

# Parallel Multilevel $k$ -Way Partitioning Scheme for Irregular Graphs\*

George Karypis<sup>†</sup>  
Vipin Kumar<sup>†</sup>

**Abstract.** In this paper we present a parallel formulation of a multilevel  $k$ -way graph partitioning algorithm. A key feature of this parallel formulation is that it is able to achieve a high degree of concurrency while maintaining the high quality of the partitions produced by the serial multilevel  $k$ -way partitioning algorithm. In particular, the time taken by our parallel graph partitioning algorithm is only slightly longer than the time taken for rearrangement of the graph among processors according to the new partition. Experiments with a variety of finite element graphs show that our parallel formulation produces high-quality partitionings in a short amount of time. For example, a 128-way partitioning of graphs with one million vertices can be computed in a little over two seconds on a 128-processor Cray T3D. Furthermore, the quality of the partitions produced is comparable (edge-cuts within 5%) to those produced by the serial multilevel  $k$ -way algorithm. Thus our parallel algorithm makes it feasible to perform frequent repartitioning of graphs in dynamic computations without compromising the partitioning quality.

**Key words.** parallel graph partitioning, multilevel partitioning methods, spectral partitioning methods, Kernighan–Lin heuristic, parallel sparse matrix algorithms

**AMS subject classifications.** 68R10, 65Y05, 65F10, 65F50

**PII.** S0036144598334138

**1. Introduction.** Graph partitioning is an important problem that has extensive applications in many areas, including scientific computing, very large scale integration (VLSI) design, geographical information systems, operations research, and task scheduling. The problem is to partition the vertices of a graph in  $p$  roughly equal partitions, such that the number of edges connecting vertices in different partitions is minimized. For example, the solution of a sparse system of linear equations  $Ax = b$  via iterative methods on a parallel computer gives rise to a graph partitioning problem. A key step in each iteration of these methods is the multiplication of a sparse matrix and a (dense) vector. A good partitioning of the graph corresponding to matrix  $A$

---

\*Received by the editors February 11, 1998; accepted for publication (in revised form) October 22, 1998; published electronically April 23, 1999. This work was supported by NSF CCR-9423082, by Army Research Office contract DA/DAAH04-95-1-0538, and by the Army High Performance Computing Research Center (AHPARC) under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement DAAH04-95-2-0003/contract DAAH04-95-C-0008. The content of this article does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by AHPARC, Minnesota Supercomputer Institute, Cray Research Inc., and the Pittsburgh Supercomputing Center. Related papers are available at <http://www.cs.umn.edu/~karypis>.

<http://www.siam.org/journals/sirev/41-2/33413.html>

<sup>†</sup>Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 (karypis@cs.umn.edu, kumar@cs.umn.edu).

can significantly reduce the amount of communication in parallel sparse matrix-vector multiplication [22].

The graph partitioning problem is NP-complete. However, many algorithms have been developed that find a reasonably good partition. Recently, a number of researchers have investigated a class of algorithms in which the original graph is successively coarsened down until it has only a small number of vertices, a partitioning of this coarsened graph is computed, and then this initial partitioning is successively refined using a Kernighan–Lin (KL) type heuristic as it is being projected back to the original graph. This multilevel paradigm was studied independently by Bui and Jones [4] in the context of computing fill-reducing matrix reordering, by Hendrickson and Leland [13] in the context of finite element grid partitioning, and by Hauck and Borriello [11] (called optimized KLFM) and Cong and Smith [5] for hypergraph partitioning. Karypis and Kumar studied this paradigm extensively in [17] and [20] for graph partitioning and presented new graph coarsening schemes that made the multilevel paradigm more robust. Multilevel schemes [13, 20, 18] are relatively fast and provide excellent partitions for a wide variety of graphs. In particular, these schemes provide significantly better partitions than those provided by spectral [24] and geometric [7] partitioning techniques, and are generally at least an order of magnitude faster than even the state-of-the-art implementation of spectral techniques [3]. Karypis and Kumar [18] also developed a multilevel  $k$ -way partitioning scheme in which a  $k$ -way partitioning of the coarsened graph is computed and refined using a variation of the KL refinement scheme. Multilevel  $k$ -way partitioning techniques are generally faster and provide better quality solutions than multilevel recursive bisection schemes [18].

Even though the multilevel partitioning algorithms produce high-quality partitions in a very small amount of time, the ability to perform partitioning in parallel is important for many reasons. The amount of memory on serial computers is not sufficient 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. In many applications, the graph is already distributed among processors, but needs to be repartitioned due to the dynamic nature of the underlying computation. For example, in the context of large-scale finite element simulations, adaptive mesh computations dynamically adjust the discretization of the physical domain. Such dynamic adjustments to the mesh lead to load imbalances and thus require repartitioning of the graph for efficient parallel computation [6]. In some problems, the computational effort in each grid cell changes over time [7]. For example, in many codes that advect particles through a grid, large temporal and spatial variations in particle density can introduce substantial load imbalance. Dynamic repartitioning of the corresponding vertex-weighted graph is crucial for balancing the computation among processors. Frequent repartitioning of meshes is also needed in some parallel mesh generation algorithms. In all such computations, if the mesh partitioning step is not performed in parallel, then the cost of moving the mesh to a single processor for partitioning can be very high. Graph partitioning is also used to compute fill-reducing ordering in the parallel formulation of direct solvers based upon sparse Cholesky factorization. With the recent development of highly parallel formulations of sparse Cholesky factorization algorithms [10, 16, 26], numeric factorization on parallel computers can take much less time than the step for computing a fill-reducing ordering on a serial computer, making that the new bottleneck. For example, on a 1024-processor Cray T3D, some matrices can be factored in less than

two seconds using our parallel sparse Cholesky factorization algorithm [10], but serial graph partitioning (needed for computing a fill-reducing ordering) takes an order of magnitude more time.

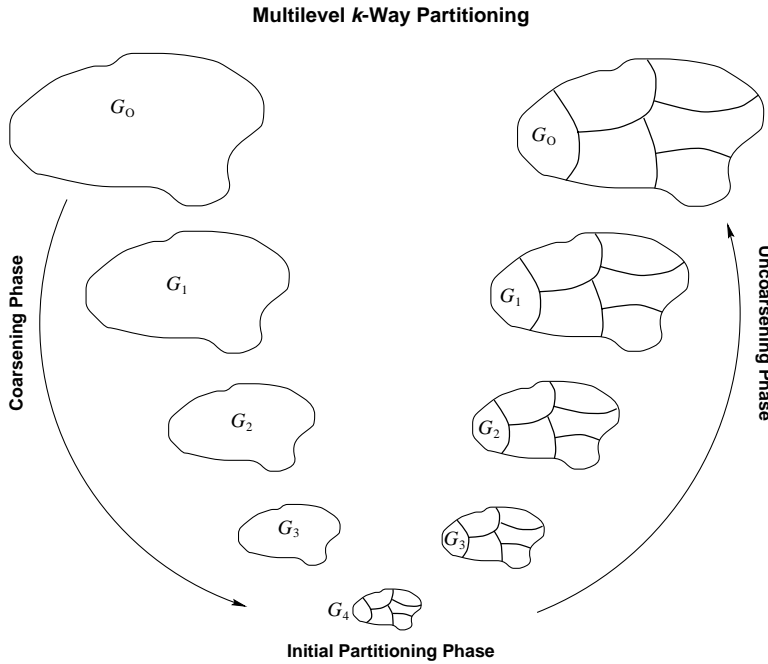
The problem of partitioning graphs on parallel computers has received a lot of attention [12, 25, 7, 14, 2, 1, 19, 29] due to its extensive applications in many areas. However, most of this work has been concentrated on algorithms based on geometric graph partitioning [12, 7] or spectral bisection [2, 1, 14]. Development of parallel formulations of multilevel graph partitioning schemes is quite challenging. Coarsening requires that nodes connected via edges be merged together. Since the graph is distributed across the processors, parallel coarsening schemes can require a lot of communication [25, 1, 19]. The KL refinement heuristic and its variant, which are used during the uncoarsening phase, appear to be serial in nature [9], and previous attempts to parallelize them have had mixed success [9, 7, 19]. In particular, parallel formulations of the refinement often have to trade quality for the available concurrency in this phase. Parallel formulation of the multilevel  $k$ -way partitioning scheme is even harder, as the refinement of the  $k$ -way partitioning appears to require global interactions.

In this paper we discuss problems encountered in parallelizing different phases of multilevel  $k$ -way partitioning schemes, and present a parallel formulation for the multilevel  $k$ -way partitioning algorithm [18]. A key feature of our parallel formulation is that it is able to achieve a high degree of concurrency while maintaining the high quality of the partitions produced by the serial multilevel partitioning algorithm. Parallel formulation of the coarsening phase is generally applicable to any multilevel graph partitioning algorithm that coarsens the graph, and the parallel formulation of the  $k$ -way partitioning refinement algorithm can also be used in conjunction with any other parallel graph partitioning algorithm that requires refinement (e.g., [7, 27]) of a  $k$ -way partitioning. We present a theoretical analysis of the scalability of our scheme as well as experimental evaluation of a large number of graphs from finite element methods and transportation domains.

