A MULTILEVEL ALGORITHM FOR PARTITIONING GRAPHS

BRUCE HENDRICKSON† AND ROBERT LELAND‡
SANDIA NATIONAL LABORATORIES
ALBUQUERQUE, NM 87185–1110

Abstract.
The *graph partitioning problem* is that of dividing the vertices of a graph into sets of specified sizes such that few edges cross between sets. This NP–complete problem arises in many important scientific and engineering problems. Prominent examples include the decomposition of data structures for parallel computation, the placement of circuit elements and the ordering of sparse matrix computations. We present a multilevel algorithm for graph partitioning in which the graph is approximated by a sequence of increasingly smaller graphs. The smallest graph is then partitioned using a spectral method, and this partition is propagated back through the hierarchy of graphs. A variant of the Kernighan-Lin algorithm is applied periodically to refine the partition. The entire algorithm can be implemented to execute in time proportional to the size of the original graph. Experiments indicate that, relative to other advanced methods, the multilevel algorithm produces high quality partitions at low cost.

Key words. graph partitioning, parallel computation, load balancing, circuit placement

AMS(MOS) subject classification. C5C85, 68R10, 65Y05

1. Introduction. The efficient use of a distributed memory parallel computer requires that the computational load be balanced in such a way that the cost of interprocessor communication is kept small. For many applications in scientific computing, the problem of decomposing a calculation among processors can be conveniently described in the language of graphs. A vertex in the graph represents a computation, while an edge between two vertices indicates a data dependency. For the calculation to run efficiently on a parallel machine, the graph must be broken into pieces with approximately equal number of vertices but with few edges crossing between pieces. This graph model has been used to model finite element and finite difference calculations, molecular dynamics simulations, particle–in–cell codes and a variety of other computations. Graph partitioning also has applications in devising efficient circuit layouts, generating efficient orderings for sparse matrix calculations and other important problems. Unfortunately, finding an optimal partition of a graph is known to be NP–complete [6], but the practical importance of the problem has inspired a variety of heuristics. This paper presents a new, multilevel algorithm for graph partitioning that runs in linear time and produces high quality partitions.

The algorithm described in this paper was largely inspired by the work of Barnard and Simon [1], in which they use a multilevel approach to calculate an eigenvector needed for a spectral partitioning algorithm. We decided to investigate avoiding the challenging numerical problem of transferring eigenvectors between levels by instead transferring *partitions* between levels. Our new algorithm has three phases. First, we

† Department 1422, email bah@cs.sandia.gov.
‡ Department 1424, email leland@cs.sandia.gov.
* Will appear in Proc. Supercomputing '95.

construct a sequence of increasingly coarse approximations to a graph. Second, we partition the smallest graph in the sequence. We use a spectral method [13, 21] for this problem, but in principle any partitioning algorithm could be used. Third, we project the coarse partition back through the sequence of graphs, periodically improving it with a local refinement algorithm. For this local improvement phase we use a variant of a popular algorithm originally devised by Kernighan and Lin [17] and improved by Fiduccia and Mattheyses [5], but other methods could be used as well.

This approach has several attractive features. First, the costs of constructing coarse graphs and of the local improvement algorithms are both proportional to the number of edges in the graph. Second, there is a close analogy with the familiar multigrid methods for numerical problems, with the local improvement scheme taking the place of the multigrid smoother. On the coarsest graph, moving vertices between partitions corresponds to moving a large collection of vertices on the original graph. Thus as we traverse the sequence of graphs, we make increasingly more refined corrections to the partition.

An algorithm with this same structure was independently discovered by Bui and Jones [2, 15]. The primary difference between their approach and ours is that as we construct coarse graphs we modify edge and vertex weights to preserve, in a weighted sense, any constraints on partition size and any cost function of edge crossings. Thus the essential properties of a partition are preserved across levels. Another graph partitioning algorithm is described in [18] in which a sequence of coarse graphs is constructed in a manner similar to ours. However, in that paper no local refinement is applied so the final partitioning is just the injection of the coarse partitioning. An algorithm that is similar in spirit but quite different in detail is described by Vanderstraeten, et al. [24].

