SAND95–2344
Unlimited Release
Printed July 1995

Distribution
Category UC–405

The Chaco User's Guide
Version 2.0

Bruce Hendrickson* and Robert Leland†
Sandia National Laboratories
Albuquerque, NM 87185–1110

Abstract

Graph partitioning is a fundamental problem in many scientific contexts. This document describes the capabilities and operation of **Chaco 2.0**, a software package designed to partition graphs. **Chaco 2.0** allows for recursive application of several methods for finding small edge separators in weighted graphs. These methods include inertial, spectral, Kernighan–Lin and multilevel methods in addition to several simpler strategies. Each of these approaches can be used to partition the graph into two, four or eight pieces at each level of recursion. In addition, the Kernighan–Lin method can be used to improve partitions generated by any of the other algorithms. Brief descriptions of these methods are provided, along with references to relevant literature. **Chaco 2.0** can also be used to address various graph sequencing problems, and this capability is briefly described. The user interface, input/output formats and appropriate settings for a variety of code parameters are discussed in detail, and some suggestions on algorithm selection are offered.

* Department 1422; email bah@cs.sandia.gov.
† Department 1424; email leland@cs.sandia.gov.

1. The quick version. If you're like us, there's no chance you'll read this full document before you start using **Chaco**[1]. So here are the basics. If you know a fair amount about graph partitioning and are experienced with computers, this should be enough to get you going. If you don't know what we're talking about, you probably need to grit your teeth and read the introduction (§2) and the section on methods (§3) before you go much further. You'll also have to read the section on input and output (§5) before you can progress beyond using the sample graphs we've provided and address your own problems. Once you're oriented you may want to return to this section, as it has some useful tips for the savvy user.

While **Chaco** has been used in many different settings, it was developed in a parallel computing context, and readers will notice a clear bias towards this application in the following documentation.

1.1. Overview. Many problems which arise in scientific computing have a combinatorial nature which can be conveniently described in the language of graphs. In these settings a recurring theme is the need to partition a graph into subgraphs that are in some measure as disjoint as possible. This is the case in, for example, divide–and–conquer algorithms for devising efficient circuit layouts or constructing nested dissection orderings for sparse matrix factorizations. It is also a fundamental problem in parallel computing, where large data structures must be decomposed and mapped to processors.

Broadly speaking, **Chaco** addresses three classes of problems. First and foremost, it partitions graphs using a variety of approaches with different properties. Discussion of these methods and the tools to control them comprise the bulk of this document. Second, **Chaco** knows how to intelligently embed the partitions it generates into several different topologies. The topologies the code knows about are those matching the common architectures of parallel machines, namely hypercubes and meshes (see §6.7). Third, **Chaco** can use spectral methods to sequence graphs in a manner that preserves locality, as described in §4.1. This capability has been used, for example, in data base organization, sparse matrix envelope reduction and DNA sequencing.

1.2. Obtaining the code. **Chaco** is available under license from Sandia National Laboratories. The source code is distributed along with technical documentation and sample input files via the internet. If you are interested in obtaining a copy, you should contact us at the addresses given on the cover page of this report.

1.3. Installing the code. **Chaco** is designed to be run on UNIX systems. To unpackage it, save the mailing in a file `Chaco.shar.Z.UU` and remove any mail header information.

[1] **Chaco** is named in honor of Chaco Canyon, the site of extensive Anasazi ruins in what is presently northwestern New Mexico. Between 1000 and 1100 AD a great society, considered the most complex and sophisticated on the continent north of Mexico, flourished there.

Then execute the following commands:

```
uudecode Chaco.shar.Z.UU
uncompress Chaco.shar.Z
chmod +x Chaco.shar
sh Chaco.shar
```

Assuming things have gone well, you may now delete the files "Chaco.shar.Z.UU" and "Chaco.shar", and follow the `README` file instructions to compile and run the code.

We have tried to make **Chaco** completely portable and we have compiled and run the code successfully on machines built by Sun, SGI, HP, IBM, DEC and Cray. If you are using an ANSI standard compiler, then **Chaco** should compile correctly, and it should do fine on many non–standard compilers as well. If you are having difficulties getting the code working on a new machine, we can suggest several possible sources of difficulty:

- **Chaco** uses several machine and compiler dependent parameters that are defined within the ANSI standard. If these values aren't defined, then **Chaco** tries to compute them, but this is difficult to do in a machine independent way. One thing the user can do to improve robustness with a non–standard compiler is to define appropriate values for three parameters in the file "code/util/machine_params.c". These parameters are `DBL_EPSILON` the machine precision, `DBL_MAX` a large double precision value, and `RAND_MAX` the largest value returned by the system random number generator 'rand'. You can examine the values the code computes for these parameters by turning on the `DEBUG_MACH_PARAMS` flag described in §6.9.
- The timing subroutine invokes a system routine called 'getrusage' which isn't supported by all compilers. We provide a second timing routine which is currently commented out in the code. You can replace the first timer routine with the second if necessary. The second routine uses a system routine that wraps around after about 36 minutes, which is why we prefer the first if it is available. Both routines are in the file "code/util/seconds.c".
- Compiler flags vary greatly from machine to machine. You may need to modify the compilation command in "code/Makefile" to make **Chaco** compile and link properly.
- **Chaco** makes extensive use of the random number generator 'rand' which is defined in ANSI C. If your compiler doesn't have this routine, you'll need to provide a random number generator that produces integers between 0 and *max*. You should then modify "code/util/randomize.c" to make the appropriate new call, and "code/util/machine_params.c" to return *max* in its third argument.

1.4. Some more things to watch out for.

- The routine "func3d.c" takes a long time to compile with optimization. It doesn't account for a significant fraction of the execution time, so if, for some reason, you are recompiling the code often, you may wish to compile this routine without optimization.