The remainder of the paper is organized as follows. Section 2 briefly describes the serial multilevel  $k$ -way partitioning algorithm. Section 4 details our parallel formulation of the multilevel  $k$ -way partitioning algorithm. Section 5 provides a theoretical performance and scalability analysis. Section 6 presents an experimental evaluation of the parallel algorithm and compares its performance to that of the serial algorithm.

**2. Multilevel  $k$ -way Graph Partitioning.** Consider a weighted graph  $G = (V, E)$ , with weights on both vertices and edges. The  $k$ -way graph partitioning problem is to partition  $V$  into  $k$  subsets such that the sum of the weight of the vertices in each subset is roughly the same and the number of edges of  $E$  whose incident vertices belong to different subsets is minimized. A  $k$ -way partitioning of  $V$  is commonly represented by a partition vector  $P$  of length  $n$ , such that for every vertex  $v \in V$ ,  $P[v]$  is an integer between 1 and  $k$ , indicating the partition to which vertex  $v$  belongs. Given a partitioning  $P$ , the number of edges whose incident vertices belong to different subsets is called the *edge-cut* of the partitioning.

The basic structure of a multilevel algorithm is illustrated in Figure 2.1. The graph  $G = (V, E)$  is first coarsened down to a small number of vertices, a  $k$ -way partitioning of this much smaller graph is computed (using multilevel recursive bisection [20]), and then this partitioning is projected back towards the original graph (finer graph) by periodically refining the partitioning. Since the finer graph has more degrees of freedom, such refinements improve the quality of the partitions.



**Fig. 2.1** *The various phases of the multilevel  $k$ -way partitioning algorithm. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a  $k$ -way partitioning of the smaller graph is computed; and during the uncoarsening phase, the partitioning is successively refined as it is projected to the larger graphs.*

In the rest of this section we briefly describe the various phases of the multilevel  $k$ -way partitioning algorithm. The reader should refer to [18] for further details.

**2.1. Coarsening Phase.** The coarsening phase of the multilevel  $k$ -way partitioning is identical to that used in the multilevel recursive bisection schemes [13, 20]. During the coarsening phase, a sequence of smaller graphs,  $G_i = (V_i, E_i)$ , is constructed from the original graph,  $G_0 = (V_0, E_0)$ , such that  $|V_i| > |V_{i+1}|$ . Graph  $G_{i+1}$  is constructed from  $G_i$  by finding a maximal matching  $M_i \subseteq E_i$  of  $G_i$  and collapsing together the vertices that are incident on each edge of the matching. Vertices that are not incident on any edge of the matching are simply copied over to  $G_{i+1}$ .

When vertices  $v, u \in V_i$  are collapsed to form vertex  $w \in V_{i+1}$ , the weight of vertex  $w$  is set equal to the sum of the weights of vertices  $v$  and  $u$ , and the edges incident on  $w$  are set equal to the union of the edges incident on  $v$  and  $u$  minus the edge  $(v, u)$ . For each pair of edges  $(x, v)$  and  $(x, u)$  (i.e.,  $x$  is adjacent to both  $v$  and  $u$ ) a single edge  $(x, w)$  is created whose weight is set equal to the sum of the weights of these two edges. Thus, during successive coarsening levels, the weight of both vertices and edges increases.

Maximal matchings can be computed in different ways [13, 20, 18]. The method used to compute the matching greatly affects both the quality of the partition and the time required during the uncoarsening phase. The matching scheme that we use is called *heavy-edge matching* (HEM) and computes a matching,  $M_i$ , such that the weight of the edges in  $M_i$  is high. The HEM is computed using a randomized algorithm as follows. The vertices are visited in a random order. However, instead

of randomly matching a vertex with one of its adjacent unmatched vertices, HEM matches it with the unmatched vertex that is connected with the heavier edge. As a result, the HEM scheme quickly reduces the sum of the weights of the edges in the coarser graphs. The coarsening phase ends when the coarsest graph,  $G_m$ , has a small number of vertices.

**2.2. Partitioning Phase.** The second phase of a multilevel  $k$ -way partitioning algorithm is to compute a  $k$ -way partitioning of the coarse graph,  $G_m = (V_m, E_m)$ , such that each partition contains roughly  $|V_0|/k$  vertex weights of the original graph. Since, during coarsening, the weights of the vertices and edges of the coarser graph were set to reflect the weights of the vertices and edges of the finer graph,  $G_m$  contains sufficient information to enforce the balanced partitioning and the minimum edge-cut requirements intelligently. In our partitioning algorithm, a  $k$ -way partitioning of  $G_m$  is computed using our multilevel recursive bisection algorithm [20]. Our experiments have shown that it produces good initial partitions in a relatively short amount of time.

**2.3. Uncoarsening Phase.** During the uncoarsening phase, the partitioning of the coarser graph,  $G_m$ , is projected back to the original graph by going through the graphs  $G_{m-1}, G_{m-2}, \dots, G_1$ . Since each vertex  $u \in V_{i+1}$  contains a distinct subset  $U$  of vertices of  $V_i$ , the projection of the partitioning from  $G_{i+1}$  to  $G_i$  is constructed by simply assigning the vertices in  $U$  to the same partition in  $G_i$  to which vertex  $u$  belongs in  $G_{i+1}$ . After a partitioning is projected from  $G_{i+1}$  to  $G_i$ , it is refined using local refinement heuristics.

Our multilevel  $k$ -way partitioning algorithm uses a variation of the KL [21] algorithm to provide  $k$ -way partitioning refinement. This algorithm is called *greedy refinement* (GR). Its complexity is largely independent of the number of partitions being refined. Key to the GR algorithm is the concept of *gain*, which is defined as the decrease in the edge-cut achieved by moving a vertex from one partition to another. The GR algorithm consists of a number of iterations, and in each iteration all the vertices are checked in a random order to see if they can be moved. Let  $v$  be such a vertex. If  $v$  is a boundary vertex, then  $v$  is moved to the partition that leads to the largest reduction in the edge-cut (i.e., the partition with the largest positive gain), subject to partition balance constraint. The balance constraint ensures that all partitions have roughly the same weight. Specifically, if the balance constraint parameter is  $b$ , then the weight of the heaviest partition divided by the average weight of the partitions should be less than  $b$ .

**3. Challenges and Related Work.** Out of the three phases of the multilevel  $k$ -way partitioning algorithm described in section 2, the coarsening and the uncoarsening phases require the bulk of the computation (over 95%). Hence, it is critical for any efficient parallel formulation of the multilevel  $k$ -way partitioning algorithm to successfully parallelize these two phases. In the following, we review the difficulties encountered in parallelizing these phases, and previous related works.

*Coarsening.* Recall that, during the coarsening phase (section 2.1), a matching of the edges is computed, and this is used to contract the graph. One possible way of computing the matching in parallel is to have each processor only compute matchings between the vertices that it stores locally, and to use these local matchings to contract the graph. Since each pair of matched vertices resides on the same processor, this approach requires no communication during the contraction step. This approach works well as long as each processor stores relatively well connected portions of the

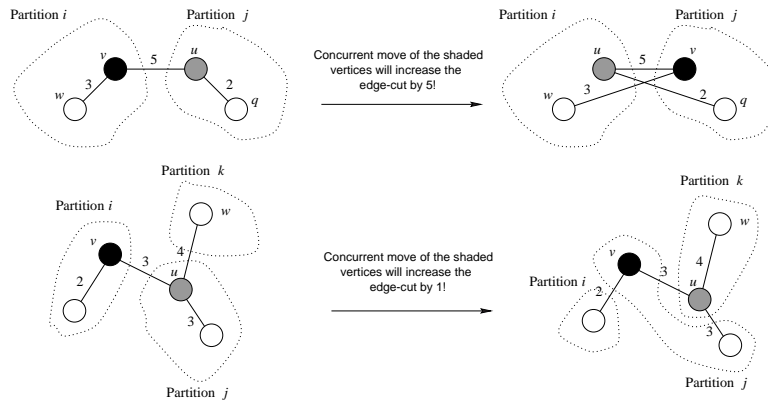
entire graph. In particular, if the graph is distributed among the processors in a partitioned fashion, then this approach works extremely well. This is not a realistic assumption in many cases, since a good partitioning of the graph is what we are trying to compute by the multilevel  $k$ -way partitioner. Nevertheless, this approach of *local matchings* can work reasonably well when the number of processors used is small relative to the size of the graph and the average degree of the graph is relatively high. The reason is that even a random partitioning of a graph among a small number of processors can leave many connected components at each processor. This approach can also work well for computing a new partitioning of meshes arising in adaptive computations, especially if the adapted graph is a small perturbation of the original well-partitioned graph.

Another scheme investigated in [19] uses a two-dimensional distribution of the adjacency matrix of the graph. This requires the vertices of the graph to be partitioned among only  $\sqrt{p}$  processors. Hence, this graph distribution allows a moderate amount of coarsening even by using purely local matchings. This local matching produces sufficient coarsening as long as the average degree of the coarse graphs is sufficiently large (proportional to the square root of the number of processors). However, if the degree of the graphs is small (as is the case for finite element meshes and their duals), then this local matching cannot sufficiently reduce the size of the graph before folding is required, resulting in poor speedup.