The structure of this paper is as follows. In §2 we present our algorithm more formally, focusing on the construction of the sequence of coarse graphs. In §3 we describe our local refinement algorithm which is critical to the overall effectiveness of the multilevel approach. Results on some sample graphs are presented in §4 and compared against other partitioning algorithms. The paper is summarized and conclusions are drawn in §5.

2. A multilevel graph partitioning algorithm. The problem we wish to solve is shown in Fig. 1.

> **Given** A graph $G = (V, E)$ with (possibly unitary) weights[1] on the edges and vertices and a parameter p,
>
> **Find** A partitioning of the vertices of G into p sets in such a way that the sums of the vertex weights in each set are as equal as possible, and the sum of the weights of edges crossing between sets is minimized.

Fig. 1. The graph partitioning problem.

As mentioned, the graph partitioning problem is NP-complete, and we therefore should not expect to solve it in polynomial time. We instead seek an approximate solution with acceptable execution time. One common approach to the general problem is to reduce it to a sequence of bisection steps. That is, first divide the graph into two pieces, then recursively bisect the two subpieces independently. This approach can generate an arbitrary number of equal sized sets if the bisections aren't required to divide evenly. For example, if nine sets are desired, the first cut should divide into pieces of size 4/9 and 5/9, and then recursively divide the two pieces until there are nine total. Unfortunately, graph bisection is also NP-complete, so this simplification doesn't resolve the intractability of the original problem. Also, this simplification has several inherent shortcomings. First, bisection algorithms are unable to accept a less attractive initial cut that will lead to net savings in later cuts. Second, the application may require the minimization of a more complicated function in which edges between some sets are more costly than edges between others. For example, in parallel computing the overhead due to a message may depend upon how far apart the communicating processors are. Bisection algorithms have a difficult time minimizing this kind of metric. In fact, graphs can be constructed on which an optimal bisection algorithm will do badly. All other things being equal, it is preferable to divide into as many sets at once as possible so as to limit the depth of the recursion. Our algorithm allows for this generality.

As mentioned above, there are three different stages to our multilevel graph partitioning algorithm. First, a sequence of smaller and smaller graphs is created from the original graph. Second, the smallest graph in the sequence is partitioned carefully. And third, the partition is propagated back through the sequence of grids, with an occasional local refinement. This structure is sketched in Fig. 2, and different stages are discussed in detail below. The algorithm due to Bui and Jones [2, 15] has a similar structure, but differs in the details. In particular, unlike their algorithm, ours uses edge and vertex weights to preserve in the coarse graphs as much structure of the original graph as possible (see detailed explanation below). This permits application of the algorithm to weighted graphs and natural extensions to other contexts, for example that of the terminal propagation technique used in VLSI layout [4, 14].

Coarsening is advantageous because is that the number of possible partitions grows exponentially with the number of vertices in the graph, so a good partition of the coarse graph is much easier to find than a good partition of the original one. The price paid for this reduction in complexity is that only a small number of the possible fine graph partitions are represented and are therefore examinable on the coarse graph. However, by working on different levels, the local refinement scheme improves the partition on multiple scales. Although the partition of the coarse graph may not directly induce a very good partition of the original graph, the repeated application of the local refinement largely resolves this problem.

[1] The use of edge and vertex weights can be important in some applications. For example, in circuit layouts weighted graphs are often used to model net lists, and in parallel computing they can model inhomogeneous computation and communication.

```
(1)     Until graph is small enough
            graph := coarsen(graph)

(2)     Partition graph

(3)     Until graph = original graph
            graph := uncoarsen(graph)
            partition := uncoarsen(partition)
            locally refine partition if desired
```

Fig. 2. A multilevel algorithm for graph partitioning.

2.1. Constructing a coarse graph. The fundamental step in our coarsening scheme is the edge contraction operation. In this step, two vertices joined by an edge are merged, and the new vertex retains edges to the union of the neighbors of the merged vertices. The weight of the new vertex is set equal to the sum of the weights of its constituent vertices. Edge weights are left unchanged unless both merged vertices are adjacent to the same neighbor. In this case the new edge that represents the two original edges is given a weight equal to the sum of the weights of the two edges it replaces. So, for example, contracting one edge of a triangle with unit edge and vertex weights would produce a single edge of weight two joining vertices of weight one and two.