- Use of the Lanczos–based eigensolvers on very large problems may cause the code to run out of memory on your system. The code will recover by computing the best approximation it can given the available memory, but if this happens it may be advisable to switch to the RQI/Symmlq eigensolver or the inertial or multilevel–KL partitioning methods. See §3.3, §3.2 and §3.6.
- It can be difficult to choose the eigentolerance for spectral methods appropriately. We've chosen a reasonable default, and **Chaco** tries hard to deliver the accuracy requested, but can't help much if that request is unwise. If you choose a very tight (small) tolerance, things will slow down considerably and you may run into memory trouble. If you choose a very loose (big) tolerance, your results will generally degrade and become erratic due to poor accuracy or misconvergence. See §3.3 and §6.2.
- The eigensolvers and the Kernighan–Lin heuristic make use of randomization techniques, so results generated using these methods are strictly reproducible only if the program is used in a way that generates the same sequence of random numbers.
- If you apply terminal propagation with spectral partitioning, several tricky precedence relations between the eigensolvers and partitioning dimensionality necessarily come into play. Refer to §4.2.

1.5. Implementation details.

- Version 2.0 of **Chaco** is written entirely in ANSI standard C and is about 30,000 lines long.
- In order to maximize the size of graphs which can be partitioned, memory is allocated dynamically when needed and released as soon as possible without seriously degrading efficiency.
- C performs floating point computations in double precision (8 byte) format, and **Chaco** stores the results in double precision format (except in a few cases where precision is clearly not an issue).
- **Chaco** can be run in a stand–alone mode or called as a subroutine from either C or Fortran programs as described in §7.

1.6. Partitioning. The five classes of partitioning algorithms currently implemented in **Chaco** are simple (§3.1), inertial (§3.2), spectral (§3.3), Kernighan–Lin (KL) §3.5, and multilevel–KL (§3.6). Each of these algorithms can work on graphs with edge and/or vertex weights and each can be used to partition into two, four or eight sets at each stage of a recursive decomposition. We consider KL to be a local refinement technique, while the other methods are global partitioning methods. **Chaco** allows the output of any of the global methods to be fed into a local method. It also allows a partition to be read from a file (§4.4) and refined with a local method or one of the various post–processing methods described in §4.3.

You can combine local and global partitioning methods by choosing from the menu in an obvious way. We encourage you to experiment with the sample graphs provided with the code.

In addition to the basic partitioning algorithms, **Chaco** includes a host of more sophisticated capabilities. Several of the methods can be invoked with a technique known as *terminal propagation* §4.2 which improves data locality by allowing consideration of how the sets are mapped to processors. These include KL, multilevel-KL and spectral (in bisection mode only). Another way to improve the mapping to processors is to invoke a post–processing algorithm devised specifically for this purpose which is discussed in §4.3.3. The partition itself can also be improved with a post–processing phase that applies KL to all pairs of sets with edges between them (§4.3.1). And, in some parallel computing settings it may be possible to overlap communication with the computation associated with vertices that need no external information; **Chaco** has the ability to increase the number of these internal vertices (§4.3.2). In addition, **Chaco** can be used to compute and sort the Fiedler vector of a graph, which is useful in many settings in which data locality is desirable.

1.7. Input and output. Input to **Chaco** consists of one or more files and the answers to several interactive queries. The format of the input file describing the graph can be found in §5.1, and examples are provided with the code. If you select inertial partitioning you will also need to provide a file with geometric coordinates as described in §5.2.

Output from the code includes a variety of metrics of partition quality. The detail with which these metrics are reported is controlled by the `OUTPUT_METRICS` parameter (§6.1). This information can be copied to a file by setting the parameter `ECHO` (§6.1) appropriately. The partition will be copied to a file if the `OUTPUT_ASSIGN` parameter is set to `TRUE` (nonzero) (§6.1).

It may be clear by now that much of the functionality in **Chaco** is controlled by a fairly large set of parameters. We ship the code with default values that seem reasonable to us, but may not be optimal for your problems. You can either change the default values in the file "code/main/user_params.c" and recompile, or you can change any value at runtime, as described in §6.10.

2. Introduction. Many problems which arise in the course of scientific computing have a combinatorial nature which can be conveniently described in the language of graphs. In these settings a recurring theme is the need to partition a graph into subgraphs that are in some measure as disjoint as possible. This is the case in, for example, divide–and–conquer algorithms for devising efficient circuit layouts or constructing nested dissection orderings for sparse matrix factorizations. It is also a fundamental problem in parallel computing, where large data structures must be decomposed and mapped to processors.

Chaco addresses three classes of problems. First and foremost, it partitions graphs using a variety of approaches with different properties. Discussion of these methods and tools to control them comprise the bulk of this document. Second, **Chaco** knows how to intelligently embed the partitions it generates into several different topologies. The topologies the code knows about are those matching the common architectures of parallel machines, namely hypercubes and meshes (refer to §6.7). Third, **Chaco** can use spectral methods to sequence graphs in a manner that preserves locality, see §4.1. This capability has been used in, for example, data base organization, sparse matrix envelope reduction and DNA sequencing.

To make things more specific, let's assume we want to solve a partial differential equation on a distributed memory parallel computer[2]. We're given a computational grid which we need to partition across processors. If we're using a finite difference technique and an explicit solver, then at each stage in the calculation a grid value must be updated by a function of its neighbor's values. On a serial computer this data transfer is accomplished by writing to and reading from memory. However, when we map this computational grid to a parallel computer, two vertices joined by an edge and not owned by the same processor must communicate to exchange values. If, as is typically the case, communication is expensive relative to computation, a mapping that minimizes it is desirable. Of course, we could assign the entire grid to a single processor and have no communication at all, but that wouldn't be an effective use of the parallel machine since one processor would do all the work while the others remained idle. We must therefore also observe the important constraint that each processor should be assigned about the same amount of work and therefore (in the simplest case) the same number of vertices. Hence we say informally that the objective of **Chaco** in this context is to produce *balanced* sets with low communication overhead.