An alternate approach is to allow vertices belonging to different processors to be matched together. Compared to local matching schemes, this type of matching provides matchings of better quality, and its ability to contract the graph does not depend on the number of processors or the existence of a good prepartitioning. However, this *global matching* requires a high degree of fine-grain interprocessor communication that can be expensive, especially on message passing architectures. Furthermore, this global matching significantly complicates the parallel formulation because it requires a distributed matching algorithm. For example, if vertices  $v$  and  $u$  are located in two different processors,  $P_1$  and  $P_2$ , then on the  $P_1$  vertex,  $v$  might be matched to  $u$ , while on the  $P_2$  vertex,  $u$  may be matched to a different vertex  $w$ . Furthermore, another processor  $P_3$  may match its vertex  $z$  to vertex  $u$  as well. Any correct and usable distributed matching algorithm must resolve both of these conflicts efficiently. Note that since pairs of vertices that are contracted together can reside on different processors, a global communication is required when the contracted graph is constructed.

A scheme for global matching has been investigated by Barnard [1] in the context of a parallel formulation of a multilevel spectral algorithm. This algorithm uses a one-dimensional mapping of the graph to the processors and uses a parallel formulation of Luby's [23] algorithm to compute a maximal independent set of vertices to construct the next-level coarser graph. Another approach was investigated by Raghavan [25] in the context of the multilevel nested dissection algorithm. Raghavan's parallel algorithm uses one-dimensional partitioning of the graphs and constructs successively coarser graphs by computing matchings between different pairs of processors at each coarsening level. Note that this matching scheme pairs vertices located on different processors, but it does not make use of global maximal matchings across all processors in each coarsening step. This allows coarsening to be performed with only limited interaction among processors by trading the quality of the matching obtained.

*Refinement During Uncoarsening.* During the uncoarsening phase, the  $k$ -way partitioning is iteratively refined while it is projected to successively finer graphs. The serial algorithm scans the vertices in a random order and moves any vertices that lead



**Fig. 3.1** *Concurrent moves of vertices may increase the overall edge-cut even if each individual vertex move leads to a reduction.*

to a reduction in the edge-cut. Any parallel formulation of this algorithm will need to move a group of vertices at a time in order to speed up the refinement process. This group of vertices needs to be carefully selected so that every vertex in the group contributes to the reduction in the edge-cut. For example, it is possible that processor  $P_i$  would decide to move a set of vertices  $S_i$  to processor  $P_j$  to reduce the edge-cut because the vertices in  $S_i$  are connected to a set of vertices  $T$  that are located on processor  $P_j$ . But, in order for the edge-cut to improve by moving the vertices in  $S_i$ , the vertices in  $T$  must not move. However, while  $P_i$  selects  $S_i$ , processor  $P_j$  may decide to move some or all of the vertices in  $T$  to some other processor. Consequently, when both sets of vertices are moved by  $P_i$  and  $P_j$ , the edge-cut may not improve, and it may even get worse. This is illustrated in the two simple examples shown in Figure 3.1. In both examples, moving vertices  $v$  and  $u$  individually will lead to a reduction in the edge-cut; however, if both of these moves are performed, the overall edge-cut will increase.

The group selection algorithm must eliminate this type of unnecessary vertex movement. Note that this problem does not arise if the movement of vertices in the refinement is restricted such that whenever vertices are considered for movement from partition  $i$  to  $j$ , then vertices in  $j$  are not considered for movement to any other partition (including  $i$ ). One possible way of avoiding redundant moves along these lines is proposed by Diniz et al. in [7]. In this method, the refinement process is restricted to disjoint pairs of partitions. Consider, for example, the 10-way partitioning of a graph shown in Figure 3.2a and assume that the graph has been placed among the processors such that processor  $P_i$  stores the  $i$ th partition of the graph. Figure 3.2b shows the corresponding partition graph, that is, a graph in which there is an edge between partitions  $i$  and  $j$  if they are neighbors (i.e., share a common boundary). Note that a partitioning refinement algorithm needs to move vertices across boundaries for each one of these shared boundaries, that is, edges in the partition graph. The boundaries that can be refined concurrently are determined by a matching of the partition graph [7]. In the scheme used by Diniz et al. [7], the refinement is performed in 6 steps as shown in Figure 3.2c–Figure 3.2h. In each step, the common boundaries corresponding to the edges of the subgraph are refined. These boundaries are determined by computing a matching of the edges that have not yet been refined. Note

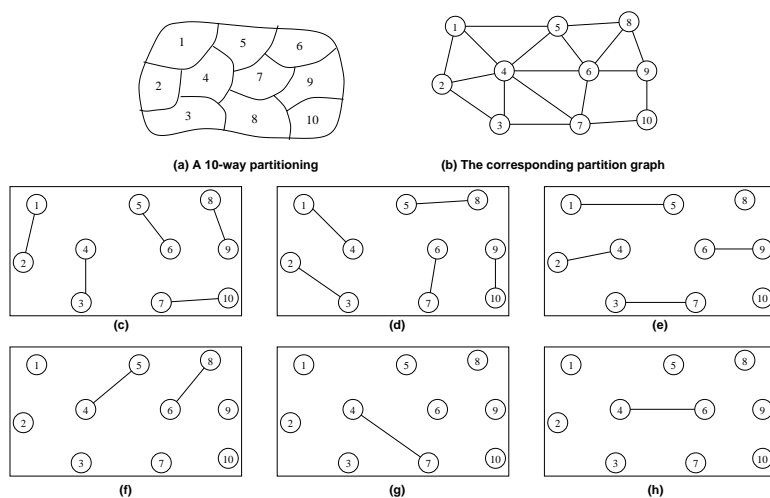


Fig. 3.2 Refinement restricted to disjoint pairs of partitions.

that in the last three steps, only a small fraction of the processors are actually busy performing refinement, which limits the concurrency available in the refinement step. A major drawback of this parallel refinement algorithm is that it restricts the type of vertex movement that can be performed in each step. Hence, it lacks the global view available in the serial refinement algorithm, in which each vertex is free to move to the partition that leads to the maximum reduction in the edge-cut. Experiments in [7] show that the quality of the partitions produced by the parallel inertial algorithm are up to 13% worse compared to the serial implementation of the inertial algorithm that uses sequential KL refinement.

**4. Parallel Formulation.** In this section, we present parallel formulations for all three phases of the multilevel  $k$ -way graph partitioning algorithm. Our parallel formulation computes a coloring of the graph at each coarsening level, which is used to eliminate conflicts in the computation of global matching in the coarsening phase and to eliminate unnecessary vertex movement during the  $k$ -way refinement performed in the uncoarsening phase. Specifically, during the coarsening phase, our algorithm computes a global matching incrementally by only matching nodes of the same color at a time. Similarly, the  $k$ -way refinement is also performed incrementally by only moving vertices of the same color at a time. Since the vertices of the same color form an independent set, this scheme ensures that every vertex movement will indeed lead to a reduction in the edge-cut. We also exploit the task-level parallelism of the initial graph partitioning algorithm to further reduce the already small run time of this phase.

This coloring-based method used for coarsening and refinement has the following drawbacks: (i) the computation of coloring requires a lot of global communication; (ii) since matching is done for one color at a time, a global synchronization is needed for each color. In our formulation, we try to limit both of these drawbacks by making some modifications to the above method.

Let  $p$  be the number of processors used to compute a  $p$ -way partitioning of the graph  $G = (V, E)$ .  $G$  is initially distributed among the processors using a



one-dimensional distribution, so that each processor receives  $n/p$  vertices and their adjacency lists. At the end of the algorithm, a partition number is assigned to each vertex of the graph. In the following sections we first describe the algorithm for computing a coloring of a graph on a distributed memory computer, and then describe our parallel formulations for the three phases of the multilevel  $k$ -way partitioning algorithm described in section 2.

**4.1. Computing a Coloring of a Graph.** A coloring of a graph  $G = (V, E)$  assigns colors to the vertices of  $G$  so that adjacent vertices have different colors. We like to find a coloring such that the number of distinct colors used is small. Our parallel graph coloring algorithm consists of a number of iterations. In each iteration, a maximal independent set of vertices  $I$  is selected using a variation of Luby's [23] algorithm. All vertices in this independent set are assigned the same color. Before the next iteration begins, the vertices in  $I$  are removed from the graph, and this smaller graph becomes the input graph for the next iteration. A maximal independent set  $I$  of a set of vertices  $S$  is computed in an incremental fashion using Luby's algorithm as follows. A random number is assigned to each vertex, and if a vertex has a random number that is smaller than all the random numbers of the adjacent vertices, it is then included in  $I$ . Now this process is repeated for the vertices in  $S$  that are neither in  $I$  nor adjacent to vertices in  $I$ , and  $I$  is augmented similarly. This incremental augmentation of  $I$  ends when no more vertices can be inserted in  $I$ . It is shown in [23] that one iteration of Luby's algorithm requires a total of  $O(\log |S|)$  such augmentation steps to find a maximal independent set of a set  $S$ .

