$ to a part with index k satisfies the *balance condition* if and only if

$$\begin{aligned} W^i(k) + w(v) &\leq CW/p, \text{ and} \\ W^i(P^i(v)) - w(v) &\geq 0.9W/p, \end{aligned}$$

where C is some constant larger than or equal to 1.

Just as the FM method, the greedy refinement algorithm consists of several iterations. In each iteration, all nodes are visited once in random order. A node $v \in N^i$ is moved to a part with index k, $k \in A^i(v)$, if one of the following conditions is satisfied:

1. $g^i(v,k)$ is positive and maximum among all moves of v that satisfy the balance condition,
2. $g^i(v,k) = 0$ and $W^i(P^i(v)) - w(v) > W^i(k)$.

When a node is moved, the g-values and part weights influenced by the move are updated.

Observe that, unlike the FM method, the greedy refinement method only moves nodes with nonnegative g-values. Experiments show that the greedy refinement method converges within a few iterations.

2.3 Evaluation of algorithms

All of the partitioning methods that we have described have been evaluated experimentally with respect to both partition quality execution time. In this section, we try to summarize these results.

Purely geometric methods, i.e., RCB and the inertial method, are very fast but produce partitions with relatively high cut sizes compared with RSB. The geometric method implemented by Gilbert et al. [GMT95] produces good partitions (comparable with those produced by RSB) when 30 or more random hyperplanes are tried at each bisection step.

Combinatorial algorithms such as RSB and recursive KL (when many random initial partitions are tried) give good partitions but are

relatively slow. The multilevel approach (as used in the multilevel-KL and multilevel-RSB methods) results in much faster algorithms without any degradation in partition quality. (Karypis and Kumar [KK95c] report that their multilevel p-way algorithm produces better partitions than multilevel-RSB, and is up to two orders of magnitudes faster. They compute a 256-way partition of a 448000-node finite element mesh in 40 seconds on a SGI Challenge.) A potential disadvantage of the multilevel approach is that it is memory intensive.

Combining a recursive global method with some local improvement method leads to significantly reduced cut sizes: both the inertial and RSB methods give much better cut sizes if combined with the KL method. Still, the multilevel-KL method gives as good cut sizes as RSB-KL in much less time. Multilevel-KL and RSB-KL give better cut sizes than inertial-KL but are slower than inertial-KL [LH94]. Methods based on helpful sets do not seem competitive with the multilevel-KL method [DMP94].

2.4 Available software packages

In this section we list some available software packages for graph partitioning:

- Chaco, [HL95a]. Developed by Hendrickson and Leland at Sandia National Labs. It contains implementations of the inertial, spectral, Kernighan-Lin, and multilevel-KL methods.
- Metis. Based on the work of Karypis and Kumar [KK95b, KK95c] at the Dept. of Computer Science, Univ. of Minnesota. They have also developed ParMetis that is based on parallel algorithms described in Section 3 [KK96, KK97, SKK97a, SKK97b].
- Mesh Partitioning Toolbox. Developed Gilbert et al., and it includes a Matlab implementation of their algorithm [GMT95].
- JOSTLE, [WCE97a]. Developed by Walshaw et al. at Univ. of Greenwich.
- TOP/DOMDEC, [SF93]. Developed by Simon and Fahrat; it includes implementations of the greedy, RGB, inertial and RSB algorithms.

3 Parallel algorithms for graph partitioning

Most of the algorithms presented in this section are parallel formulations of methods presented in the previous section. Some of these methods, in particular the geometric methods, are relatively easy to parallelize. On the other hand, the KL and SA methods appear to be inherently sequential (they have been shown to be P-complete

[SW91]). (Multiple runs of the KL method can, of course, be done in parallel on different processors [BSSV94].) In this section, several parallel "approximations" of the KL method are presented.

The RSB and multilevel methods are more difficult to parallelize than the geometric methods. These methods involve graph computations (e.g., finding maximal matchings, independent sets, and connected components) that are nontrivial in the parallel case.

3.1 Local improvement methods

The algorithms described in this subsection take as input a partition of a graph, which they then try to improve. More specifically, the first two methods take as input a balanced bisection and try to improve the cut size. Thus, to compute a p-way partition these algorithms need to be applied recursively. (We assume that p does not exceed the number of processors of the parallel computer executing the algorithm.)

The remaining algorithms are repartitioning algorithms, that is, they take as input a p-way partition that may need to be improved both with respect to balance and cut size. As already mentioned, repartitioning algorithms are required, for example, in adaptive finite elements methods where the graph may be refined or coarsened during a parallel computation. In this situation, a part in the partition corresponds to the nodes stored in a processor, and the computational load of a processor is equal to the weight of the corresponding part. An efficient repartitioning algorithm must not only improve the balance and cut size of a given partition, it must also try to achieve this while minimizing the node movements between processors.

Gilbert and Zmijevski [GZ87] proposed one of the earliest parallel algorithms for graph bisection; it is based on the KL algorithm (see p. 4). The algorithm consists of a sequence of iterations, and the input to each iteration is a balanced bisection $\{N_1, N_2\}$. This bisection corresponds to a balanced bisection $\{P_1, P_2\}$ of the set of processors, i.e., if node $v \in N_1$ then the adjacency list of v is stored in a processor belonging to P_1.

The first step in each iteration is to select one processor in each subset of processors as the *leader* of that subset. Then, for each node v, the gain $g(v)$ is computed and reported to the corresponding leader. This is done by the processor storing the adjacency list of node v. The leader unmarks all nodes in its part of the bisection.