Not all problems have such a convenient correspondence between the computational grid and the mapping requirements of the application program. For instance in a finite element calculation, a more appropriate approach may be to consider each element as a vertex with some associated update work. We would then construct connecting edges corresponding to each face or corner in the discretization mesh since these edges correspond to the non–zero pattern in the global stiffness matrix. The most appropriate graph will depend upon the application and its determination is necessarily left to the user.

[2] While **Chaco** has been used in many different settings, it was developed in a parallel computing context, and readers will notice a clear bias towards this application in the following documentation.

Furthermore, all vertices are not necessarily of equal significance. For example, a vertex encoding a computation on the boundary may have less work associated with it than a vertex in the interior of a domain. **Chaco** therefore allows weights to be associated with each vertex. The weight is supposed to correspond to the amount of work associated with the vertex. Similarly, edges may correspond to varying amounts of communication. For example, two finite elements touching at a corner may need to exchange less information than two sharing a face. **Chaco** also allows the use of edge weights.

The problem of interest can now be described more precisely. Given a graph G with n weighted vertices and m weighted edges, divide the vertices into p sets in such a way that the sum of the vertex weights in each set is as close as possible, and the sum of the weights of edges crossing between sets is minimized. Unfortunately, even in the simple case where $p = 2$ and the edge and vertex weights are uniform, this graph partitioning problem is NP–complete[7]. Hence there is no known efficient algorithm to solve the problem generally, and it seems unlikely that such an algorithm exists. We must therefore resort to heuristic solutions in which balance may be partially compromised or (more typically) the minimization is approximate.

A variety of heuristic partitioning methods with different cost/quality tradeoffs have been previously studied. **Chaco** includes methods based on several of these as well as several substantially new methods. The algorithms in **Chaco** are based on inertial, spectral, Kernighan–Lin (KL), and multilevel principles in addition to several simpler strategies. The methods are categorized as either local (currently just KL) or global (everything else). **Chaco** allows for the combination of global and local methods, and we have found that this leads to significant improvements in both performance and robustness. Another advantage of **Chaco**'s design philosophy is that it offers flexibility. This is important because we believe that, given the complexity of the partitioning problem, *no single method will always work well*. **Chaco** provides a fall–back option when your favorite method works poorly or has an inappropriate cost/quality ratio for a given problem. It also facilitates investigation into the relative strengths and weakness of a wide variety of methods.

Having set the basic context, we should raise some finer but nevertheless important issues. One such issue is the *dimensionality* of the partitioning scheme. Most graph partitioning codes rely on recursive bisection. That is, the graph is partitioned into two pieces, each of these pieces is partitioned into two more, etc. until a desired number of sets is reached. This strategy is simple and convenient, but may be somewhat limiting. Graphs can be constructed for which any bisection algorithm must necessarily perform poorly, and in practice we observe that bisection algorithms often choose separators which look very good at one stage of recursion but not so good with the benefit of hindsight. All the partitioning algorithms implemented in **Chaco** are capable of partitioning graphs into two, four or eight sets at each stage of recursion[3]. We have accumulated some empirical evidence that the quadrisection and octasection algorithms do perform better in some respects than their bisection counterparts. But we have also

[3] Currently the spectral terminal propagation technique can be applied in bisection mode only.

found bisection algorithms preferable in some situations.

A basic difficulty in choosing the appropriate partitioning dimensionality is that the correct representation of costs in the graph model is often ambiguous. Assuming for simplicity that the graph is unweighted, most graph partitioning schemes work to suppress the total number of edges crossing between sets without regard to the identity of the sets. We say these methods try to minimize the total number of *cuts*. But in contexts like parallel computing and circuit placement, the identity of the sets matters. The partitions may need to be mapped to processors or regions of an integrated circuit in a manner that minimizes the number of connections between architecturally distant sets. Several of the multidimensional schemes we have developed can take into account the identity of the sets an edge crosses between and work to minimize the architectural distance between these sets. We say they try to minimize the total number of *hops*.

Sadly, the question of the correct graph metric is more complicated still. For example, in the parallel computing context, when the communicated messages are short enough, the total communication time will correlate best with message *startups*. In the graph metric this measure corresponds to the number of neighboring sets each set has. We have also included methods designed to deal with this contingency by suppressing the maximum number of neighbors any set has. Another graph metric which is important in some common situations is *boundary vertices*. This is the number of vertices which have an incident edge (they may have several) connecting them to a vertex in another set. When these are weighted by the architectural distance between sets we arrive at yet another metric, *boundary vertex hops*. These last two metrics are often relevant in accurate modeling of the execution time of parallel sparse matrix–vector multiplication.

Because applications of graph partitioning are so diverse and because even for the much studied case of parallel computing the appropriate model is uncertain, **Chaco** tracks a variety of potentially relevant metrics and provides methods designed to minimize them. This document describes the capabilities of the code and how to exploit them. Because the questions it addresses are fundamental and pervasive, we hope that **Chaco** will prove to be a valuable tool in a wide variety of applications.

3. Partitioning algorithms. The five classes of partitioning algorithms currently implemented in **Chaco** are simple, spectral, inertial, Kernighan–Lin (KL)[4] and multilevel–KL. Each of these algorithms can be used to partition into two, four or eight sets at each stage of a recursive decomposition. We consider KL to be a local refinement technique, while the other methods are global partitioning methods. **Chaco** allows the output of any of the global methods to be fed into a local method. It also allows a partition to be read from a file (§4.4) and refined with a local method or one of the various post-processing methods described in §4.3.