A coarsening step in our algorithm requires the contraction of a large number of edges that are well dispersed throughout the graph. This can be accomplished by first finding a *maximal matching*. This is a maximal set of edges, no two of which are incident on the same vertex. Maximal matchings can be generated quickly using a depth-first search or a randomized algorithm. In our maximal matching, the vertices were visited in a random order, and if not already matched, a random unmatched neighbor was selected and the edge included in the maximal matching. This simple algorithm requires time proportional to the number of edges in the graph.

Our algorithm for graph coarsening can now be sketched in Fig. 3. First a maximal matching in the graph is identified. Each edge in the matching is then contracted, and vertex and edge weights are adjusted as previously described. This coarsening procedure has the property that each vertex in the fine graph is mapped to a unique vertex in the coarse graph.

This coarsening procedure has a number of attractive properties. First any partition of the coarse graph corresponds naturally to a partition of the fine graph. Second, since vertex weights are summed, constraints on the set sizes that depend just on the number of vertices in a set are preserved in a weighted sense in the coarse graph. Third, since edge weights are combined, any linear penalty function on the edges crossing between partitions will be preserved as a weighted metric under the coarsening process. Thus for most metrics of partition quality, a good partition of the coarse graph will correspond to a good partition of the fine graph. Lastly, this coarsening algorithm is fast. The

Find a maximal matching in the graph

For Each matching edge (i, j)
    Contract edge to form new vertex v
    Vertex weight(v) := weight(i) + weight(j)
    If i and j are both adjacent to a vertex k **Then**
        Edge weight(v, k) := weight(i, k) + weight(j, k)

Fig. 3. Constructing a coarse approximation to a graph.

entire operation can be implemented in time proportional to the number of edges in the original graph.

2.2. Partitioning the coarsest graph. The details of the partitioning of the coarsest graph are not central to the algorithm, so we will not dwell on them. However, it is important to note that the partitioning algorithm must be able to handle edge and vertex weights, even if the original graph is unweighted. Our implementation uses a spectral partitioner for the coarse graph which requires one, two or three eigenvectors of the Laplacian matrix of a graph to partition into two, four or eight sets respectively [13]. This is followed by an invocation of the Kernighan–Lin refinement algorithm described in §3 which can refine a partition into an arbitrary number of sets with any inter–set cost metric. By dividing into more than two sets at once, the algorithm can attempt to minimize more complicated cost functions. For example, in parallel computing it is generally desirable to keep message traffic between architecturally close processors. This consideration can't easily be included in a recursive bisection approach, but it can be addressed when a greater amount of partitioning is performed at once. In our experience, virtually any initial partitioner followed by Kernighan–Lin works well on small graphs, so the use of spectral partitioning is not essential to the overall algorithm.

2.3. Uncoarsening the partition. The uncoarsening of a partition is trivial. Each vertex in a coarse graph is simply the union of one or two vertices from a larger graph. We simply assign a vertex from the larger graph to the same set as its coarse graph counterpart. Since the weight of the coarse vertex was the sum of the weights of its constituents, the uncoarsening procedure preserves the sum of the vertex weights in each set. And it similarly preserves the sum of edge weights connecting different sets.

The uncoarsened graph has more degrees of freedom than the smaller coarse graph. Consequently, the best partition for the coarse graph may not be optimal for its uncoarsened counterpart. We therefore apply the local refinement scheme described in the next section to the uncoarsened partition.

3. Locally refining the partition. For our multilevel graph partitioning algorithm to be useful it must incorporate a local refinement scheme that is both fast and effective. In this section we describe our implementation of one such method, a generalization of the graph bisection algorithm originally due to Kernighan and Lin (KL) [17]. This algorithm is quite good at finding locally optimal answers, but unless it is initialized with a good global partition, the resulting local optimum can be far from the best possible. However, since the multilevel nature of our algorithm allows the refinement technique to work on different scales, the local finesse of KL is all we require.