The following procedure is then repeated $n = \min(|N_1|, |N_2|)$ times. Each leader selects (among the unmarked nodes in its part of the bisection) a node with largest g-value, and marks this node. (Observe that this selection is different from the KL algorithm since the edge (if any) between the selected nodes is ignored.) The leaders request the adjacency lists of the two selected nodes, and update the g-values of the unmarked nodes adjacent to the selected nodes. When n pairs of nodes have been selected, the leaders decide which

Figure 4: Example of (a) the quotient graph corresponding to the partition in Figure 1 (numbers denote part weights); (b) the tree formed by requests for loads.

node pairs to exchange (if any) using the same procedure as the KL algorithm. Finally, nodes (with their adjacency lists) are exchanged between the subsets of processors, and a new iteration is begun.

Savage and Wloka [SW91] propose the *Mob* heuristic for graph bisection. (This method has also been applied to the graph-embedding problem [SW93].) Their algorithm consists of several iterations. The input to each iteration is a bisection $\{N_1, N_2\}$ and a positive integer s. Two size s mobs, i.e., subsets of N_1 and N_2, are computed, and then swapped. If the swap resulted in an increased cut size, s is reduced according to a predetermined schedule before the next iteration begins.

The selection of a mob of size s is done as follows. First, a so-called *premob* is found. For each node v, let $g(v)$ denote the decrease in cut size caused by moving v to the other part of the bisection. The premob consists of nodes of with g-values at least g_s, where g_s is the largest value such that the premob has at least s nodes. The mob nodes are then selected from the premob by a randomized procedure. Thus, the mobs are not guaranteed to have exactly s nodes.

The Mob heuristic has been implemented on a CM-2; the implementation is based on the use of virtual processors. More specifically, the implementation uses a linear array in which each edge appears twice, e.g., (u, v) and (v, u). The edges are sorted by their left node, and with each edge is stored the address of its twin. A virtual processor is associated with each edge. The first processor associated with a group of edges with the same left node represents this node. Using this data structure it is easy to compute cut sizes and gains, and to select mobs.

Özturan, deCougny, Shephard and Flaherty [OdSF94] give a repartitioning algorithm based on iterative local load exchange between processors. To describe their algorithm, we first introduce *quotient graphs*. Given a partition $\{N_i : 1 \leq i \leq p\}$ of a graph $G = (N, E)$, the quotient graph is a graph $G_q = (V_q, E_q)$ where node $v_i \in V_q$ represents the part N_i, and edge $(v_i, v_j) \in E_q$ if and only if there are nodes $u \in N_i$ and $v \in N_j$ such that $(u, v) \in E$. See Figure 4.

In the algorithm of Özturan et al., there is a one-one correspondence between nodes in G_q and processors. That is, the nodes stored in a processor form a part in the partition. (However, adjacent nodes in the quotient graph need not correspond to processors adjacent in the interconnection network of the parallel computer.) They apply the following procedure repeatedly until a balanced partition is obtained.

1. Each processor informs its neighbors in the quotient graph of its computational load.
2. Each processor that has a neighbor that is more heavily loaded than itself requests load from its most heavily loaded neighbor. These requests form a forest F of trees.
3. Each processor computes how much load to send or receive. The decision which nodes to send is based on node weight and gain in cut size.
4. The forest F is *edge-colored*. I.e., using as few colors as possible, the edges are colored such that no two edges incident on the same node (of the quotient graph) have the same color.
5. For each color, nodes are transferred between processor pairs connected by an edge of that color.

This algorithm was implemented on a MasPar MP-1 computer.

Chung, Yeh, and Liu [CYL95] give a repartitioning algorithm for a 2D torus, i.e., a mesh-connected parallel computer with wrap-around connections. In the following, we assume that the torus has the same number of rows and columns. The algorithm consists of two steps. After the first step, the computational load is equally balanced within each row. After the second step, the computational load is equally balanced within each column (and over all processors). The balancing of a row is done as follows. First, each processor in an even-numbered column balances with its right neighbor, and then with its left neighbor. This is repeated until the row is balanced. The decision which nodes to transfer between neighboring processors is made with the aim to reduce the communication between the processors. Columns are balanced similarly.

The algorithm was implemented and evaluated on a 16-processor NCUBE-2. Its performance was evaluated by applying it to a sequence of refinements of a finite element mesh. The results show that using their repartitioning algorithm is faster than to use a parallel version of RCB (and partition from scratch) at each refinement step.

Ou and Ranka [OR97] propose a repartitioning algorithm based on linear programming. Suppose that a graph has been refined by adding and deleting nodes and edges. A new partition is computed as follows. First, the new vertices are assigned to a part. Next, for each node is determined which neighbor part to which it is closest. Then, the current partition is balanced so that the amount of node movements between parts is minimized. More specifically, let a_{ij} denote the number of nodes in the part N_i that have the part N_j as their closest neighboring part. To compute l_{ij}, the number of nodes that need to be moved from the part N_i to the part N_j to achieve balance, they solve the following linear programming problem:

Minimize

$$\sum_{1 \le i \ne j \le p} l_{ij}$$

subject to

$$0 \le l_{ij} \le a_{ij}$$

$$\sum_{1 \le i \le p} (l_{ji} - l_{ij}) = W(j) - W/p,\ 1 \le j \le p,$$

where $W(i)$ denotes the current weight of part N_i. (For simplicity, node and edge weights are assumed to be of unit value.) The last step in the algorithm of Ou and Ranka is aimed at reducing the cut size, while still maintaining balance. This is also done by solving a linear programming problem.

A sequential version of the above algorithm was compared with the RSB algorithm. It produced partitions of the same quality as the RSB algorithm and was one to two orders of magnitude faster. A parallel implementation on a 32-processor CM-5 was 15 to 20 times faster than the sequential algorithm.