3.1. Simple partitioning methods. For completeness and in order to facilitate comparisons, **Chaco** includes three very simple partitioning schemes. In the *linear*

[4] This algorithm is often referred to as Fiduccia–Mattheyses (FM) or KL/FM in recognition of the important contributions of those authors.

scheme, vertices are assigned in order to processors in accord with their numbering in the original graph. For an unweighted graph with n vertices being divided into p sets, the first n/p vertices would be assigned to set 0, the next n/p to set 1, etc. This often produces surprisingly good results because data locality is often implicit in the vertex numbering. In the *random* scheme, vertices are assigned randomly to sets in a way that preserves balance. In the *scattered* method, vertices are handed out in order, with the next vertex going to whichever set is smallest. In the unweighted case this reduces to dealing out the vertices in card fashion. In our experience the random ordering produces partitions with quality between that of the linear and scattered partitioners. The run time of these simple schemes is negligible.

3.2. The inertial method. The *inertial* method is a relatively simple and fast partitioner that uses geometric information. In addition to a graph, it requires geometric coordinates for each vertex in one, two or three dimensions. The code then considers the vertices as point masses with mass values set equal to the vertex weights. The principle axis of this structure, which is likely to be a direction in which the graph is elongated, is computed. The vertices are then divided into sets of equal mass by plane(s) orthogonal to the principle axis. Descriptions of this method can be found in [24, 28].

Chaco allows inertial partitioning into two, four or eight sets at once by using one, three or seven planes, orthogonal to the principle axis. Partitions generated by inertial quadrisection or octasection will appear to be banded, with parallel planes dividing the sets. This "striping" will typically lead to a fairly large surface-to-volume ratio, implying a large volume of communication. However, each set only has a small number of neighboring sets which helps reduce the number of message startups each processor must perform. If the cost of initiating messages is important, then partitions using inertial quadrisection or octasection may lead to shorter application execution times than those generated with inertial bisection. Furthermore, the multidimensional inertial methods are somewhat faster than inertial bisection since fewer inertial axes must be computed, and some overhead due to recursion is avoided. The four or eight sets are assigned in such a way that communication is predominantly between adjacent processors.

In our experience inertial methods are quite fast but give partitions of fairly low quality in comparison with spectral methods. In particular, the partitions are often of poor quality in local detail. However, when coupled with the Kernighan–Lin local optimization method described below, the results significantly improve. Our experiments indicate that inertial–KL usually produces better partitions than pure spectral partitioning, whereas spectral coupled with KL does better than inertial paired with KL. For very large problems in which coordinates are available and the emphasis is more on low partitioning time rather than high partitioning quality, we are inclined to recommend the inertial–KL method [18].

3.3. Spectral partitioning. Spectral methods use eigenvectors of a matrix constructed from the graph to decide how to partition the graph. A full accounting of this surprising connection between eigenvectors and partitions is too involved to present here, but the articles mentioned below explain the method in detail.

The simplest spectral method in **Chaco** is a weighted version of *spectral bisection*. A description of the unweighted algorithm is given in [22, 24], and the extension to use both edge and vertex weights is described in [12]. This method uses the second lowest eigenvector of the *Laplacian* matrix of the graph to divide the graph into two pieces. This eigenvector is known as the *Fiedler* vector in recognition of the pioneering work of Miroslav Fiedler [5, 6].

The *spectral quadrisection* algorithm divides a graph into four pieces at once using the second and third lowest eigenvectors of the Laplacian matrix. Similarly, *spectral octasection* uses the second, third and fourth eigenvectors to divide into eight pieces. These *multidimensional* spectral methods were introduced in [11, 12], where they were shown to have certain advantages over spectral bisection.

In particular, spectral quadrisection and octasection try to minimize communication cost in a more complex metric. Suppose the partitioned sets are numbered from 0 to 3 for quadrisection or 0 to 7 for octasection. Spectral bisection would try to minimize the total weight of edges crossing between different sets, whereas the multidimensional methods would use a metric in which the cost of an edge crossing between two sets is the edge weight multiplied by the number of bits that are different in a binary representation of the two sets.

Although this *hops* metric may seem odd at first, it has a nice interpretation in the context of parallel computing. In a parallel computer consisting of four processors connected in a square and numbered in typographic order, a message traveling between processors 0 and 3 must travel over two wires, whereas one between processors 0 and 1 need only traverse a single wire. This number of wires is exactly the weighting implicit in spectral quadrisection. Similarly, spectral octasection counts wires used on a three-dimensional mesh architectures, and both quadrisection and octasection apply to hypercubes.

One might suppose that this correspondence between cost metric and wires used was irrelevant given the advent of cut–through routing in which the delay associated with a message is nearly independent of the number of links it traverses. In fact this independence only holds for isolated messages in which there is no competition for the links in the communication network. In a great many computations, and most scientific applications, communication occurs in the form of bursts of messages during which there is very significant competition for the network. Hence, when network congestion is important, weighting messages by the number of wires they consume should lead to better problem mappings. Empirical evidence supporting this and further discussion of the issue can be found in [9].

The computational kernel of spectral methods is the calculation of a small number of eigenvectors. We have implemented a variety of eigen solvers with different speed/robustness tradeoffs. Roughly in order of increasing speed, these are Lanczos with full orthogonalization, Lanczos with selective orthogonalization, and a multilevel method combining Rayleigh Quotient Iteration [8] and the linear solver Symmlq [19]. We have also implemented a specialized version of Lanczos capable of solving *extended* eigen problems of the form $Au = \lambda u + g$ which arise when terminal propagation is used (§4.2). Several of the issues governing the choice between these methods are dealt with in the next section. Section 3.4 can be skipped by the typical user, who will simply encounter a choice between the default Lanczos procedure (selective orthogonalization) which is designed for small and medium sized graphs and the RQI/Symmlq method which is designed for larger graphs (of say more than several thousand vertices).