Several improvements to the original KL algorithm have been developed over the years, including the important linear time implementation described by Fiduccia and Mattheyses (FM) [5]. Our implementation is closely patterned after the algorithm in [5], but is generalized and optimized in several ways. First, we can refine a partition into an arbitrary number of sets, rather than just two sets. A generalization to four sets is described by Suaris and Kedem for circuit layout problems [22], and we have extended their idea. Second, we allow for integer weights on edges and vertices (Fiduccia and Mattheyses allow for weighted vertices). Third, we can handle an arbitrary (integral) inter–set cost metric. This can be important in mapping parallel computations since the cost of sending a message between two processors may depend on how near these processors are in the machine. Fourth, we include an element of randomness in the algorithm which improves robustness. Fifth, when only a small fraction of the vertices are moved, we are able to reuse computations in a manner that significantly reduces the overall run time of the algorithm. And sixth, we use a lazy evaluation strategy to minimize the number of gain values computed, and thereby significantly reduce run time. These generalizations complicate the implementation in a variety of ways which are discussed in detail below.

The fundamental idea behind all KL–like algorithms is the concept of the *gain* associated with moving a vertex into a different set. The gain is simply the net reduction in the weight of cut edges that would result from switching a vertex to a different set. More formally, if vertex i were to move from set l to set k, the corresponding gain $g^k(i)$ can be expressed as

$$g^k(i) = \sum_{(i,j)\in E} \begin{cases} w_{ij}c_{kl} & \text{if } P(j) = k \\ -w_{ij}c_{kl} & \text{if } P(j) = l \\ w_{ij}(c_{lm} - c_{km}) & \text{if } P(j) = m, m \neq k, m \neq l, \end{cases} \tag{1}$$

where $P(j)$ is the current set for vertex j, w_{ij} is the weight of the edge between vertices i and j, and c_{lk} is the symmetric inter–set cost metric for an edge between sets l and k.

Clearly, if the gain associated with moving a vertex is positive, then making that move will reduce the total cost of the edges cut in the partition. We could simply make all such moves and see what partition results, but this approach has two shortcomings. First, we need to enforce some constraints on how large the difference in set sizes can become and hence may disallow certain moves. Second, a purely greedy strategy that stops when no single allowed move improves the partition may miss more complicated sequences of moves that lead to a net improvement. Hence the algorithm continues for some time to consider moves that actually make the partition worse. The basic structure of our algorithm is sketched in Fig. 4.

The algorithm consists of two nested loops. The inner loop presides over a sequence of moves of vertices from one set to another. The outer loop continues allowing attempted sequences until no further improvement is detected. We will refer to a single

```
Until No better partition is discovered
    Best Partition := Current Partition
    Compute all initial gains
    Until Termination criteria reached
        Select vertex to move
        Perform move
        Update gains of all neighbors of moved vertex
        If Current Partition balanced and better than Best Partition Then
            Best Partition := Current Partition
    End Until
    Current Partition := Best Partition
End Until
```

Fig. 4. An algorithm for refining graph partitions.

iteration of the outer loop as a *pass* of the algorithm. At each step in the inner loop, a vertex is selected to be moved from one set to another. To avoid the possibility of an infinite loop we insist that a vertex may be moved at most once within a single pass. The move that is selected is the one with the largest gain, subject to some balance constraints. It is important to note that this gain may be negative; that is, the current move may make the partition worse. However, several moves that reduce the partition quality may lead to later moves that more than compensate for the initial regression. This ability to climb out of local minima is a crucial feature of the KL algorithm.

When a vertex is moved, the gains of all its neighbors must be modified. Then another move is selected and the process repeated. The best partition that is encountered in this sequence is recorded and the data is moved into that configuration prior to the start of the next pass of the algorithm. In practice, the number of passes required is quite small so the run time of the algorithm consists of two terms, a startup cost and a cost proportional to the execution time of a single pass through the inner loop.

Another important detail is the data structure used for representing gains. The key difference between the KL and FM algorithms is that Fiduccia and Mattheyses use a more complicated data structure that allows a single pass through the outer loop to be performed in time proportional to the number of edges in the graph. The basic idea is to bucket sort the gains associated with all possible moves. Each bucket consists of a doubly linked list of moves with a particular gain value. Selecting a move with the highest gain simply requires finding the highest ranked nonempty bucket. The gains of neighbor vertices are updated by moving the appropriate markers from one bucket to another, which can be done in constant time.