Walshaw, Cross and Everett [WCE97a, WCE97b] describe an iterative algorithm (JOSTLE-D) for graph repartitioning. In this algorithm, they first decide how much weight must be transferred between adjacent parts to achieve balance. They then enter an iterative node migration phase in which nodes are exchanged between parts.

The first step of this algorithm is done using an algorithm of Hu and Blake [HB95] . To decide how much weight must be transferred between adjacent parts, they solve the linear equation

$$L\lambda = b,$$

where L is the Laplacian matrix (see p. 10) of the quotient graph $G_q = (V_q, E_q)$, and $b_i = W(i) - W/p$, where $W(i)$ is the current weight of the nodes in the part N_i. The *diffusion solution*, λ, describes how

much weight must be transferred between parts to achieve balance. More specifically, if the parts N_i and N_j are adjacent, then weight equal to $\lambda_i - \lambda_j$ must be moved from the part N_i to the part N_j.

Experiments carried out on a Sun SPARC Ultra show that JOSTLE-D is orders of magnitude faster than partitioning from scratch using multilevel-RSB.

Schloegel, Karypis and Kumar [SKK97a, SKK97b] present sequential and parallel repartitioning algorithms based on the multilevel approach. Briefly, their algorithms consist of three phases: *graph coarsening, multilevel diffusion*, and *multilevel improvement*.

The coarsening phase is exactly as in [KK95c] (see p. 11) except that only the subgraphs of the given partition are coarsened; this phase is thus essentially parallel. (The *subgraph* corresponding to a subset $N_i \subset N$ is the graph $G_i = (N_i, E_i)$, where $E_i = \{(u, v) \in E : u, v \in N_i\}$.)

The goal of the multilevel diffusion phase is to obtain a balanced partition. This is done by moving boundary nodes (i.e., nodes that are adjacent to some node in another part) from overloaded parts. If balance cannot be obtained (i.e., the boundary nodes are not "fine" enough), the graph is uncoarsened one level, and the process is repeated.

The manner in which nodes are moved between parts is either *undirected* or *directed*. In the first case balance is obtained using only local information. In the second case, the algorithm of Hu and Blake (see p. 18) is used to compute how much weight needs to be moved between neighboring parts.

The goal of the multilevel improvement phase is to reduce the cut size. More specifically, this phase is similar to the uncoarsening phase in [KK97], see p. 26.

Schloegel et al. implemented their repartitioning algorithms on a Cray T3D using the MPI library for communication. The algorithms were evaluated on graphs arising in finite element computations. The experimental results show that, compared with partitioning from scratch using the algorithm in [KK97] (p. 26), the repartitioning algorithms take less time and produce partitions of about the same quality. A graph with eight million nodes can be repartitioned in less than three seconds on a 256-processor Cray T3D.

3.2 Global methods

Parallel global graph partitioning algorithms are primarily useful when we need to partition graphs that are too large to be conveniently partitioned on a sequential computer. In principle, they could be used for dynamic graph partitioning, but the results from the previous section indicate that the dynamic case is better handled with repartitioning algorithms. Throughout this section, p denotes both

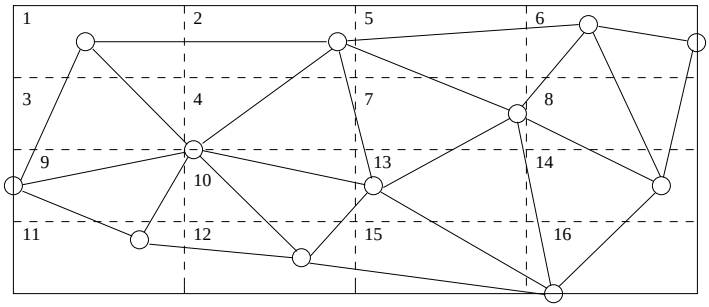

Figure 5: Example of the IBP method using shuffled row-major indexing.

the number of parts in a partition and the number of processors of the parallel computer executing the algorithms.

3.2.1 Geometric methods

The parallel geometric methods presented in this section use well-studied operations such as sorting, reduction, finding medians etc. [KGGK94]. In principle, they are thus easy to implement. However, in particular recursive methods may involve much data movement, and the most efficient way to carry out these movements may depend on the characteristics of the parallel computer.

Ou, Ranka and Fox [ORF96, OR94] propose *index-based partitioning* (IBP). Their method is as follows. First, a hyperrectangular bounding box for the set of nodes is computed. This box is divided into identical hyperrectangular cells. The cells are indexed such that the indexes of cells that are geometrically close are numerically close (they use, e.g., a generalization of shuffled row-major indexing). See Figure 5. The nodes are then sorted with respect to the indexes of the cells in which they are contained. Finally, the sorted list is partitioned into p equal-sized lists. Ou et al. also describe how the graph can be repartitioned if the geometric positions of the nodes change or if nodes are added or deleted.

The algorithm is analyzed for a coarse-grained parallel machine, and experimentally evaluated on a 32-processor CM-5. The experiments show that the IBP method is two to three orders of magnitude faster than a parallel version of the RSB method. However, the cut size is worse.

Jones and Plassman [JP94] propose *unbalanced recursive bisection* (URB). As in RCB (see p. 7), the cut planes are always perpendicular to some coordinate axis. However, they do not require that a cut

plane subdivide the nodes of the graph into two equal-sized subsets. It is sufficient that it subdivides the nodes into subsets whose sizes are integer multiples of $|N|/p$. Let $w(d, S)$ denote the width of a set of points S along coordinate direction d. We define the *aspect ratio* of a set of points S as

$$\max(w(x,S), w(y,S), w(z,S))/\min(w(x,S), w(y,S), w(z,S)).$$

To bisect a graph, Jones and Plassman find a cut plane that yields the smallest maximum aspect ratio for the subsets of nodes lying on either side of the plane. They further subdivide the resulting subsets in proportion to their sizes. That is, if the number of nodes in a subset is $k|N|/p$, for some integer k, $2 \leq k < p$, then it is further subdivided until eventually k equal-sized subsets have been obtained.