Spectral methods are usually quite good at finding the right general area of the graph in which to cut. However, they often do poorly in the fine details. Consequently, we have found that it is advantageous to apply a local refinement to the spectral output. The procedure we use is a generalized version of an algorithm due to Kernighan and Lin, and is described in §3.5 and in more detail in [13]. The actual improvement due to this clean–up phase is problem dependent, but is typically 10–30%. The cost of this clean–up is generally a small fraction of the total partitioning cost, typically less than 10% on large graphs.

3.4. Choosing an eigen solver. This section, which may be skipped without loss of continuity, describes the characteristics of the eigen solvers in **Chaco**. The input menu will indicate a choice between two methods only, a Lanczos based solver and the multilevel RQI/Symmlq solver. We recommend the Lanczos method for small and medium size problems and the RQI/Symmlq solver for larger problems. (We say, rather arbitrarily, that larger graphs are those of order 10,000 vertices or more; you should investigate this for yourself if run time is very critical.) There may be occasions, however, when the sophisticated user will want to change the type of Lanczos algorithm by modifying the `LANCZOS_TYPE` flag or may wish to alter one of the eigen solver control parameters. See §6 for details on how to make these changes.

Finally, while we do express some clear opinions in what follows, it should be carefully noted that our conclusions about the relative merits of the different eigen solvers are based on limited testing with the particular class of matrices arising in our applications, and may not be applicable to any other domain. *These are all iterative methods!*

In our experience, full orthogonalization Lanczos is the most robust method for problems of order up to a few hundred. The requirement of saving all the Lanczos vectors for orthogonalization is not that burdensome since the problems are small and we use them anyway in assembling the eigenvectors. The weak point of this method is that for larger problems the orthogonalization work becomes prohibitively expensive.

The inverse operator full orthogonalization Lanczos method replaces the matrix vector multiply in the basic Lanczos iteration with a linear solve using Symmlq. It is generally less accurate and robust than direct Lanczos with full orthogonalization and is often slower as well because the total number of matrix vector multiplies (which are hidden within Symmlq) may be significantly higher. In addition it introduces the tricky problem of how to tune the inner/outer loop combination. Thus the only reason to recommend this method is that it requires much less memory since it converges in many fewer Lanczos iterations.

Our implementation of selective orthogonalization is based on the original paper by

Parlett and Scott [20], with the main differences being that the Ritz spectrum is monitored directly to assess the need for orthogonalization and that this orthogonalization is performed against the left end of the spectrum only. Various heuristics governing which Ritz pairs to monitor are used to keep this overhead small. The Ritz pairs are computed using the classic bisection algorithm on the Sturm sequence [27] or the standard QL algorithm for tridiagonal matrices [8, 23], whichever is expected to be cheaper based on a simple complexity model. There are rare circumstances under which each of these algorithms can fail, so the code monitors for these and switches to the other algorithm if a problem is detected. Orthogonalizing at the left end only generally produces more accurate eigen pairs in substantially less time than the standard technique of orthogonalizing against both ends of the spectrum. With proper tuning this algorithm seems, for our purposes, essentially as accurate as full orthogonalization and is our method of choice for small and medium sized systems.

This version of Lanczos does however have one drawback. Since all the Lanczos vectors must be saved for the contingency that the iterate must be orthogonalized against a convergent Ritz vector, *this method can cause the program to run out of memory on very large problems.* This difficulty can be avoided by employing a restarting scheme or by giving up on maintaining orthogonality in the Lanczos basis. These alternatives, however, have their own undesirable attributes. Restarting schemes exhibit slower convergence, and schemes such as [21] which do not orthogonalize and hence do not need to save the Lanczos vectors must run through the entire Lanczos recurrence a second time (or use inverse iteration) in order to compute the desired eigenvector. Furthermore, if the desired eigenvector is not the first to converge significantly, non-orthogonalizing schemes may fail badly. Convergence usually *is* led by the Fiedler vector in the spectral bisection application, but there is no guarantee of this. So for robustness, and because we often need to compute higher eigenvectors to perform quadrisection or octasection, we chose selective orthogonalization. If memory is exhausted, each Lanczos routine computes the best available approximation to the required eigenvectors using the existing Lanczos basis. This approach represents a decision to optimize over the likely range of application and an assumption that for problems in which memory would be a problem a partitioning method designed for larger problems (*e.g.* the RQI/Symmlq method) will be employed.

For partitioning very large graphs using the spectral method, we recommend the multilevel RQI/Symmlq eigen solver. This is based on the method developed by Barnard and Simon [1], with the main difference being that we have used an edge contraction coarsening scheme described in [13]. This contraction scheme preserves the low modes of the operator sufficiently well that we need only perform RQI refinement periodically as we work back through the grid hierarchy. We have also modified the Symmlq iteration to terminate when the norm of the iterate reaches a preset limit. We do this because RQI relies essentially on inverse iteration in which a large iterate indicates convergence. The resulting method may be several times faster than Lanczos with selective orthogonalization for solving large problems to the same accuracy, and also requires far less memory. A drawback is that the method seems more prone to misconvergence than Lanczos. Experience indicates, however, that for large graphs, eigenvectors other than the Fiedler vector usually give partitions of similar quality to those generated with the Fiedler vector (occasionally better!). So slight misconvergence is not that serious a problem, especially if you are applying a local refinement method. Another drawback of the RQI/Symmlq algorithm is that its run time is essentially proportional to the number of eigenvectors solved for. This erodes its speed advantage when used as the eigen solver for one of the multidimensional spectral partitioning schemes.