Up to this point, the discussion describes Fiduccia and Mattheyses' algorithm. Our algorithm extends the data structure from FM to allow for an arbitrary number of sets. If there are s sets, n vertices and m edges, then there are $s(s-1)$ different types of moves. We maintain a bucket sorted data structure for each of these move types. Since each vertex can move to any of $s-1$ other sets, this requires memory of size $\Theta(n(s-1))$. Before entering the innermost loop, gain values are computed and placed in the appropriate doubly linked lists comprising the different buckets. If all the gains are computed, the cost of this operation is $O((s-1)n+m)$. However, using the tricks described in §3.4 and §3.5, only a fraction of the gains need be computed. If the innermost loop continues through all vertices, move selection requires time $O(s(s-1)n)$, and the remainder of the inner loop requires $O(sm)$ time. However, for large problems the innermost loop always ejects early due to the termination criteria discussed in §3.2 below. Hence, the execution time of KL can be sublinear in the size of the graph, and for large problems, the performance of the overall algorithm is limited by the time spent constructing the sequence of coarse graphs.

3.1. Move selection. To select the next move, the algorithm finds the top bucket for all the $s(s-1)$ different types of moves. Rather than simply choosing whichever of these moves has the highest gain, the algorithm also considers the effect of each move on set balance[2]. Balance is encouraged by considering only moves from sets of at least average size to those that are at or below average size. This does not guarantee that a balanced partition will result, but in practice it seems to work quite well.

We note that move selection can be easily modified to allow for nonequal set sizes. To do so, in the selection of a move, we simply compare the size of a set to its desired size, which may not be the same as the desired size of the other sets. Moves are then allowed only from a set at least as large as desired to a set that is at least as small as desired.

3.2. Termination criteria. Our implementation has two types of termination criteria. The most obvious condition is when there are no possible moves from large to small sets. The second condition allows for early termination when further improvement seems unlikely. This is particularly important in the context of our multilevel partitioner since we expect the local refinement algorithm to make only small changes in the partition. In the second condition, we monitor the difference in quality between the best partition yet encountered and the current partition. When this difference becomes large the inner loop is exited. A discussion of other possible termination criteria can be found in [7].

3.3. Randomization. Since there are typically many different vertex moves that have the same gain value, the manner in which ties are broken can significantly affect the results. Our implementation avoids this issue by using randomization. The initial gain values of the vertices are computed in random order. When a value is updated, the corresponding vertex is placed at the front of the list of vertices with its gain value. Thus, collections of adjacent vertices are likely to be considered together, which is in keeping with the spirit of the overall algorithm. Also, with randomization a failure in the outermost loop to find any better partition does not necessarily mean that no

[2] We consider a partition to be balanced if the difference between the largest and smallest set sizes is no larger than the largest vertex weight.

further improvement is possible. Another pass through the algorithm may manage to find a better partition.

3.4. Avoiding recomputing gains. Our multilevel partitioning algorithm typically provides a fairly good partition to the refinement algorithm, so only a small number of vertices must be moved. In this case, the cutoffs typically come into play quite quickly and for large problems the cost of the algorithm is dominated by the initial calculation of gain values. Although these values need to be computed once, a simple observation alleviates the need to recompute and sort most of them. The only vertices whose gain values are modified in a single pass of the algorithm are (1) the vertices that are selected to be moved and (2) the neighbors of these vertices. All other vertices remain unaffected. By reconsidering only the necessary vertices, the full bucket sort is performed just once rather than on every pass through the outer loop. In the absence of the lazy evaluation technique described in §3.5, this simplification reduces the overall run time of the multilevel algorithm by 30% to 95%.

3.5. Lazy initialization. As observed by several other researchers [25, 16, 26], when initialized with a good partition a local refinement algorithm need only move vertices that are on or near the initial set boundaries. In principle, the algorithm need never compute gain values for most of the vertices since they aren't near a boundary. Unfortunately, exploiting this observation requires determining which vertices are on the boundary, a task which is generally not much cheaper than computing all the gains. However, if these vertices are already known, a significant reduction in run time is possible.

Fortunately, in the multilevel algorithm the identity of vertices near the boundary can be easily propagated between levels during the uncoarsening. On the smallest graph all vertices are treated as boundary vertices. After each invocation of KL the vertices on the boundary are determined. The uncoarsened counterparts of these boundary vertices are the only vertices initialized in the next call to KL.