Nakhimovski [Nak97] presents an algorithm that is similar to URB except that to bisect a graph, he tries to find a plane that cuts as few edges as possible. However, the number of edges cut by a plane is not calculated exactly but approximated using node coordinates only. Experiments show that this algorithm usually computes partitions of higher quality than standard RCB.

Diniz, Plimpton and Hendrickson [DPH95] give two algorithms based on parallel versions of the inertial method (see p. 8), and the local improvement heuristic of Fiduccia and Mattheyses (see p. 5).

The inertial method is relatively straightforward to parallelize. Initially, all processors collaborate to compute the first bisection. That is, first a bisecting cut plane for the entire graph is computed. Then, the set of processors is also bisected, and all nodes lying on the same side of the cut plane are moved to the same subset of the processors. This continues recursively until there are as many subsets of nodes as there are processors. (See also Ding and Ferraro [DF96].)

Since the FM and KL methods are closely related, the FM method is also inherently sequential. Diniz et al. briefly describe a parallel variant of the FM method in which processors form pairs, and nodes are exchanged only between paired processors.

In the first algorithm of Diniz et al., called *Inertial Interleaved FM* (IIFM), the parallel variant of FM is applied after each bisection performed by the parallel inertial method.

In the second algorithm, *Inertial Colored FM* (ICFM), the graph is first completely partitioned by the parallel inertial method. Then, all pairs of processors whose subsets of nodes share edges, apply FM to improve the partition. More specifically, the quotient graph is edge-colored (see p. 16), and pairwise FM is applied simultaneously to all edges of the same color.

Experiments suggest that ICFM is superior to IIFM both concerning partitioning time and partition quality. ICFM reduces the cut size with about 10% compared with the pure parallel inertial method, but is an order of magnitude slower, in particular when many parts are made.

Al-furaih, Aluru, Goil, Ranka [AfAGR96] give a detailed description and analysis of several parallel algorithms for constructing multidimensional binary search trees (also called *k-D* trees). In particular, they study different methods to find median coordinates. It is often suggested that presorting of the point coordinates along every dimension is better than using a median-finding algorithm at each bisection step. The results of Al-furaih et al. show that randomized or bucket-based median-finding algorithms outperform presorting.

Al-furaih et al. also propose a method to reduce data movements during sorting and median finding. Due to the close relationship between *k-D* trees and the RCB method (see p. 7) their results are highly relevant for parallelization of the RCB and related methods.

3.2.2 Coordinate-free methods

The sequential coordinate-free methods use algorithms for breadth-first search, finding maximal independents sets and matchings, graph coarsening, etc., that are nontrivial to parallelize. The local improvement steps in multilevel methods are also difficult to do efficiently in parallel. The methods presented in this section present various approaches to these problems.

A parallel implementation of the RSB method (see p. 10) on the Connection Machine CM-5 is described by Johan, Mathur, Johnson and Hughes [JMJH93, JMJH94].

Barnard [Bar95] gives a parallel implementation of the multilevel-RSB method (see p. 10) on the Cray T3D. For the coarsening phase, the PRAM algorithm of Luby [Lub86] is used to compute maximal independent sets.

Karypis and Kumar [KK95d] give a parallel version of their multilevel-KL bisection method [KK95b] (see p. 11). Recall that to bisect a graph $G = (N, E)$, they first compute a sequence $G^1, G^2, \ldots$, of successively smaller graphs until they have obtained a sufficiently small graph (the *coarsening* phase). This graph is then bisected, and the resulting bisection is successively projected back to a bisection for G (the *uncoarsening* phase). During the uncoarsening phase, the partition is periodically improved by a version of the KL method.

To describe the parallel version of the coarsening phase, it is convenient to regard the processors as arranged in a two-dimensional array. (We assume that p is an integer power of 4.) The graph G^{i+1} is obtained from the graph $G^i = (N^i, E^i)$ as follows. N^i is assumed to be divided into p_i equal-sized subsets $N^i_1, N^i_2, \ldots, N^i_{p_i}$. Processor $P_{k,l}$, $1 \leq k, l \leq p_i$, contains the graph $G^i_{k,l} = (N^i_k \cup N^i_l, E^i_{k,l})$, where $E^i_{k,l} = \{(u, v) \in E^i : u \in N^i_k \,\&\, v \in N^i_l\}$. See Figure 6. To construct G^{i+1}, each processor $P_{k,k}$ first computes a maximal matching M^i_k of the edges in $G^i_{k,k}$. The union of these local matchings is regarded as the overall matching M^i. Next, each processor $P_{k,k}$ sends M^i_k to