When the terminal propagation method is applied (§4.2), the solution vector u of the extended eigen problem $Au = \lambda u + g$ must be computed. We have developed a variant of Lanczos for this which follows that of Van Driessche and Roose [26]. The main difference is that we have incorporated selective orthogonalization and some (but not all) of the safety features previously described.

A critical issue in the proper use of iterative eigen solvers is the choice of the tolerance on the eigen residual. This is treated in some detail later during the discussion of the various code parameters in §6.2, but it is appropriate to mention here that all of the eigen solvers have direct residual checks to determine whether the requested eigen tolerance has been achieved. In addition, the selective orthogonalization schemes have safety checks to monitor the effectiveness of the orthogonalization, and the multilevel RQI/Symmlq code incorporates a heuristic to detect misconvergence. From time to time and depending upon how the error and warning condition flags are set, one or more of these conditions will be noted by **Chaco**. In most cases these are not show-stoppers: the desired safety standards have not been met, but the computation will proceed and generate reasonable partitions. If certain error or warning conditions occur chronically, you may need to choose different tuning parameters. (Or, of course, there may be a problem with the code.)

3.5. Kernighan–Lin. One of the most popular methods for partitioning graphs dates back to work done in the early 70's by Kernighan and Lin [17]. Various extensions and improvements of the original idea have been proposed through the years, including the important linear time implementation due to Fiduccia and Mattheyses [4], who are often jointly credited with the algorithm. At its heart, Kernighan–Lin (KL) is simply a greedy, local optimization strategy. Vertices are moved between sets in an effort to reduce the number of edges cut by the partition. Although the original algorithm was designed for graph bisection, Suaris and Kedem [25] showed how to extend it to the quadrisection case. We have generalized this idea so that our code works on an arbitrary number of sets at once, and also works with edge and vertex weights [13]. Unfortunately, the runtime of the algorithm and its memory requirements increase with the partitioning dimension, so in practice we use only bisection, quadrisection and octasection to match the other methods in **Chaco**.

In our experience KL does not find very good partitions of large graphs unless it is given a good initial partition. Hence we find its value to be greatest when used in conjunction with one of the global partitioners. If you are interested in verifying this by testing KL essentially on its own, we recommend that you invoke the simple random method to provide an initial partition.

Typically, **Chaco** tries to generate partitions which are as balanced as possible. In some applications, it is preferable to allow a bit of imbalance if the edges crossing between sets can be reduced. **Chaco** allows KL (and Multilevel–KL described below) to look for unbalanced partitions. If this functionality is of interest to you, you should set the `KL_IMBALANCE` parameter described in §6.4 to something larger than its default of zero.

3.6. Multilevel–KL. Our method of choice for large problems in which high quality partitions are sought is the multilevel–KL[5] algorithm described in [13]. This method is very similar in approach to the method of Bui and Jones described in [2, 15]. It works by creating a sequence of increasingly smaller graphs approximating the original graph, partitioning the smallest graph, and projecting this partition back through the intermediate levels. Kernighan–Lin is invoked every few levels of projection to refine the partition. We use a spectral method to partition the smallest graph, but this does not seem to be critical.

The algorithm for constructing smaller approximations to the graph relies upon finding a maximal matching in the graph, and then contracting edges in the matching. This generates a new graph with typically about half as many vertices as the original graph. Edge contraction is intuitively attractive because it largely preserves the graph topology. When edges are contracted, a single vertex is created out of the two endpoints with weight given by the sum of the weights of the endpoints. In addition, any edges which become coincident have their weights summed and become a single edge. These operations have the effect of preserving the essential properties of a partition as it is moved between graphs in the hierarchy. The number of vertices in the smallest graph is an input option (we typically use a value between 50 and 500), and the frequency with which to invoke KL is controlled by the `COARSE_NLEVEL_KL` and `COARSE_KL_BOTTOM` parameters described in §6.5.

The method of Bui and Jones does not use edge and vertex weights, but is otherwise equivalent to ours. **Chaco** allows the user to turn off edge and/or vertex weights in the coarsening process by setting the `COARSEN_EWGTS` and/or `COARSEN_VWGTS` parameters to `FALSE` as discussed in §6.5. This allows for application of Bui and Jones' method as well as algorithms intermediate between ours and theirs. In our experience, the difference between the methods is small with neither method being consistently superior.

Our experience indicates that the multilevel–KL method gives very high quality answers in moderate time. It is not as quick as the inertial method plus KL, but it generally produces better partitions. In most cases it produces partitions which are

[5] Some confusion has arisen in the past regarding the naming of this algorithm. We have referred to it in writing as the *multilevel–FM* algorithm because our implementation of the Kernighan–Lin algorithm is based on that advocated by Fiduccia and Mattheyses [4]. We have also referred to it simply as the *multilevel* algorithm because we believe the power of the algorithm derives essentially from its strategy of applying local refinement on multiple scales and that refinement schemes other than the one we have chosen would also work well. Finally, there has been confusion regarding the algorithm's relationship to the multilevel RQI/Symmlq algorithm used for computing the eigenvector(s) needed in spectral partitioning methods. These are entirely different partitioning algorithms, although they do happen to share the same graph coarsening scheme in our implementation.

better than those generated by spectral coupled with KL and runs significantly faster than any of the spectral methods. More on the workings and performance of this multilevel–KL method can be found in [13].

4. Additional functionality.

4.1. Spectral sequencing. Spectral graph algorithms are becoming increasingly popular for a variety of applications. Often the key computation in these algorithms is the generation of the Fiedler vector. Unfortunately, calculation of eigenvectors of large matrices can be difficult, and the scarcity of robust, efficient tools well tuned for these graph applications has impeded development of spectral graph algorithms. To address this problem we provide easy access to the Fiedler vector computed by **Chaco**. Although a full exposition on this topic is beyond the scope of this user's guide, the ordering of vertices produced by their values in the Fiedler vector has some nice properties. In particular, vertices connected by an edge will tend to be assigned numbers that are close to each other. This property has already been successfully exploited in a number of applications including chromosomal mapping, matrix reordering and database organization applications. We expect many more uses will be found.