When running KL in this way, it is possible that a vertex which was not initialized should be selected for movement. To deal with this possibility we use a *lazy* evaluation strategy in which uninitialized vertices are initialized on an as–needed basis. Specifically, when we select a vertex to move, we try to update the gain values of all its neighbors. If a neighbor is uninitialized, we activate and initialize it then. Since for large graphs only a small fraction of the vertices are involved in a pass of KL, this avoidance of unnecessary calculation results in a significant performance improvement. Similar strategies are utilized by Walshaw, et al. [26] and Karypis and Kumar [16].

4. Results and evaluation. We have implemented the algorithm described in the previous sections in the C programming language as part of the **Chaco 2.0** graph partitioning code [12]. We ran our code on several graphs corresponding to computational grids for large scientific computing problems. Vertices of these grids represent computation, while edges indicate data dependencies. To perform these scientific computations on a parallel machine, the grids must be partitioned into pieces which are assigned to processors. For optimal performance, each processor should have the same number of vertices to ensure that the work load is balanced, and since edges crossing between sets represent communication, the number of these cross edges must be kept small.

We compared the run times and partitions produced by our multilevel algorithm against several other partitioning strategies that are popular in the parallel computing community and also implemented in **Chaco 2.0**. The first competing algorithm is *inertial* bisection, descriptions of which can be found in [21, 27]. This approach ignores the graph, and instead uses geometric information which is typically available for meshes used in scientific and engineering computations. Each node is assigned a unit weight, and the principle axis of the resulting structure is determined. The grid is then divided by a plane orthogonal to this axis. The intuition behind this approach is that the principle axis is typically a direction in which the grid is elongated, so a cut orthogonal to this direction will hopefully cut the grid at a narrow point. The inertial bisection algorithm can be implemented to run in linear time.

The second competing algorithm is spectral bisection. In this method a constrained quadratic minimization problem is formulated which corresponds approximately to finding a partitioning which retains balance and has a small number of cross edges. This minimization is cast in terms of the *Laplacian* matrix of the graph and is solved by an eigenvector of the Laplacian. The eigenvector defines a partitioning with the desired properties. More detailed descriptions of spectral bisection can be found in [10, 11, 13, 19, 21].

Both the inertial and spectral partitioning methods can be coupled to advantage with a local refinement scheme like Kernighan–Lin. These coupled methods provide the third and fourth alternatives we compare against. All of these methods are included in **Chaco 2.0** [12], making comparison convenient. A similar coupling of global and local methods is described in [25].

We have run these algorithms on a variety of graphs, and present results here for three that are typical of what we have generally observed. The first graph is a small two dimensional finite element mesh of space around an airfoil, which is due to Hammond and can be obtained from RIACS at NASA Ames via anonymous ftp [8]. The second graph, known as Barth5 is a similar but larger mesh, and can be obtained in the same way. The third graph is a fairly large finite difference grid of the Earth's oceans provided by Swisshelm [23].

We recursively divided each graph through six levels of bisection, producing 64 sets. Results of running the different partitioning algorithms on these grids are presented in the following tables. The run times shown on the last row are the times for the entire division into 64 sets. All calculations were performed on a Sun Sparc Station 20 with a 50 MHz clock and 64 Mbytes of memory.

Based on these and similar experiments, we make several observations:

- The local refinement scheme significantly improves the partitions generated by both the inertial and spectral partitioners and has modest cost. In fact application of KL may actually *reduce* net run time (see the spectral column of the ocean or hammond tables) because in the process of locally refining at

Table 1. Performance of partitioning algorithms on Hammond mesh. ($|V| = 4720$, $|E| = 13722$)

	EDGES CUT				
	Inertial		Spectral		Multi–Level
	Alone	With KL	Alone	With KL	
2 sets	209	117	117	97	119
4 sets	559	312	262	227	236
8 sets	880	486	475	388	393
16 sets	1342	727	769	668	652
32 sets	1791	1152	1219	1121	1065
64 sets	2392	1732	1885	1753	1688
Time (sec)	0.24	1.35	9.32	8.64	3.44

Table 2. Performance of partitioning algorithms on Barth5 mesh. ($|V| = 15606$, $|E| = 45878$)

	EDGES CUT				
	Inertial		Spectral		Multi–Level
	Alone	With KL	Alone	With KL	
2 sets	245	190	169	146	196
4 sets	897	425	527	413	412
8 sets	1441	878	866	699	648
16 sets	2267	1382	1432	1220	1118
32 sets	3138	1988	2163	1869	1779
64 sets	4257	3168	3273	2893	2906
Time (sec)	0.74	3.30	23.7	28.6	6.17

one recursion level, subgraphs are produced which are more efficiently handled at the next level of recursion, *i.e.* local refinement effectively preconditions the problem.