Luby's algorithm can be implemented quite efficiently on a shared memory parallel computer, since for each vertex  $v$ , a processor can easily determine if the random value assigned to  $v$  is the smallest among all the random values assigned to the adjacent vertices. However, on a distributed memory parallel computer, for each vertex, random values associated with adjacent vertices that are not stored on the same processor need to be explicitly communicated. Furthermore, a faithful implementation of Luby's algorithm will also suffer from high synchronization overheads, as it requires a global synchronization step during each iteration. Jones and Plassmann [15] have developed an asynchronous variation of Luby's algorithm that is particularly suited for distributed memory parallel computers. In their algorithm, each vertex is assigned a single random number and, after a communication step, each vertex determines the number of its adjacent vertices that have smaller and greater random numbers. At this point each vertex gets into a loop waiting to receive the color values of its adjacent vertices that have smaller random numbers. Once all these colors have been received, the vertex selects a consistent color and sends it to all of its adjacent vertices with greater random numbers. The algorithm terminates when all vertices have been colored. Note that, besides the initial communication step to determine the number of smaller and greater adjacent vertices, this algorithm proceeds asynchronously.

Rather than using the algorithm by Jones and Plassmann to compute the coloring of a graph, we decided to use a direct implementation of the original algorithm by Luby, as it is easier to implement. In our implementation of Luby's algorithm, we perform only a single augmentation step to compute the independent set during each iteration. Hence, the independent set computed is not maximal. Even though this leads to an increase in the number of colors required to color the entire graph, it significantly reduces the overall run time. Furthermore, we do not color all nodes of the graph, but stop when a large fraction of the graph is colored. This is acceptable because it still allows most of the nodes at any level to participate in the coarsening and

refinement phase while significantly limiting the required number of synchronization steps.

In our implementation of Luby's algorithm, prior to performing the coloring in parallel, we perform a *communication setup* phase, in which appropriate data structures are created to facilitate this exchange of random numbers. In particular, we predetermine which vertices are located on a processor boundary (i.e., a vertex connected with vertices residing on different processors) and which are internal vertices (i.e., vertices that are connected only to vertices on the same processors). These data structures are used in all the phases of our parallel multilevel graph partitioning algorithm.

**4.2. Coarsening Phase.** Recall from section 2.1 that during the coarsening phase a sequence  $G_1, G_2, \dots, G_m$  of successively smaller graphs is constructed. Graph  $G_{i+1}$  is derived from  $G_i$  by finding a matching  $M_i$  of  $G_i$  and then collapsing the vertices incident on the edges of  $M_i$ . Since the matching  $M_i$  is an independent set of edges, we can use Luby's parallel algorithm on the line graph<sup>1</sup> of  $G_i$  to compute a global matching in parallel. However, computing a matching using this algorithm can be quite expensive since the line graph usually has significantly more vertices than  $G_i$ , and it is somewhat denser. For this reason, we use a matching algorithm based on the coloring of the graph. This coloring algorithm also happens to be essential for parallelizing the partitioning refinement phase.

Our parallel matching algorithm is based on an extension of the serial algorithm and utilizes graph coloring to structure the sequence of computations. Consider the graph  $G_i = (V_i, E_i)$  that has been colored using our parallel formulation of Luby's algorithm, and let *Match* be a variable associated with each vertex of the graph, which is initially set to  $-1$ . At the end of the computation, the variable *Match* for each vertex  $v$  stores the vertex to which  $v$  is matched. If  $v$  is not matched, then *Match* =  $v$ . To simplify the presentation, we first describe the algorithm assuming that the target parallel computer has a shared memory architecture, and later show how this algorithm is implemented on a distributed memory machine.

The matching  $M_i$  is constructed in an iterative fashion. During the  $c$ th iteration, vertices of color  $c$  that have not yet been matched (i.e., *Match* =  $-1$ ) select one of their unmatched neighbors using the heavy-edge heuristic, and modify the *Match* variable of the selected vertex by setting it to their vertex number. Let  $v$  be a vertex of color  $c$  and  $(v, u)$  be the edge that is selected by  $v$ . Since the color of  $u$  is not  $c$ , this vertex will not select a partner vertex at this iteration. However, there is a possibility that another vertex  $w$  of color  $c$  may select  $(w, u)$ . Since both vertices  $v$  and  $w$  perform their selections at the same time, there is no way of preventing this. This is handled as follows. After all vertices of color  $c$  select an unmatched neighbor, they synchronize. The vertices of color  $c$  that have just selected a neighbor read the *Match* variable of their selected vertex. If the value read is equal to their vertex number, then their matching was successful, and they set their *Match* variable equal to the selected vertex; otherwise the matching fails, and the vertex remains unmatched. Note that if more than one vertex (e.g.,  $v$  and  $w$ ) want to match with the same vertex (e.g.,  $u$ ), only one of the *writes* in the *Match* variable of the selected vertex will succeed, and this determines which matching survives. However, by using coloring we restrict which vertices select partner vertices during each iteration; thus, the number of such conflicts is significantly reduced. Also note that, even though a vertex of color  $c$  may

<sup>1</sup>The line graph  $G'$  of  $G$  is constructed by creating a vertex for each edge of  $G$  and connecting two vertices in  $G'$  if the corresponding edges in  $G$  are incident on a common vertex.

fail to have its matching accepted due to conflicts, this vertex can still be matched during a subsequent iteration corresponding to a different color.

The above algorithm is implemented quite easily on a distributed memory parallel computer as follows. The *writes* into the *Match* variables are gathered together and are sent to the corresponding processors in a single message. If a processor receives multiple *write* requests for the same vertex, the one that corresponds to the heavier edge is selected. Any ties are broken arbitrarily. Similarly, the *reads* from the *Match* variables are gathered by the processors that store the corresponding variables and they are sent in a single message to the requesting processors. Furthermore, during this *read* operation, the processors who own the *Match* variables also determine if they will be the ones storing the collapsed vertex in  $G_{i+1}$ . This is done by using a uniformly distributed random variable. The vertex is kept or given away with the same probability. Our experiments have shown that this simple heuristic leads to a very good load balance.

After a matching  $M_i$  is computed, each processor knows how many vertices (and the associated adjacency lists) it needs to send and how many it needs to receive. Each processor then sends and receives these subgraphs, and it forms the next-level coarser graph by merging the adjacency lists of the matched vertices. The coarsening process ends when the graph has  $O(p)$  vertices.

**4.3. Partitioning Phase.** During the partitioning phase, a  $p$ -way partitioning of the graph is computed using a recursive bisection algorithm. Since the coarsest graph has only  $O(p)$  vertices, this step can be performed serially without significantly affecting the performance of the entire algorithm. Nevertheless, in our algorithm we also parallelize this phase using a recursive decomposition. This is done as follows: the various pieces of the coarse graph are gathered to all the processors using an all-to-all broadcast operation [22]. At this point the processors perform recursive bisection using an algorithm that is based on nested dissection [8] and greedy partitioning refinement. However, each processor explores only a single path of the recursive bisection tree. At the end each processor stores the vertices that correspond to its partition of the  $p$ -way partitioning. Note that after the initial all-to-all broadcast operation, the algorithm proceeds without any further communication.

Note that the algorithm used for computing the initial partitioning of the graph in the parallel multilevel algorithm is different from the multilevel recursive bisection used in the serial algorithm. The multilevel algorithm produces significantly better initial partitions than nested dissection but it requires more time. Consequently, the initial partitioning step may become a bottleneck for a very large number of processors, particularly for smaller graphs. However, due to the  $k$ -way refinement performed in the uncoarsening phase, the final partitions are only slightly worse than those produced by the serial  $k$ -way algorithm (that uses the multilevel recursive bisection algorithm for computing initial partitions). Thus, the use of nested dissection for initial partitioning (in place of a more accurate multilevel recursive bisection scheme) trades a slight reduction in quality for better run time and scalability.

**4.4. Uncoarsening Phase.** In the uncoarsening phase, the partitioning is projected from the coarse graph to the next-level finer graph, and it is refined using the GR algorithm (section 2.3). Recall that during a single phase of the refinement in the serial algorithm, vertices are randomly traversed and moved to a partition that leads to greater decrease in the edge-cut subject to the balance constraint. After each such vertex movement, the external degrees of the adjacent vertices are updated.

In the parallel formulation of greedy refinement, we retain the spirit of the serial algorithm, but we change the order in which the vertices are traversed to determine whether they can be moved to different partitions. In particular, the single phase of the refinement algorithm is broken up into  $c$  subphases, where  $c$  is the number of colors of the graph to be refined. During the  $i$ th phase, all the vertices of color  $i$  are considered for movement, and the subset of these vertices that leads to a reduction in the edge-cut (or improves the balance without increasing the edge-cut) is moved. Since the vertices with the same color form an independent set, the total reduction in the edge-cut achieved by moving all vertices at the same time is equal to the sum of the edge-cut reductions achieved by moving these vertices one after the other. After performing this *group* movement, the external degrees of the vertices adjacent to this group are updated, and the next color is considered.

During the parallel refinement step, it appears natural to physically move the vertices as they change partitions. That is, each processor initially stores all the vertices of a single partition, and as vertices move between partitions during refinement, they can also move between the corresponding processors. However, in the context of multilevel graph partitioning such an approach requires significant communication. This is because, for each vertex  $v$  in the coarse graph  $G_i$  that we move, we need to send not only the adjacency list of  $v$  but also the adjacency lists of all the vertices collapsed in  $v$  for the higher level finer graphs  $G_{i-1}, G_{i-2}, \dots, G_0$ . In our parallel refinement algorithm we solve this problem as follows. Vertices do not move from processor to processor—only the partition number associated with each vertex changes. This also ensures that the computations performed during the refinement are reasonably load balanced provided that the vertices are initially distributed in a random order. In this case, during refinement, each processor will have some boundary vertices that need to be moved since each processor stores a roughly equal number of vertices from all  $p$  partitions. This also leads to a simpler implementation of the parallel refinement algorithm, since vertices (and their adjacency lists) do not have to be moved around. Of course, all the vertices are moved to their proper location at the end of the partitioning algorithm, using a single all-to-all personalized communication [22].