If the `SEQUENCE` parameter described in §6.8 is `TRUE` (or nonzero), the Fiedler vector will be sorted and written to the file whose name is specified by the parameter `SEQ_FILENAME`. These parameters set up an alternate execution path that doesn't perform any partitioning. The code uses whichever eigensolver would be used by a spectral partitioning algorithm. That is, if you select a spectral method and the RQI/Symmlq eigensolver, that will be used; otherwise, the Lanczos solver specified by the `LANCZOS_TYPE` parameter (§6.2) will be used.

Since spectral methods break down if the graph is disconnected, the spectral sequencing code works on the connected components of the graph in turn. The perturbation of the matrix associated with the `PERTURB` parameter in §6.3 is unnecessary and hence is disabled. The code sorts the vertices in each connected component by their value in the Fiedler vector and prints them in sorted order. Each line in the output file contains the vertex number followed by its value in the Fiedler vector. A change to a new connected component is signaled by a switch from a positive value for the Fiedler component to a negative one, since values for each component must be nondecreasing. If a connected component consists of a single isolated vertex, this vertex is assigned a value 0 in the returned vector.

4.2. Terminal propagation to improve the mapping. *Terminal propagation* is an algorithmic insight proposed by Dunlop and Kernighan [3] to improve the placement of circuit elements on a chip by adding additional constraints to a Kernighan–Lin algorithm. Van Driessche and Roose [26] have recently shown that these same constraints can be encoded into a spectral method, significantly extending the applicability of the original idea.

Terminal propagation isn't a new partitioning method but rather a modification of some of the methods discussed above. It is essentially a method for coupling the mapping to sets with the partitioning in an effort to improve locality. In the circuit context it is undesirable to have long wires criss–crossing the chip since they use up valuable space. In parallel computing, messages traveling between architecturally distant processors should be minimized since they tie up many communication links. Terminal propagation allows these considerations to be factored into the partitioning.

To understand how terminal propagation works, first consider partitioning without terminal propagation. After each step in a recursive decomposition the pieces are decoupled and interact no further. An edge crossing between two sets does not affect the later partitioning of either set. Consequently, there is nothing preventing the two adjacent vertices from being assigned to sets that are quite far from each other.

Terminal propagation ameliorates this by including information about the outgoing edges (or terminals) in the recursive partitioning. Details about how this is accomplished are given in references [3, 26, 14]. **Chaco** includes code for terminal propagation in the bisection mode of the spectral partitioner, and for an arbitrary number of sets for KL and multilevel–KL. (If you are using multilevel–KL in quadrisection or octasection mode, the spectral method at the bottom cannot perform terminal propagation, but all the invocations of KL can.) Terminal propagation is switched on by setting the `TERM_PROP` parameter to `TRUE` (or nonzero) as described in §6.8. We also note that terminal propagation comes into play only when there are edges to other sets, so it has no effect on the first step of bisection. If the quality of the mapping to sets is important for your application, you should also consider the post–processing method described in §4.3.3.

When terminal propagation is applied, the necessary modifications to KL (and hence multilevel–KL) are fairly minor, but the spectral formulation is significantly complicated. We must solve an *extended* eigenproblem of form $Au = \lambda u + g$ for u such that $u^T u = \sigma^2$ where σ is a constant. We could do this conceptually by choosing a λ near the corresponding eigenvalue of A, solving the resulting linear system and checking the norm constraint. By adjusting λ correctly and iterating we can converge to the solution (λ, u) reliably in the bisection case. This, however, is unacceptably expensive for large systems. The trick is to transform the λ iteration into Lanczos space where it is performed on small tridiagonal systems. A detailed formulation of this for edge and vertex weighted graphs is presented in [14, 26]. We have extended this to include selective orthogonalization in our implementation.

In our experience, the terminal propagation variant of the multilevel–KL algorithm consistently improves mappings, while the spectral algorithm seems less consistent.

4.3. Post–processing to improve the partition and mapping. **Chaco** includes several techniques that accept an existing decomposition and modify the partition or the mapping of sets to processors. These can be used to improve the quality of output generated either by **Chaco** or, if you read a partition from a file as described in §4.4, by other partitioning software.

4.3.1. Refining the partition. In the recursive generation of a decomposition, some information is lost with each recursion level. For example, a local refinement is performed between only a fraction of the total number of adjacent sets. If requested,

Chaco can perform a local refinement between *all* pairs of sets. First the weight of edges crossing between each pair of sets is determined. Kernighan–Lin refinement is then performed between each pair with a nonzero boundary, in order from the pair with the largest boundary to that with the smallest. Terminal propagation may be used to incorporate considerations of the quality of the mapping to sets. The parameter `REFINE_PARTITION` (§6.6) indicates how many cycles of refinement will be performed. Its default value is zero since a full refinement is fairly expensive. In our experience this option can significantly reduce the number of edges cut in the partition, but it generally increases the number of pairs of sets with some boundary between them.

4.3.2. Increasing the number of internal vertices. In some applications it is desirable to increase the number of vertices that have no edges connecting them to other sets. For instance, in parallel computing applications such vertices require only local data. This may allow for overlap of communication and computation since the computation associated with an internal vertex can be performed while waiting for data from other processors to arrive. If the `INTERNAL_VERTICES` parameter (§6.6) is `TRUE`, **Chaco** will try to increase the number of internal vertices in sets with a small number of them. To accomplish this the code first determines the number of internal vertices in each set. Then the set with the fewest internal vertices steals vertices from other sets to make some of its own vertices become internal, and trades back other vertices to preserve balance. The default value for `INTERNAL_VERTICES` is `FALSE` since this fairly specialized functionality is probably not required for most applications. If the partition is of low quality this option can be quite time consuming.