- The inertial approach is quite fast, but produces low quality partitions. However, when combined with a good local refinement technique, the combination often produces fairly good answers quite quickly. The partitions thus produced are generally better than those from spectral bisection; that is, the combination of simple global and local methods is usually superior to a sophisticated global method. However, the inertial algorithm must be used with caution since it sometimes produces significantly worse partitions, *eg.* bisection of the ocean graph.
- Spectral bisection generally produces better partitions than inertial bisection, albeit at a substantial increase in run time. This improvement remains after each method is coupled with a local refinement.
- Our multilevel method takes somewhat longer than inertial combined with KL, but it generally produces partitions similar in quality to those generated

Table 3. Performance of partitioning algorithms on ocean mesh. ($|V| = 143437$, $|E| = 409593$)

	EDGES CUT				
	Inertial		Spectral		Multi–Level
	Alone	With KL	Alone	With KL	
2 sets	2932	2576	660	533	499
4 sets	5607	4674	2392	2107	1991
8 sets	8695	7479	5264	4763	4743
16 sets	13240	11245	10204	9158	9160
32 sets	19292	16604	16348	14624	14628
64 sets	29230	24741	25547	22410	22343
Time (sec)	4.67	26.1	392.6	316.0	38.0

by spectral with KL.

A few comments are in order regarding the parameter choices used in generating these results. For the multilevel partitioner, we applied coarsening until the graph had at most 200 vertices and applied our local refinement strategy every third uncoarsening step. These choices are roughly in the middle of the recommended range [12], and were chosen to be representative.

In the case of the spectral methods, the calculation of an eigenvector of an $n \times n$ sparse, symmetric matrix is required. **Chaco 2.0** includes several different eigensolvers; their strengths and weaknesses are discussed in detail in [12]. For these test problems we used a multilevel eigensolver based on the algorithm described in [1] since that proved fastest[3]. This algorithm combines a graph coarsening strategy with Rayleigh Quotient Iteration (RQI) using the linear solver Symmlq to refine approximate eigenvectors projected from a coarse graph onto a finer graph. An important difference between the algorithm described in [1] and our version is that we coarsen the graph by edge contraction as described in §2. We coarsened until there were at most 200 vertices in the coarse graph, applied RQI/Symmlq every second uncoarsening step to refine the eigenvector and used an eigen residual tolerance of 10^{-3}.

5. Summary and conclusions. We have presented a multilevel algorithm for graph partitioning. The algorithm generates a sequence of smaller graphs that approximate the original graph, partitions the smallest graph in the sequence, and propagates the resulting partition back to the original graph while periodically performing a local refinement. This approach is very general and is applicable with a variety of different metrics on the partition quality.

Our implementation of this method demonstrates excellent performance on a variety of graphs typical of those encountered in parallel scientific computing. It found better

[3] On the Hammond graph our RQI/Symmlq algorithm was 1.2 times faster than Lanczos with selective orthogonalization when solving to the same residual tolerance. On the Barth graph RQI/Symmlq was 2.9 times faster. On the ocean graph Lanczos was unable to obtain a good answer due to memory limitations.

partitions than spectral bisection in a small fraction of the time. One shortcoming of the algorithm is that the construction of a sequence of graphs is memory intensive. A more aggressive coarsening strategy which generates fewer intermediate graphs would be one way of reducing this problem.

It is worth noting that a parallel implementation of the algorithm described here is problematic because Kernighan–Lin is known to be P-complete [20]. Furthermore, one attempt at a parallel implementation proved disappointing in performance [7]. However, there are local refinement techniques that do parallelize well, and they could easily take the place of Kernighan–Lin in our implementation [20, 9]. We are currently studying these and related issues [3].

Multilevel ideas have proved extremely powerful in continuous mathematics but have not yet made a large impact on discrete problems. We hope that our results will inspire further work on related algorithms for other combinatorial problems.