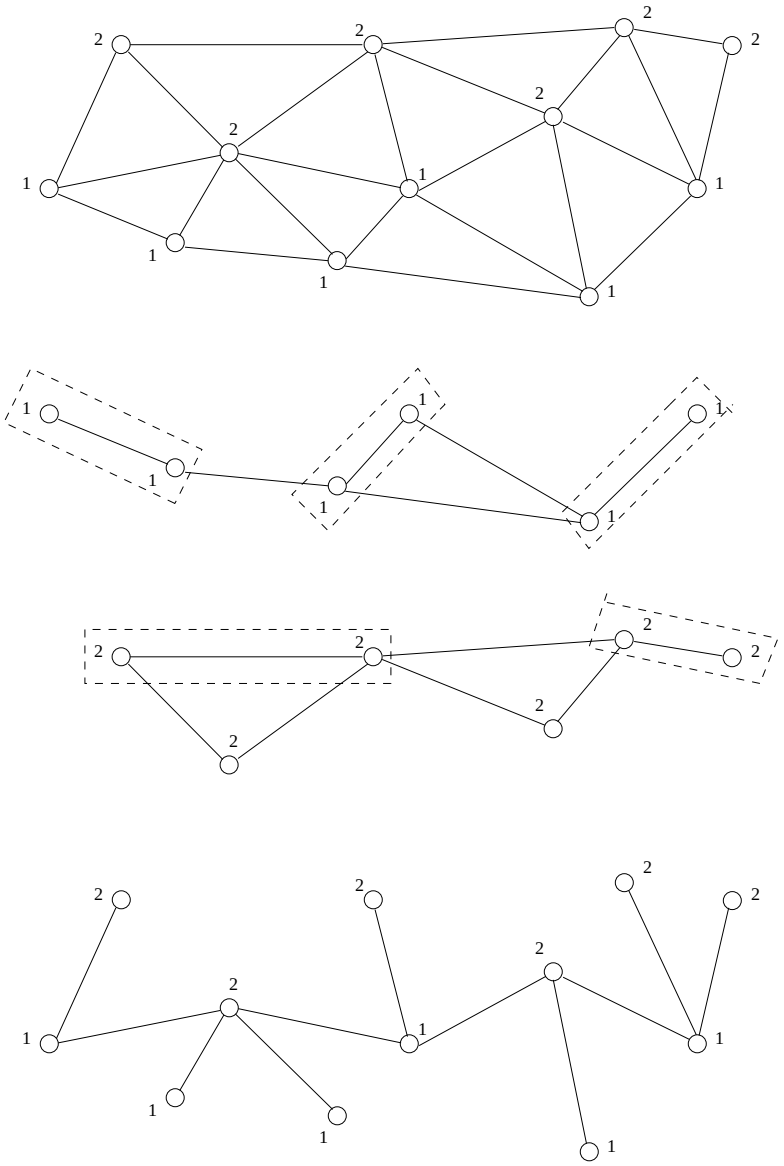

Figure 6: Example of the graph decomposition used in [KK95d] for $p_i = 2$. The graphs in this figure are (from top to bottom): G^i, $G^i_{1,1}$, $G^i_{2,2}$, and $G^i_{1,2} = G^i_{2,1}$. The numbers by the nodes show which subset N^i_j, $j = 1, 2$, they belong to.

each processor in its row and column. Finally, G^{i+1} is obtained by modifying G^i according to the matching.

In the beginning of the coarsening phase, i.e., for small values of i, $p_i = \sqrt{p}$. After a while, when the number of nodes between successive coarser graphs does not substantially decrease, the current graph is copied into the lower quadrant of the processor array. That is, the current value of p_i is halved. In this way, we get more nodes and edges per (active) processor and can find larger matchings. This procedure continues until the current graph is copied to a single processor, after which the coarsening is done sequentially. When a sufficiently small graph has been obtained, it is bisected either sequentially or in parallel.

The uncoarsening phase is simply a reversal of the coarsening phase, except that a local improvement method is applied at each step to improve the current partition. In the following, we describe the parallel implementation of the improvement step.

As in the coarsening phase, we assume that the nodes of G^i are divided into p_i subsets $N_1^i, N_2^i, \ldots, N_{p_i}^i$, and that processor $P_{k,l}$, $1 \leq k, l \leq p_i$ contains the graph $G_{k,l}^i$. We also assume that, for each node $v \in N_l^i$, processor $P_{k,l}$ knows $g_{k,l}^i(v)$, the gain in cut size *within* $G_{k,l}^i$ obtained by moving v from the part to which it currently belongs to the other. The overall gain for node $v \in N_l^i$, $g^i(v)$, can thus be computed by adding along the l-th processor column. We assume that $g^i(v)$, $v \in N_k^i$, is stored in the processor $P_{k,k}$.

The parallel local improvement method consists of several iterations. During an iteration, each diagonal processor $P_{k,k}$ selects from one of the parts a set $U_k \subset N_k^i$ consisting of all nodes with positive g-values. The nodes in U_k are moved to the other part. (This does not mean that they are moved to another processor.) Each diagonal processor then broadcasts the selected set along its row and column, after which gain values can be updated. Unless further improvements in cut size are impossible, or a maximum number of iterations is reached, the next iteration is started. The part from which nodes are moved alternates between iterations. If necessary, the local improvement phase is ended by an explicit balancing iteration.

The above algorithm was implemented on a 128-processor Cray T3D using the SHMEM library for communication. The experimental evaluation shows that, compared with the sequential algorithm [KK95b], the partition quality of the parallel algorithm is almost as good, and that the speedup varies from 11 to 56 depending on properties of the graph. For most of the graphs tested it took less than three seconds to compute an 128-way partition on 128 processors.

Karypis and Kumar [KK96, KK97] give parallel implementations of their multilevel p-way partitioning algorithm [KK95c]. We begin with a description of the algorithm given in [KK96].

Let us first consider how a matching M_i for the graph $G^i =$

(N^i, E^i) is computed. We assume that the nodes in N^i are colored using c_i colors, i.e., for each node $v \in N^i$, the color of v is different from the colors of its neighbors. How such a coloring can be found is described below. We assume also that G^i is evenly distributed over the processors, and that each node N_i (with its adjacency list) is stored in exactly one processor.