The balance constraint is maintained as follows. Initially, each processor knows the weights of all  $p$  partitions. During each refinement subphase, each processor enforces balance constraints based on these partition weights. For every vertex it decides to move, it locally updates these weights. At the end of each subphase, the global partition weights are recomputed, so that each processor knows the exact weights. Even though the balance constraints maintained by this scheme are less exact than those maintained by the serial algorithm, our experiments have shown that the hybrid of local and global partition balance constraints is able to produce well-balanced partitions.

Furthermore, the above parallel refinement algorithm is highly concurrent, as long as the number of colors is small compared to the total number of vertices in the graph. For three-dimensional finite element meshes with tetrahedral elements, the number of colors tends to be less than 20, and for the graphs corresponding to their duals, it tends to be less than 5. Since the serial and parallel refinement algorithms are similar in spirit, both exhibit similar partitioning refinement capabilities.

**5. Performance and Scalability Analysis.** The parallel formulation of the multilevel  $k$ -way partitioning algorithm described in section 4 consists of five different parallel algorithms, namely, coloring, matching, contraction, initial partitioning, and

refinement. Out of these five algorithms, four of them (coloring, matching, contraction, and refinement) have similar requirements in terms of computation and communication. The exact computational and communication requirements of these phases depend on the characteristics of the graph that is partitioned. In particular, the computational requirements depend on the average degree of the graph, whereas the communication requirement depends on the average degree, as well as the chromatic number of the graph. A graph with a higher chromatic number will require more iterations and global synchronizations during the coarsening and the uncoarsening steps, as the number of inner iterations that are performed in each such step is proportional to the number of colors of the graph. In the rest of this section, we will analyze the parallel complexity and scalability of our algorithm under the following two assumptions: (i) each vertex in the graph has a small bounded degree; and (ii) this property is also satisfied by the successive coarser graphs. These assumptions allow us to make the following observations and simplifications:

- i. The chromatic number of graphs at all coarsening levels is a small constant.
- ii. The size of the successively coarser graphs decreases by a factor of  $1 + \epsilon$ , where  $0 < \epsilon \leq 1$ .
- iii. We can ignore the number of edges from the analysis, as it is of the same order as the number of vertices.

It may seem that these assumptions limit the applicability of the analysis; however, they are true for all graphs that correspond to well-shaped finite element meshes or their duals, the class of problems that often requires parallel graph partitioning. We further assume that the graph is initially distributed randomly among processors. This is, in general, a worst-case assumption. If there is a locality in the distribution of the graph, then often it can be exploited to reduce the time of different phases.

The amount of computation performed by each one of these four algorithms is proportional to the size of the graph stored locally on each processor. Since the size of the successively coarser graphs decreases by a factor of  $1 + \epsilon$  for  $0 < \epsilon \leq 1$ , the computation performed for the original graph dominates the computation performed for the subsequent  $O(\log n)$  coarser graphs. Thus, the amount of overall computation performed is

$$(5.1) \quad T_{calc} = O\left(\frac{n}{p}\right).$$

The amount of communication performed by each one of these four algorithms depends on the number of interface vertices. For example, during coloring, each processor needs to know the random numbers associated with the vertices adjacent to the locally stored vertices. Similarly, during refinement, every time a vertex is moved the adjacent vertices need to be notified to update their external degrees. Each processor stores  $n/p$  vertices and  $nd/p$  edges, where  $d$  is the average degree of the graph. Thus, the number of interface vertices is at most  $O(n/p)$ . Since the vertices are initially distributed randomly, these interface vertices are equally distributed among the  $p$  processors. Hence, each processor needs to exchange data with  $O(n/p^2)$  vertices of each processor. Alternatively, each processor needs to send information for about  $O(n/p^2)$  locally stored vertices to each other processor. This can be accomplished by using the all-to-all personalized communication operation [22], whose complexity is  $O(n/p) + O(p)$ . The communication complexity over all  $O(\log n)$  coarsening levels is

$$(5.2) \quad T_{comm} = O\left(\frac{n}{p}\right) + O(p \log n),$$

since the size of the graph is successively halved.

Note that for (5.2) to be valid, the data to be communicated among processors must be roughly equally distributed. Since the graph is randomly distributed, this is a reasonable assumption (otherwise, a somewhat more expensive generalized all-to-all personalized communication [28] is needed). Furthermore, the successively coarser graphs also remain randomly distributed because, during matching, decisions regarding where the contracted vertex will reside are made randomly.

In addition to the communication of the interface vertices, both matching and refinement perform additional communication. During matching, a prefix-sum is performed to determine the numbering of the vertices in the coarser graph. The complexity of this operation over all  $O(\log n)$  coarsening levels is  $O(\log p \log n)$  [22]. During refinement, a reduction of a  $p$ -vector is performed to compute the weights of the partitions. Since the size of the vector is equal to  $p$ , each such reduction can be done in  $O(p)$  time [22]. Thus, the complexity over all  $O(\log n)$  coarsening levels is  $O(p \log n)$ . Finally, all four algorithms require global synchronizations, whose complexity is  $O(\log p \log n)$ . However, the complexity of the above communication overheads is subsumed by the complexity of sending information about interface vertices (cf. (5.2)).

During the initial partitioning phase, a graph of size  $O(p)$  is partitioned into  $p$  partitions using recursive bisection. As described in section 4.3, the graph is gathered on each processor using an all-to-all broadcast operation [22], whose complexity is  $O(p)$ . After that, each processor performs recursive bisection, but keeps only one of the two bisections. Thus, the computational complexity of the initial partitioning is  $O(p)$ .

Thus, from (5.1) and (5.2) we have that the parallel run time of the multilevel partitioning algorithm is

$$(5.3) \quad T_{par} = O\left(\frac{n}{p}\right) + O(p \log n).$$

Since the sequential complexity of the serial algorithm is  $O(n)$ , the isoefficiency function [22] of our algorithm is  $O(p^2 \log p)$ .

In most application domains where parallel graph partitioning is required, it is followed by a step that permutes the graph according to the computed partitioning. For this reason, it is instructive to compare the run time of our parallel  $k$ -way partitioning algorithm with the amount of time required to perform this permutation. If we assume that the graph is randomly distributed among the processors, this permutation is equivalent to an all-to-all personalized communication of the original graph. The run time of this communication operation is  $O(n/p) + O(p)$  [22]. Thus, the run time of our parallel graph partitioning algorithm is only slightly higher (by a factor of  $O(\log n)$  in the second term of (5.3)) than the amount of time required to actually permute the graph; hence, it imposes limited additional computational overhead on the underlying application.

**6. Experimental Results.** We evaluated the performance of our parallel multilevel  $k$ -way graph partitioning algorithm on a wide range of graphs arising in different application domains. The characteristics of these graphs are described in Table 6.1.

We implemented our parallel multilevel algorithm on a 128-processor Cray T3D parallel computer. Each processor on the T3D is a 150 MHz Dec Alpha (EV4). The processors are interconnected via a three-dimensional torus network that has a peak unidirectional bandwidth of 150 bytes per second, and a small latency. We used the SHMEM message passing library for communication. In our experimental

**Table 6.1** *Various graphs used in evaluating the parallel multilevel  $k$ -way graph partitioning algorithm.*

Graph name	No. of vertices	No. of edges	Description
144	144649	1074393	3D finite element mesh
598A	110971	741934	3D finite element mesh
AUTO	448695	3314611	3D finite element mesh
BRACK2	62631	366559	3D finite element mesh
COPTER2	55476	352238	3D finite element mesh
M14B	214765	1679018	3D finite element mesh
MAP1	267241	334931	Highway network
MDUAL	258569	513132	Dual of a 3D finite element mesh
MDUAL2	988605	1947069	Dual of a 3D finite element mesh
OCEAN	143437	409593	2D finite element mesh
ROTOR	99617	662431	3D finite element mesh
WAVE	156317	1059331	3D finite element mesh

setup, we obtained a peak bandwidth of 90 Mbytes and an effective startup time of 4 microseconds.

Since each processor on the T3D has only 64 Mbytes of memory, some of the larger graphs could not be partitioned on a single processor. For this reason, we compare the parallel run time on the T3D with the run time of the serial multilevel  $k$ -way algorithm running on an SGI Challenge with 0.5 Gbytes of memory and 150 MHz Mips R4400. Even though the R4400 has a peak integer performance that is 10% lower than the Alpha, due to the significantly higher amount of secondary cache available on the SGI machine (1 Mbyte on SGI versus 0 Mbytes on T3D processors), the code running on a single-processor T3D is about 20% slower than that running on the SGI. Since the nature of the multilevel algorithm discussed is randomized, we performed all experiments with fixed seed. Our extensive experiments with both parallel and serial multilevel  $k$ -way partitioning algorithms have shown that neither the quality nor the run time varies significantly for different randomized runs (they are usually within a few percentage points of each other), especially when the number of partitions is greater than eight.