Acknowledgements. We would like to acknowledge John Lewis and Alex Pothen for encouragement and for independently suggesting an algorithm similar to the one we have described. We would also like to thank Horst Simon, Steve Barnard, Ray Tuminaro and David Womble for discussions regarding multilevel eigensolvers, Cindy Phillips for suggesting the use of edge contraction, and Thang Bui for very helpful discussions of his experiences with similar algorithms.

This work was supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research, and was performed at Sandia National Laboratories, operated for the U.S. Department of Energy under contract No. DE-AC04-76DP00789.

REFERENCES

[1] 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, SIAM, 1993, pp. 711–718.

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

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

[4] A. E. Dunlop and B. W. Kernighan, *A procedure for placement of standard–cell VLSI circuits*, IEEE Trans. CAD, CAD-4 (1985), pp. 92–98.

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

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

[7] J. R. Gilbert and E. Zmijewski, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, Intl. J. Parallel Programming, 16 (1987), pp. 498–513.

[8] S. Hammond. Personal Communication, November 1992. Available through anonymous ftp at riacs.edu, file /pub/grids/3elt.grid.

[9] ———, *Mapping unstructured grid computations to massively parallel computers*, PhD thesis, Rensselaer Polytechnic Institute, Dept. of Computer Science, Troy, NY, 1992.

[10] B. Hendrickson and R. Leland, *An improved spectral load balancing method*, in Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, 1993, pp. 953–961.

[11] ———, *Multidimensional spectral load balancing*, Tech. Rep. SAND93–0074, Sandia National Laboratories, Albuquerque, NM, January 1993.

[12] ———, *The Chaco user's guide, version 2.0*, Tech. Rep. SAND94–2692, Sandia National Laboratories, Albuquerque, NM, October 1994.

[13] ———, *An improved spectral graph partitioning algorithm for mapping parallel computations*, SIAM J. Sci. Comput., 16 (1995).

[14] B. Hendrickson, R. Leland, and R. Van Driessche, *Enhancing data locality by using terminal propagation*, in Proc. 29th Hawaii Conf. System Science, January 1996. To appear.

[15] C. A. Jones, *Vertex and Edge Partitions of Graphs*, PhD thesis, Penn State, Dept. Computer Science, State College, PA, 1992.

[16] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Rep. CORR 95–035, University of Minnesota, Dept. Computer Science, Minneapolis, MN, June 1995.

[17] B. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 29 (1970), pp. 291–307.

[18] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox, *Graph contraction and physical optimization methods: a quality–cost tradeoff for mapping data on parallel computers*, in Intl. Conf. Supercomputing, Tokyo, Japan, July 1993.

[19] A. Pothen, H. Simon, and K. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal., 11 (1990), pp. 430–452.

[20] J. E. Savage and M. G. Wloka, *Parallelism in graph–partitioning*, J. Par. Dist. Comput., 13 (1991), pp. 257–272.

[21] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, in Proc. Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications, Pergammon Press, 1991.

[22] P. Suaris and G. Kedem, *An algorithm for quadrisection and its application to standard cell placement*, IEEE Trans. Circuits and Systems, 35 (1988), pp. 294–303.

[23] J. Swisshelm. Personal Communication, February 1992.

[24] D. Vanderstraeten, C. Farhat, P. S. Chen, R. Keunings, and O. Zone, *A retrofit based methodology for the fast generation and optimization of mesh partitions: beyond the minimum interface size criteria*, Comput. Meths. Appl. Mech. Eng., (1995). To appear.

[25] D. Vanderstraeten, R. Keunings, and C. Farhat, *Optimization of mesh partitions and impact on parallel cfd*, in Parallel Computational Fluid Dynamics, New Trends and Advances, A. Ecer, J. Hauser, P. Leca, and J. Periaux, eds., Elsevier, 1995, pp. 233–239. (Also in Proc. Parallel CFD '93).

[26] C. Walshaw, M. Cross, and M. Everett, *A parallelisable algorithm for optimising unstructured mesh partitions*, Tech. Rep. 95/IM/03, University of Greenwich, London SE18 6PF, UK, 1995. (submitted for publication).

[27] R. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency, 3 (1991), pp. 457–481.