The matching algorithm consists of c_i iterations. During the j-th iteration, $j = 1, 2, \ldots, c_i$, each processor visits its local, unmatched nodes of color j. For each such node v an unmatched neighbor u is selected using the heavy edge heuristic. If u is also a local node, v and u are matched. Otherwise, the processor creates a request to match (u, v). When all local unmatched nodes of color j have been visited, the processor sends the match requests to the appropriate processors. That is, the matching request for (v, u) is sent to the processor that has the adjacency list of node u. The processors receiving matching requests process these as follows. If a single matching request is received for a node u, then the request is accepted immediately; otherwise, the heavy-edge heuristic is used to reject all requests but one. The accept/reject decisions are then sent to the requesting processors. At this point it is also decided in which processors collapsed nodes (i.e., nodes in the graph G^{i+1}) should be stored.

After the matching M^i is computed, each processor knows which of its nodes (and associated adjacency lists) to send to or to receive from other processors. The coarsening phase ends when a graph with $O(p)$ nodes has been obtained.

The partitioning of the coarsest graph is done in parallel as follows. First, a copy of the coarsest graph is created on each processor. Then a recursive bisection method is used. However, each processor follows only a single root-to-leaf path in the recursive bisection tree. At the end of this phase, the nodes stored in a processor correspond to one part in the p-way partition.

As usual the uncoarsening phase is a reversal of the coarsening phase, except the local improvement phase applied at each step. We describe how the initial partition of the graph $G^i = (N^i, E^i)$ is improved using a parallel version of the greedy refinement method (see p. 12). We assume that the nodes in N^i (with their adjacency lists) are randomly distributed over the processors, and that the nodes are colored using c_i colors. Recall that, in a single iteration of the sequential greedy refinement each node is visited once, and node movements that satisfy certain conditions on cut-size reduction and weight balance are carried out immediately. When a node is moved, the g-values and part weights influenced by the move are updated.

In the parallel formulation of the greedy refinement method, each iteration is divided into c_i phases, and during the j-th phase only nodes with color j are considered for movement. The nodes satisfying the above conditions are moved as a group. The g-values and part weights influenced by the moves are updated only at the end of each phase, before the next color is considered.

Karypis and Kumar [KK96] also give a parallel algorithm for graph coloring. This algorithm uses a simplified version of Luby's parallel algorithm for finding maximal independent sets [Lub86]. The coloring algorithm consists of several iterations. The input to each iteration consists of a subgraph of the initial graph. In this subgraph, an independent set of nodes is selected as follows. A random number is associated with each node, and if the number of a node is smaller than the numbers of its neighbors, it is selected. The selected nodes are all assigned the same color. (This color is only used in one iteration.) The input to the next iteration is obtained by removing all selected nodes from the graph.

The above graph partitioning algorithm was implemented on a 128-processor Cray T3D using the SHMEM library for communication. The experimental evaluation shows that, compared with the sequential algorithm [KK95c], the partition quality of the parallel algorithm is almost as good, and that the speedup varies from 14 to 35 depending on properties of the graph. For most of the graphs tested it took less than one second to compute an 128-way partition on 128 processors.

In [KK97], Karypis and Kumar propose a parallel multilevel p-way partitioning algorithm specifically adapted for message-passing libraries and architectures with high message startup. They observe that the algorithm in [KK96] is slow when implemented using the MPI library. This is due to high message startup time. To remedy this they introduce new algorithms for maximal matching and local improvement that require fewer communication steps.

The new matching algorithm consists of several iterations during which each processor tries to match its unmatched nodes using the heavy edge heuristic. More specifically, suppose that during the j-th iteration a processor wants to match its local node u with the node v. If v is also local to the processor, then the matching is granted right away. Otherwise the processor issues a matching request (to the processor storing v) only if

1. j is odd and $i(u) < i(v)$, or
2. j is even and $i(u) > i(v)$,

where $i(v)$ denotes a unique index associated with v. Then, each processor processes the requests, i.e., rejects all requests but one in case of conflicts. Karypis and Kumar report that satisfactory matchings are found within four iterations.

The new local improvement method is also based on the greedy refinement method (see p. 12). Each iteration is divided into two phases. In the first phase, nodes are moved from lower- to higher-indexed parts, and during the second phase nodes are moved in the opposite direction.

4 Conclusions

Researchers have developed sequential algorithms for static graph partitioning for a relatively long time. The multilevel-KL method seems to be the best general-purpose method developed so far. No other methods produce significantly better partitions at the same cost. The multilevel-RSB method may produce partitions of the same quality but appear to be much slower than multilevel-KL. It is possible that SA-like methods can produce partitions of higher quality than multilevel-KL, but SA-like methods seem to be very slow. Purely geometric methods may be faster than multilevel-KL methods, but are less general (since they are not coordinate-free) and generate lower-quality partitions. If coordinates are available, the inertial-KL algorithm may be fast and still give reasonably good cut sizes.

Karypis and Kumar [KK95c] tested their multilevel p-way algorithm on more than twenty graphs, the largest of which had 448000 nodes and 3310000 edges. For none of these graphs it took more than 40 seconds to compute a 256-way partition. So, do we need to do more research on sequential graph partitioning algorithms?

If we would like to partition even larger graphs, we may either run out of memory or find that our algorithms spend most of their time on external-memory accesses. In both cases an alternative is to partition the graph in parallel. However, the second problem may also be solved by algorithms designed to do as few external-memory accesses as possible. Recently, there has been an increased interest in sequential algorithms for extremely large data sets, see [CGG+95, KS96].

If the partitions produced by multilevel-KL methods were known to be close to optimal for most graphs, then there would be little need to develop other approaches. However, usually it is hard to know how far away from optimal the generated partitions are. Further analysis and evaluations of the partition quality obtained by, e.g., multilevel-KL methods would be interesting.