*Partition Quality.* Table 6.2 shows the quality of the partitions produced by the parallel  $k$ -way algorithm as well as the amount of time it took to produce these partitions on a Cray T3D for the problems of Table 6.1. Partitions in 16, 32, 64, and 128 partitions are shown, each produced on 16, 32, 64, and 128 processors, respectively. Table 6.3 shows the quality of the partitions and the amount of time required by the serial algorithm running on the SGI for the same problems.

The quality of the partitions produced by the parallel algorithm relative to those produced by the serial  $k$ -way partitioning algorithm is shown graphically in Figure 6.1. From this figure we see that the edge-cut produced by the parallel algorithm is quite close to that produced by the serial algorithm. For most graphs, the edge-cut of the parallel algorithm is worse than that of the serial algorithm by at most a factor of 5%, while for some graphs, the parallel algorithm is somewhat better (by 1% to 3%). Since both the coarsening and uncoarsening phases of the parallel algorithm are similar (sections 4.2 and 4.4), the reason for the small deviation compared to the serial algorithm can be traced back to the use of nested dissection in the initial partitioning phase. However, the quality differences can be eliminated if multilevel bisection is used during the initial partitioning phase.

**Table 6.2** The performance of the parallel multilevel  $k$ -way partitioning algorithm on the Cray T3D. For each graph, the performance is shown for 16-, 32-, 64-, and 128-way partitions on 16, 32, 64, and 128 processors, respectively. The times are in seconds.

Graph name	16-way		32-way		64-way		128-way	
	Edge-cut	Time	Edge-cut	Time	Edge-cut	Time	Edge-cut	Time
144	44742	3.021	65845	1.956	87573	1.270	120365	0.977
598A	31211	2.436	47037	1.557	61225	1.039	90724	0.805
AUTO	91540	8.384	139760	5.071	197166	3.255	264662	2.215
BRACK2	13454	0.858	20675	0.589	30161	0.426	43937	0.389
COPTER2	20677	0.938	30714	0.638	42047	0.475	58470	0.424
M14B	50554	4.455	77944	2.834	108294	1.834	159825	1.383
MAP1	343	0.942	701	0.583	1174	0.398	1956	0.344
MDUAL	13144	2.637	20004	1.600	25575	1.058	35457	0.795
MDUAL2	24800	10.241	36227	5.778	50114	3.442	71355	2.250
OCEAN	10392	1.240	16529	0.799	25311	0.551	35846	0.446
ROTOR	25146	1.684	38134	1.128	53547	0.819	78163	0.670
WAVE	49502	1.914	72969	1.245	98572	0.894	131896	0.743

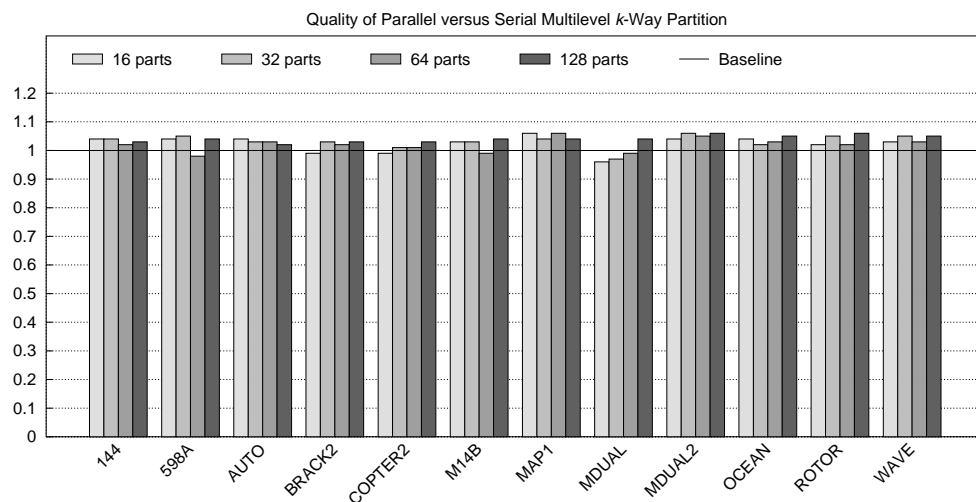
**Table 6.3** The performance of the serial multilevel  $k$ -way partitioning algorithm. For each graph, the performance is shown for 16-, 32-, 64-, and 128-way partitions. The times are in seconds on an SGI Challenge workstation.

Graph name	16-way		32-way		64-way		128-way	
	Edge-cut	Time	Edge-cut	Time	Edge-cut	Time	Edge-cut	Time
144	42987	12.140	63425	12.900	85967	13.620	116870	15.380
598A	30081	9.230	44604	9.320	62520	10.190	86891	11.050
AUTO	88125	48.490	135629	49.880	190508	51.640	259948	54.610
BRACK2	13539	3.680	20133	4.000	29515	4.520	42775	5.740
COPTER2	20852	3.970	30273	4.510	41672	5.160	56619	5.990
M14B	49029	18.830	75316	19.440	108874	20.560	153048	22.070
MAP1	323	9.580	674	10.020	1104	10.140	1881	11.210
MDUAL	13688	14.840	20715	15.890	25946	16.560	34235	18.790
MDUAL2	23891	74.050	34144	76.800	47628	76.910	67364	79.380
OCEAN	10033	7.160	16183	7.650	24483	8.370	34015	9.640
ROTOR	24723	7.140	36396	7.680	52463	8.380	73881	9.530
WAVE	47939	10.850	69370	11.490	95747	12.240	125925	14.100

*Parallel Run Time.* From Table 6.2 we can see that the run time of the parallel algorithm is very small. For 9 out of the 12 graphs, the parallel algorithm requires less than one second to produce a 128-way partitioning on 128 processors. Even for the larger graphs (AUTO with half a million vertices, and MDUAL2 with one million vertices) it requires only 2.2 seconds.

Table 6.4 shows the amount of time required by the different phases of the parallel graph partitioning algorithm for some of the graphs of our experimental testbed. Note that during the *communication setup* phase, the processors determine how many interface vertices they need to send and receive, and set up the appropriate data structures for this communication. From this table we see that, as the number of processors increases, the amount of time required by each phase decreases. The only exception is the initial partitioning phase, for which the time actually increases. This is because both the size of the coarsest graph and the number of partitions increase with the number of processors. However, the amount of time required by this phase is very small compared to the run time of the entire partitioning algorithm.



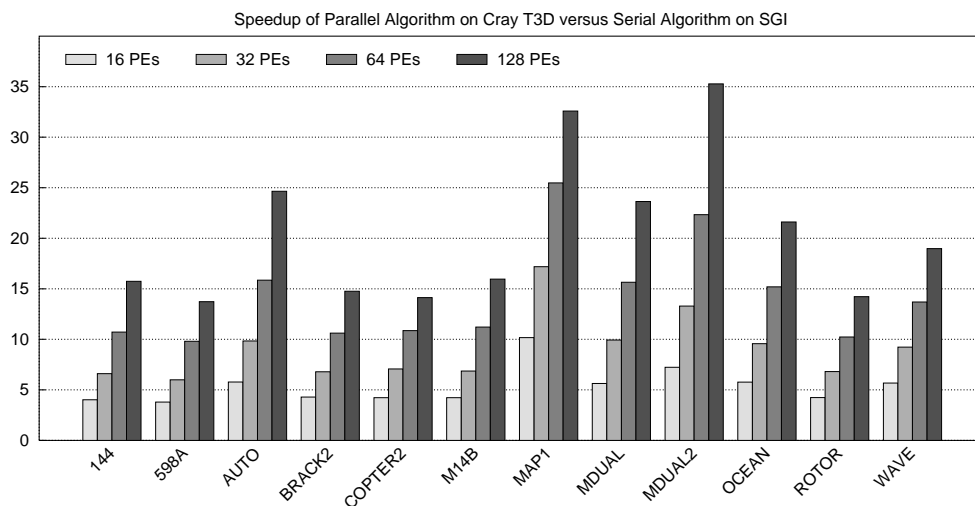


**Fig. 6.1** *Quality of the partitions produced by the parallel, relative to the serial, multilevel  $k$ -way partitioning algorithm. For each graph, the ratio of the edge-cut of the parallel to that of the serial algorithm is plotted for 16-, 32-, 64-, and 128-way partitions. Bars under the baseline indicate that the parallel algorithm produces partitions with a smaller edge-cut than the serial algorithm.*

**Table 6.4** *The amount of time (in seconds) required by the different phases of the parallel partitioning algorithm for some graphs, on 16 and 128 processors.*

Phase name	AUTO		MDUAL		MDUAL2	
	16 PEs	128 PEs	16 PEs	128 PEs	16 PEs	128 PEs
Communication setup	0.978	0.279	0.290	0.114	1.730	0.386
Graph coloring	2.480	0.477	0.581	0.102	2.239	0.351
Computing matching	1.271	0.353	0.458	0.111	1.752	0.385
Graph contraction	2.115	0.421	0.676	0.122	2.674	0.436
Initial partition	0.006	0.051	0.009	0.079	0.004	0.098
$k$ -way refinement	1.534	0.634	0.623	0.267	1.842	0.594
<b>Total run time</b>	<b>8.384</b>	<b>2.215</b>	<b>2.637</b>	<b>0.795</b>	<b>10.241</b>	<b>2.250</b>