Most parallel algorithms for static graph partitioning are essentially parallelizations of well-known sequential algorithms. Although the parallel multilevel-KL algorithms [KK95d, KK96, KK97] perform coarsening, partitioning and uncoarsening slightly differently than their sequential counterparts, they seem to produce partitions of the same quality. Parallel multilevel-KL algorithms have been found to be much faster than parallel multilevel-RSB (see [KK96]), but as far as we know parallel multilevel-KL has not been compared with parallel versions of geometric methods such as inertial or inertial-KL. It is possible that such methods require less time and space than multilevel-KL methods while still producing partitions with acceptable cut sizes.

The parallel multilevel-KL methods are based on algorithms for solving graph problems such as finding maximal matchings and inde-

pendent sets, graph contraction, etc. These problems are of independent interest, and further research on practical parallel algorithms for these problems can be valuable.

The graph partitioning problem is only an approximation of the problem that we need to solve to parallelize specific applications efficiently. Vanderstraeten, Keunings and Fahrat [VKF95] point out that cut size may not always be the most important measure of partition quality. Thus, the multilevel-KL algorithm may not be the most suitable partitioner for all applications. They propose that evaluation of partitioning algorithms should focus on reductions in parallel application's running time instead of cut sizes. To simplify such evaluations researchers should select a set of benchmark applications.

This situation is even more pronounced in dynamic graph partitioning. For example, for dynamic finite element methods there seems to be several possible strategies ranging from doing nothing, doing only local balancing, doing both local balancing and cut size improvement, and partitioning from scratch. Which of these strategies are best (and which algorithms should be used to implement them) depend not only on the application, but may unfortunately also depend on which parallel computer is being used.

The repartitioning algorithms presented in Section 3.1 are fairly obvious approaches to repartitioning, and they could be used as good starting points for programmers developing software for specific applications. As for further research on general-purpose repartitioning algorithms, it would be good either if a set of benchmark applications could be selected or the problem could be properly formalized. The latter alternative should also include selection of an appropriate model of computation, such as, e.g., the Bulk Synchronous Parallel (BSP) model [Val93]

References

[AfAGR96] I. Al-furaih, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. In *Proc. International Conference on Supercomputing (ICS'96)*, 1996.

[AK95] C.J. Alpert and A.B. Kahng. Recent directions in netlist partitioning: A survey. *Integration: The VLSI Journal*, (19):1–81, 1995.

[Bar95] S.T. Barnard. Pmrsb: Parallel multilevel recursive spectral bisection. In *Supercomputing 1995*, 1995.

[BB87] M.J. Berger and S.H. Bokhari. A partitioning strategy for non-uniform problems across multiprocessors. *IEEE Transactions on Computers*, C-36:570–580, 1987.

[BCN93] S.H. Bokhari, T.W. Crockett, and D.M. Nicol. Parametric binary dissection. Technical Report 93-39, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1993.

[Ben75] J.L. Bentley. Multidimensional binary search trees used associative searching. *Communications of the ACM*, 18:509–517, 1975.

[BJ93] 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.

[BM96] T.N. Bui and B.R. Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7):841–855, 1996.

[BS93] S.T. Barnard and H.D. 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.

[BSSV94] P. Buch, J. Sanghavi, and A. Sangiovanni-Vincentelli. A parallel graph partitioner on a distributed memory multiprocessor. In *Proc. Frontiers'95. The Fifth Symp. on the Frontiers of Massively Parallel Computation*, pages 360–6, 1994.

[CGG+95] Yi Jen Chiang, M.T. Goodrich, E.F. Grovel, R. Tamassia, D.E. Vengroff, and J.S. Vitter. External-memory graph algorithms. In *Proc. 6th Annual ACM-SIAM Symp. Discrete Algorithms*, pages 139–49, 1995.

[CJL94] P. Ciarlet Jr and F. Lamour. Recursive partitioning methods and greedy partitioning methods: a comparison on finite element graphs. Technical Report CAM 94-9, UCLA, 1994.

[CYL95] Y-C. Chung, Y-J. Yeh, and J-S. Liu. A parallel dynamic load-balancing algorithm for solution-adaptive finite element meshes on 2D tori. *Concurrency: Practice and Experience*, 7(7):615–631, 1995.

[DF96] H.Q. Ding and R.D. Ferraro. An element-based concurrent partitioner for unstructured finite element meshes. In *Proc. of the 10th International Parallel Processing Symposium*, pages 601–605, 1996.

[DMP94] R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. Technical Report TR-RF-94-008, Dept. of Comp. Science, University of Paderborn, 1994.

[DPH95] P. Diniz, S. Plimpton, and R. Hendrickson, B. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proc. 7th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 615–620, 1995.

[Dut93] S. Dutt. New faster Kernighan-Lin-type graph-partitioning algorithms. In *Proc. IEEE Intl. Conf. Computer-Aided Design*, pages 370–377, 1993.

[Far88] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[FL93] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Internat. J. Numer. Meth. Engrg.*, 36(5):745–764, 1993.

[FM82] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *19th IEEE Design Automation Conference*, pages 175–181, 1982.

[GJS76] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[Glo89] F. Glover. Tabu search – part I. *ORSA J. Comput.*, 1:190–206, 1989.

[Glo90] F. Glover. Tabu search – part II. *ORSA J. Comput.*, 2:4–32, 1990.

[GMT95] J.R. Gilbert, G.L. Miller, and S-H. Teng. Geometric mesh partitioning: Implementations and experiments. In *Proc. International Parallel Processing Symposium*, pages 418–427, 1995.

[Gol89] D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, 1989.

[Gup97] A. Gupta. Fast and effective algorithms for graph partitioning and sparse-matrix ordering. *IBM J. Res. Develop.*, 41(1):171–183, 1997.

[GZ87] J.R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International J. of Parallel Programming*, 16(1):427–449, 1987.

[HB95] Y.F. Hu and R.J. Blake. An optimal dynamic load balancing algorithm. Technical Report DL-P-95-011, Daresbury Laboratory, Warrington, UK, 1995.

[HL95a] B. Hendrickson and R. Leland. The Chaco user's guide. Version 2.0. Technical Report SAND95-2344, Sandia National Laboratories, 1995.

[HL95b] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16(2):452–469, 1995.

[HL95c] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing'95*, 1995.

[JAMS89] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[JMJH93] Z. Johan, K.K. Mathur, S.L. Johnsson, and T.J.R. Hughes. An efficient communication strategy for finite element methods on the Connection Machine CM-5 System. Technical Report TR-11-93, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard Univ., 1993.

[JMJH94] Z. Johan, K.K. Mathur, S.L. Johnsson, and T.J.R. Hughes. Parallel implementation of recursive spectral bisection on the Connection Machine CM-5 System. Technical Report TR-07-94, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard Univ., 1994.

[JP94] M.T. Jones and P.E. Plassman. Computational results for parallel unstructured mesh computations. Technical Report UT-CS-94-248, Computer Science Department, Univ. Tennesee, 1994.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[KK95a] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. Technical Report 95-037, University of Minnesota, Department of Computer Science, 1995.

[KK95b] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, University of Minnesota, Department of Computer Science, 1995.

[KK95c] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report 95-064, University of Minnesota, Department of Computer Science, 1995.

[KK95d] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partioning and sparse matrix ordering. Technical Report 95-036, University of Minnesota, Department of Computer Science, 1995.

[KK96] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report 96-036, University of Minnesota, Department of Computer Science, 1996.

[KK97] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[KL70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical J.*, 49:291–307, 1970.

[KRC97] S.E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. Technical Report DIKU-TR-97/9, Dept. of Computer Science, Univ. Copenhagen, 1997.

[KS96] V Kumar and E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE Symp. Parallel and Distributed Processing*, pages 169–76, 1996.

[Lan50] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Stand.*, 45:255–282, 1950.

[Las91] G. Laszewski. Intelligent structural operators for the k-way graph partitioning problem. In *Proc. Fourth Int. Conf. Genetic Algorithms*, pages 45–52, 1991.

[LH94] R. Leland and B. Hendrickson. An empirical study of static load balancing algorithms. In *Proc. Scalable High-Performance Comput. Conf.*, pages 682–685, 1994.

[Lub86] M. Luby. A simple parallel algorithm for the maximal set problem. *SIAM J. on Scientific Computation*, 15(4):1036–1053, 1986.

[MTTV93] G.L. Miller, S-H. Teng, W. Thurston, and S.A. Vavasis. Automatic mesh partitioning. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *The IMA Volumes in Mathematics and its Applications*, pages 57–84. Springer Verlag, 1993.

[Nak97] I. Nakhimovski. Bucket-based modification of the parallel recursive coordinate bisection algorithm. *Linköping Electronic Articles in Computer and Information Science*, 2(15), 1997. See http://www.ep.liu.se/ea/cis/1997/015/.

[OdSF94] C. Özturan, H.L. deCougny, M.S. Shephard, and J.E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory computers. *Computer Methods in Applied Mechanics and Engineering*, 119:123–137, 1994.

[OR94] C-W. Ou and S. Ranka. Parallel remapping algorithms for adaptive problems. Technical Report CRPC-TR94506, Center for Research on Parallel Computation, Rice University, 1994.

[OR97] C-W. Ou and S. Ranka. Parallel incremental graph partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 8(8):884–96, 1997.

[ORF96] C-W. Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular problems. *Journal of Supercomputing*, 10(2):119–40, 1996.

[PSL90] A. Pothen, H.D. Simon, and K.P. Liu. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[RPG96] E. Rolland, H. Pirkul, and F. Glover. Tabu search for graph partitioning. *Ann. Oper. Res.*, 63:209–232, 1996.

[SF93] H.D. Simon and C. Farhat. TOP/DOMDEC; a software for mesh partitioning and parallel processing. Technical Report RNR-93-011, NASA, 1993.

[Sim91] H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2-3):135–148, 1991.

[SKK97a] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97-013, University of Minnesota, Department of Computer Science, 1997.

[SKK97b] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion schemes for repartitioning of adaptive meshes. Technical Report 97-014, University of Minnesota, Department of Computer Science, 1997.

[SR90] Y. Saab and V. Rao. Stochastic evolution: A fast effective heuristic for some genetic layout problems. In *Proc. 27th ACM/IEEE Design Automation Conf.*, pages 26–31, 1990.

[SW91] J.E. Savage and M.G. Wloka. Parallelism in graph-partitioning. *Journal of Parallel and Distributed Computing*, 13:257–272, 1991.

[SW93] J.E. Savage and M.G. Wloka. Mob – a parallel heuristic for graph-embedding. Technical Report CS-93-01, Brown Univ., Dep. Computer Science, 1993.

[Val93] L.G. Valiant. Why BSP computers? (bulk-synchronous parallel computers). In *Proc. 7th International Parallel Processing Symp.*, pages 2–5, 1993.

[VKF95] D. Vanderstraeten, R. Keunings, and C. Farhat. Beyond conventional mesh partitioning algorithms and the minimum edge cut criterion: Impact on realistic applications. In *Proc. 7th SIAM Conf. Parallel Processing for Scientific Computing*, pages 611–614, 1995.

[WCE97a] C. Walshaw, M. Cross, and M.G. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In *Proc. Parallel & Distributed Computing for Computational Mechanics*, 1997.

[WCE97b] C. Walshaw, M. Cross, and M.G. Everett. Parallel dynamic graph-partitioning for unstructured meshes. Technical Report 97/IM/20, University of Greenwich, Centre for Numerical Modelling and Process Analysis, 1997.

[Wil91] R.D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481, 1991.