The speedup achieved by the parallel algorithm on the Cray T3D relative to the serial algorithm running on the SGI is shown in Figure 6.2. For the smaller graphs, the parallel algorithm achieves a speedup in the range of 14 to 17 on 128 processors, and as the size of the graphs increases, the speedup improves to the 20 to 35 range. As discussed earlier, due to architectural differences between the Cray T3D and the SGI Challenge, the run time of the multilevel partitioning code running on a single processor of the SGI is somewhat smaller than that running on a single processor of the Cray T3D. Thus, the actual speedups (i.e., with respect to the serial algorithm running on a single processor of the Cray T3D) are higher by a factor of about 20%. Furthermore, as discussed in section 4, the parallel algorithm incurs the additional computational overhead of computing graph coloring during the coarsening phase, an overhead that it is not present in the serial algorithm. In addition to the coloring overhead, the parallel algorithm also requires a communication setup phase that is used to exchange information about the interface vertices. Again, on the serial algorithm, this overhead is not present. For instance, for AUTO, from Table 6.4

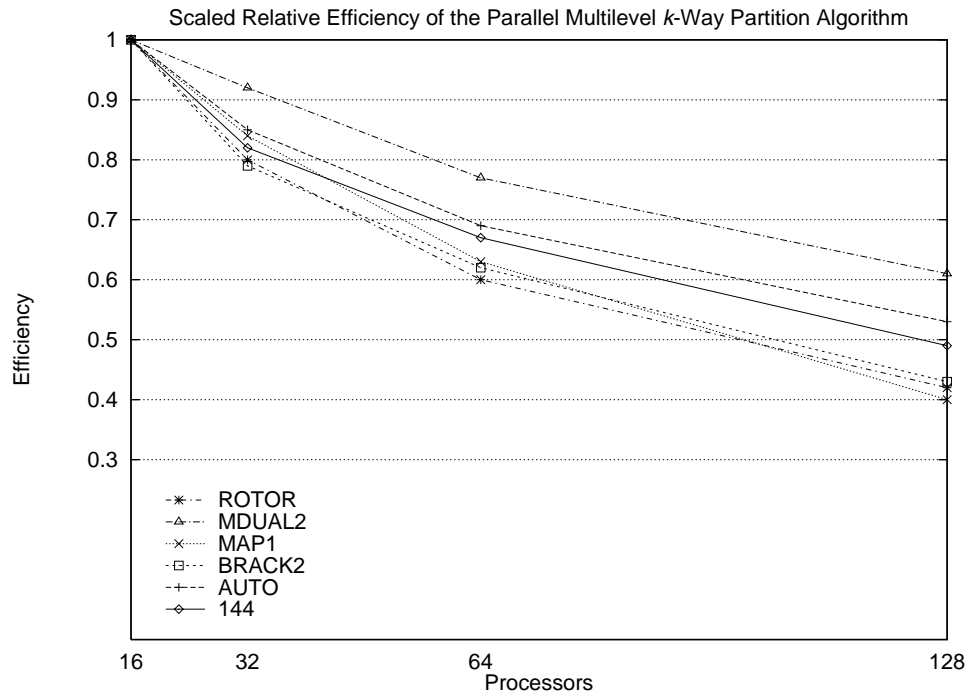


**Fig. 6.2** The speedup achieved by the parallel partitioning algorithm running on a Cray T3D relative to the serial algorithm running on an SGI. For each graph, the speedup on 16, 32, 64, and 128 processors is shown.

we see that out of the run time of 2.2 seconds on 128 processors, the coloring and communication setup overheads take 0.8 seconds, which is 36% of the total run time. Also note that for MAP1, MDUAL, MDUAL2, and OCEAN, for which the above two overheads are smaller (since these graphs have smaller average degrees), they achieve better speedup than other graphs with similar numbers of vertices.

**Experimental Scalability.** From Table 6.2, we see that for each graph, the run time of the parallel algorithm decreases as the number of processors and partitions increases. From Table 6.3, we see that the run time of the serial multilevel  $k$ -way partitioning algorithm increases as  $k$  increases. Since the asymptotic complexity of the serial algorithm is  $O(n)$  [18], this increase in run time is due to an increase in the number of interface vertices that exist as the number of partitions increases. Refining these interface vertices also leads to more work, but does not increase the asymptotic complexity of the algorithm. Evidence of the increased computational requirements during the  $k$ -way refinement can also be seen in Table 6.4. From this table we see that as the number of processors increases, the amount of time required for refinement decreases at a slower rate than the time required for coloring, matching, or contraction. For instance, for MDUAL, going from 64 to 128 processors, the run time of matching decreased by 43%, while the run time for refinement decreased by only 19%.

This modest increase in the computational requirements makes it hard to draw any conclusions about the experimental scalability of the parallel algorithm from the raw parallel run times in Table 6.2. However, from the serial run times, we know by how much the computational requirements increase as  $k$  increases. For this reason, we use the increase in the serial run time to compute the *scaled relative efficiencies* shown in Figure 6.3 for some graphs. These efficiencies are relative to the 16-processor run times, and they are scaled to reflect the increase in the computation. For example, we know from Table 6.3 that, for AUTO, going from a 16-way to a 128-way partition, the run time increases from 48.49 seconds to 54.61 seconds, a 12.6% increase. To compute the speedup of the parallel algorithm on 128 processors relative to that on



**Fig. 6.3** *The scaled efficiencies achieved by the parallel algorithm for some graphs. The efficiencies are relative to the 16-processor runs, and the run times are scaled to reflect the increase in the amount of work performed as the number of partitions increases.*

16 processors, we multiply the run time on 16 processors (8.38 seconds) by 1.126 (i.e., a 12.6% increase in computational requirements), and divide it by the run time on 128 processors (2.21 seconds). This relative speedup is 4.27, thus yielding a relative efficiency of 0.53 (since  $128 = 8 * 16$ ). From Figure 6.3 we see that for any graph, as the number of processors increases, the efficiency decreases. This is to be expected for any nontrivial parallel algorithm, since the communication overhead increases with the number of processors. Similarly, as the size of the graphs increases, the achieved efficiency improves because the communication overhead increases more slowly than the amount of computation performed.

From the analysis in section 5, we have shown that the isoefficiency function of our parallel algorithm is  $O(p^2 \log p)$ . That is, in order to maintain a fixed efficiency, the graph size should increase as  $O(p^2 \log p)$  [22]. For example, if we double the number of processors, then we need to increase the size of the graph by a factor a little over 4 in order to achieve the same efficiency. In order to evaluate the scalability of the algorithm experimentally, we use the speedup obtained by the parallel algorithm over the serial algorithm running on the SGI. Because of the differences between the serial and parallel algorithms discussed earlier (additional use of coloring by the parallel algorithm), it is important to compare the efficiencies achieved on graphs that have a similar structure, so they will lead to similar coloring overheads. Among the graphs in our experimental testbed (Table 6.1), the following pairs of graphs (144 with AUTO and MDUAL with MDUAL2) require the same number of colors, and they have appropriate relative sizes. AUTO is about 3.1 times larger than 144, while MDUAL2

**Table 6.5** The amount of time (in seconds) required by the different phases of the parallel partitioning algorithm for different initial vertex distributions, on 16 and 128 processors. The columns labeled “Rand” correspond to a random distribution of the graph, whereas the columns labeled “PrePart” correspond to a prepartitioned distribution of the graph.

	AUTO				MDUAL2			
	16 PEs		128 PEs		16 PEs		128 PEs	
Phase name	Rand	PrePart	Rand	PrePart	Rand	PrePart	Rand	PrePart
Communication setup	1.002	0.391	0.288	0.180	1.732	0.447	0.403	0.311
Graph coloring	2.503	1.840	0.493	0.231	2.257	1.384	0.354	0.191
Computing matching	1.265	0.726	0.362	0.129	1.772	0.812	0.386	0.139
Graph contraction	2.122	1.192	0.429	0.163	2.692	1.296	0.438	0.181
Initial partition	0.007	0.005	0.060	0.054	0.004	0.010	0.088	0.075
$k$ -way refinement	1.541	1.216	0.663	0.550	1.853	1.264	0.597	0.400
<b>Total run time</b>	8.430	5.370	2.295	1.310	10.310	5.213	2.266	1.297

is about 3.83 times larger than MDUAL. From Figure 6.2 we see that the speedup achieved by AUTO (and MDUAL2) on 32, 64, and 128 processors are comparable to the speedup achieved by 144 (and MDUAL) on 16, 32, and 64 processors, respectively. Thus, the experiments confirm that the isoefficiency function of our parallel graph partitioning algorithm is  $O(p^2 \log p)$ .

*Effects of Initial Graph Distribution.* The experiments shown in Table 6.2 were performed by initially distributing the graphs between the processors in a block distribution. That is, as the graphs were read from the file, consecutive  $n/p$  vertices were assigned to each processor. We refer to this as the *as-is distribution*. This ordering is somewhat different than the random distribution that was assumed in the description and analysis of the parallel algorithm (sections 4 and 5) and was chosen for its simplicity. To study the performance of our parallel algorithm under different initial graph distribution schemes, we performed experiments using both random and prepartitioned distributions. In both cases, a permutation was applied to the graph before distribution onto the processors. In the case of random distribution, this permutation was computed randomly, whereas in the case of the prepartitioned distribution, this permutation was computed from a serial  $p$ -way partitioning of the graph.

Table 6.5 shows the run times of these two different distribution schemes for two of the larger graphs in our experimental testbed. Comparing these run times with those shown in Table 6.4, we see that there is little difference between the random and the as-is distributions. The run time of the random distribution is only higher by less than 1%, which was expected, since both distributions result in initial partitions that cut more than 90% of the edges. However, the run time is significantly reduced when the prepartitioned distribution is used. For example, in the case of MDUAL2 on 16 processors, the run time of the prepartitioned distribution is almost half that achieved by either the random or the as-is distribution. This reduction in run time is due to the following two reasons: (a) reduced communication requirements, and (b) better cache utilization.

For the prepartitioned graph distribution, the number of edges that get cut as a result of the initial distribution is significantly reduced to only 7.5% for AUTO, and 3.6% for MDUAL2. Consequently, the distributed graph has significantly fewer interface vertices. In each of the graph coloring, matching, contraction, and partitioning refinement algorithms, communication takes a significant fraction of the overall run time. Therefore, reduction in the run time is due to the reduced communication required for the prepartitioned graph. The reduced communication requirements can be

clearly seen in the amount of time required by the *communication setup* phase (especially for 16 processors), whose complexity depends highly on the number of interface vertices. Besides reducing communication overheads, the much better data locality that is produced by the prepartitioned distribution also significantly improves cache utilization. This is particularly important on a machine like the Cray T3D, since it has only a small amount of primary cache (8 kbytes) and no secondary cache. This improved cache reuse is the primary reason for the almost 50% improvement achieved by the coloring, matching, and contraction algorithms. The primary significance of the cache can also be seen when looking at the time required by the  $k$ -way refinement. In this case, the improvements are not as dramatic (somewhere between 27% and 40% on 16 processors). This is because, during  $k$ -way refinement, only a few vertices get moved; hence, there is limited cache reuse.

**7. Conclusion.** In this paper we presented a parallel formulation of the multi-level  $k$ -way partitioning algorithm. We show that the algorithm is scalable for a class of graphs that includes the commonly used finite element meshes. In particular, the time taken by our parallel graph partitioning algorithm is only slightly longer than the time taken for rearrangement of the graph among processors according to the new partition. Experiments with a variety of finite element graphs show that our parallel formulation produces high-quality partitioning in a short amount of time. For example, a 128-way partitioning of graphs with one million vertices can be computed in a little over two seconds on a 128-processor Cray T3D. Furthermore, the quality of the produced partitions is comparable (edge-cuts within 5%) to that of the partitions produced by the serial multilevel  $k$ -way algorithm. Even though the parallel formulation is implemented for non-cache-coherent shared-address space architectures such as the Cray T3D, the formulation can be easily adapted for message passing architectures such as the IBM SP2. On such architectures, all interactions are done via message passing that has much larger startup latency than that of the one-sided communication operations on architectures such as the Cray T3D. Due to this worse ratio of computation and communication, comparable efficiency will be obtained only for proportionately larger graphs.

Even though our parallel formulation is able to obtain high efficiencies on up to 128 processors, obtaining similar efficiencies on much larger numbers of processors (thousands) will be much harder, as our scalability analysis of this algorithm shows that its isoefficiency function is  $O(p^2)$ . This term is due to an operation similar to an all-to-all broadcast inherent in the setup, matching, and refinement phases. If the graph is considered to be completely randomly distributed, then this  $O(p^2)$  term in the isoefficiency cannot be avoided, as even the redistribution of the graph would require a similar all-to-all operation. However, if the parallel graph partitioner is being used only to repartition an adaptively refined graph, then this  $O(p^2)$  can be eliminated from the isoefficiency by appropriately modifying the setup, coloring, matching, and refinement phases to take advantage of this inherent locality.

#### REFERENCES

- [1] S. T. BARNARD, *PMRSB: Parallel multilevel recursive spectral bisection*, in Supercomputing 1995, ACM and IEEE Computer Society, San Diego, CA, 1995.
- [2] S. T. BARNARD AND H. SIMON, *A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes*, in Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1995, pp. 627–632.

- [3] S. T. BARNARD AND H. D. SIMON, *A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems*, in Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1993, pp. 711–718.
- [4] T. N. BUI AND C. JONES, *A heuristic for reducing fill-in in sparse matrix factorization*, in Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1993, pp. 445–452.
- [5] J. CONG AND M. L. SMITH, *A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design*, in Proc. ACM/IEEE Design Automation Conference, Dallas, TX, 1993, pp. 755–760.
- [6] K. D. DEVINE AND J. E. FLAHERTY, *Dynamic load balancing for parallel finite element methods with adaptive h- and p-refinement*, in Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1995, pp. 593–598.
- [7] P. DINIZ, S. PLIMPTON, B. HENDRICKSON, AND R. LELAND, *Parallel algorithms for dynamically partitioning unstructured grids*, in Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1995, pp. 615–620.
- [8] A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.
- [9] J. R. GILBERT AND E. ZMIJEWSKI, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, Internat. J. Parallel Programming, (1987), pp. 498–513.
- [10] A. GUPTA, G. KARYPIS, AND V. KUMAR, *Highly scalable parallel algorithms for sparse matrix factorization*, IEEE Trans. Parallel and Distributed Systems, 8 (1997), pp. 502–520; also available online from <http://www.cs.umn.edu/~karypis>.
- [11] S. HAUCK AND G. BORRIELLO, *An evaluation of bipartitioning technique*, in Proc. Chapel Hill Conf. on Advanced Research in VLSI, IEEE Computer Society, San Diego, CA, 1995.
- [12] M. T. HEATH AND P. RAGHAVAN, *A Cartesian parallel nested dissection algorithm*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 235–253.
- [13] B. HENDRICKSON AND R. LELAND, *A Multilevel Algorithm for Partitioning Graphs*, Tech. Rep. SAND93-1301, Sandia National Laboratories, Albuquerque, NM, 1993.
- [14] Z. JOHAN, K. K. MATHUR, S. L. JOHNSON, AND T. J. R. HUGHES, *Finite Element Methods on the Connection Machine CM-5 System*, Tech. Rep., Thinking Machines Corporation, Burlington, MA, 1993.
- [15] M. T. JONES AND P. E. PLASSMANN, *A parallel graph coloring heuristic*, SIAM J. Sci. Comput., 14 (1993), pp. 654–669.
- [16] G. KARYPIS AND V. KUMAR, *Fast Sparse Cholesky Factorization on Scalable Parallel Computers*, Tech. Rep., Department of Computer Science, University of Minnesota, Minneapolis, 1994; a short version appears in the *Eighth Symposium on the Frontiers of Massively Parallel Computation*, IEEE Computer Society, San Diego, CA, 1995. Also available online from <http://www.cs.umn.edu/~karypis>.
- [17] G. KARYPIS AND V. KUMAR, *METIS3.0: Unstructured Graph Partitioning and Sparse Matrix Ordering System*, Tech. Rep. 97-061, Department of Computer Science, University of Minnesota, Minneapolis, 1997; also available online from <http://www.cs.umn.edu/~metis>.
- [18] G. KARYPIS AND V. KUMAR, *Multilevel k-way partitioning scheme for irregular graphs*, J. Parallel Distrib. Comput., 48 (1998), pp. 96–129; also available online from <http://www.cs.umn.edu/~karypis>.
- [19] G. KARYPIS AND V. KUMAR, *A parallel algorithm for multilevel graph partitioning and sparse matrix ordering*, J. Parallel Distrib. Comput., 48 (1998), pp. 71–95; also available online from <http://www.cs.umn.edu/~karypis>. A short version appears in Proc. Internat. Parallel Processing Symposium, CRC Press, Boca Raton, FL, 1996.
- [20] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., 20 (1998), pp. 359–392; also available online from <http://www.cs.umn.edu/~karypis>. A short version appears in Proc. Internat. Conf. on Parallel Processing, CRC Press, Boca Raton, FL, 1995.
- [21] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell System Tech. J., 49 (1970), pp. 291–307.
- [22] V. KUMAR, A. GRAMA, A. GUPTA, AND G. KARYPIS, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, Redwood City, CA, 1994.
- [23] M. LUBY, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comput., 15 (1986), pp. 1036–1053.
- [24] A. POTHEN, H. D. SIMON, AND K.-P. LIOU, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452.
- [25] P. RAGHAVAN, *Parallel Ordering Using Edge Contraction*, Tech. Rep. CS-95-293, Department of Computer Science, University of Tennessee, Knoxville, 1995.

- [26] E. ROTHBERG, *Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers*, in Proc. 1994 Scalable High Performance Computing Conference, IEEE Computer Society, San Diego, CA, 1994.
- [27] K. SCHLOEGEL, G. KARYPIS, AND V. KUMAR, *Multilevel diffusion algorithms for repartitioning of adaptive meshes*, J. Parallel Distrib. Comput., 47 (1997), pp. 109–124; also available online from <http://www.cs.umn.edu/~karypis>.
- [28] R. V. SHANKAR AND S. RANKA, *Random data accesses on coarse-grained parallel machine*, J. Parallel Distrib. Comput., 44 (1997), pp. 24–34.
- [29] C. WALSHAW, M. CROSS, AND M. G. EVERETT, *Dynamic load-balancing for parallel adaptive unstructured meshes*, in Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1997.