4.3.3. Improving the mapping to processors. Although **Chaco** tries to assign sets to processors in a way that preserves locality, this mapping can often be improved. **Chaco** contains code to greedily renumber sets to improve the mappings to hypercube and mesh architectures. Note that this doesn't change the composition of the sets, just which processor each set is assigned to. To perform this refinement, the code determines how the mapping would change if it flipped the two sets connected by a wire in the parallel machine. The flip which maximally improves the mapping is performed and the process repeated until no further improvement is possible. This functionality is activated by setting the `REFINE_MAP` parameter (§6.6) to `TRUE`. Since **Chaco** is used for many applications other than parallel computing, the default for this parameter is `FALSE`. But if you are really interested in the quality of your assignment to processors, you should try this option. You should also familiarize yourself with terminal propagation as described in §4.2.

4.4. Working with existing partitions. As mentioned in §3, **Chaco** has the ability to read an existing partition from a file with one of the formats described in §5.3 and modify or evaluate it in several ways. This option is specified in the menu as an additional global partitioning option. Any of the post–processing operations described in §4.3 can then be activated to improve the partition and/or the mapping to processors. Evaluation of the partition can also be performed as described in §5.5.

Since an existing partition is considered a global partition, you can invoke KL as a local refinement. There are a few necessary restrictions on the use of this capability. You can use KL only if the existing partition has 2, 4 or 8 sets, and you request bisection, quadrisection or octasection respectively. The restriction to a small number of sets is necessary to avoid ambiguities about how to recurse. Also, the architecture you specify must have the same number of sets as the partition.

5. Input and output formats. Input to **Chaco** consists of one or more files, and the response to several interactive queries. Files are used to describe the graph, and if necessary to give geometric coordinates or an existing partition. The interactive input specifies the partitioning method and the number of sets you require. An additional optional file can be used to modify the values of various parameters that control algorithmic choices and output options. This functionality is discussed in §6.10.

5.1. Format of graph input files. The standard **Chaco** input is a graph, which is read from a file. Leading lines in this file that begin with the character '%' or '#' are considered comments and ignored. At its simplest a correct input file contains $n + 1$ uncommented lines, where n is the number of vertices in the graph. The first of these lines contains two required integers and may have a third. The first integer is the number of vertices in the graph, and the second is the number of edges. (Note that the number of edges is half of the sum of the number of neighbors of each vertex.) The remaining n lines contain neighbor lists for each vertex from 1 to n in order. These lists are just sets of integers which are separated by spaces and contain all the neighbors of a given vertex. The neighbors may be listed in any order. Note that vertices are numbered from 1 to n, not from 0 to $n - 1$. Sample graph files can be found in subdirectory "exec" under file names ending with ".graph".

Chaco also accepts graphs with weights on vertices and/or edges. A third parameter on the first line of the input file controls input of weighted graphs. This number may have up to three digits. If the 1's digit is nonzero, edge weights will be read. If the 10's digit is nonzero, vertex weights will be read. And if the 100's digit is nonzero then vertex numbers will be read, as described below.

Vertex weights should have small integer values. (To be conservative, the sum of all vertex weights should be representable as a standard integer.) If any vertex has a weight, then weights must be given for all of them. Vertex weights appear immediately before the corresponding neighbor list.

Edge weights can be any positive floating point value, but you are encouraged to make them small integers. *Kernighan–Lin and multilevel–KL will not work properly if edge weights are not integers.* If any edge is weighted, they all must be. Edge weights are included in the graph file immediately after the corresponding entry in the neighbor list.

If you have some vertices with many neighbors, it may be inconvenient to write the entire vertex data on a single line of the graph input file. You can split the data across multiple lines by using vertex numbers. The vertex number is the first value on a line containing data for that vertex. If you specify a vertex number for any vertex you must specify one for all of them, and vertices must still be entered in increasing order.

The most general form of the graph input file is illustrated below. The different types of optional entries are indicated by different styles of parenthesis. The digit on the first line which controls each type of optional entry is indicated by the same style of parenthesis.

```
% This is the format of the graph input file
Number-of-vertices   Number-of-edges   {1}[1](1)
{Vertex-number} [Vertex-weight]   neighbor_1 (edge-weight_1) ...
                                  ⋮
```

There is one exception to this general graph format. If you are using the inertial method or one of the simple methods without Kernighan–Lin, then it is not necessary to provide a graph since the partitioner does not make any use of connectivity information. A graph file is still needed to read the number of vertices, but the remaining lines describing the edge lists can be skipped. Note however that the code will be unable to evaluate the quality of a partition or perform any of the post–processing options without edge information. Normally several measures of the partition quality are computed and printed, but this is skipped if the graph is not present.

5.2. Format of coordinate input files. If you are using the inertial method, you will need to provide geometric coordinates for all vertices. These are placed in a different file, examples of which can be found in subdirectory "exec" with names ending with ".coords". These geometry files must have n uncommented lines, with line i containing the coordinates of vertex i. Each line must have 1, 2 or 3 real values, corresponding to a one-, two- or three-dimensional geometry. **Chaco** determines the dimensionality by looking at the number of values on the first line. Any number of comment lines can appear at the front of this file beginning with '%' or '#'.

5.3. Format of assignment input files. As discussed in §4.4, **Chaco** can take an existing partition and modify or evaluate it in several different ways. The existing partition is read from a file using one of two possible formats. In the standard format, the top of the file has an arbitrary number of comment lines indicated by a leading '%' or '#'. There follow as many lines in the file as vertices in the graph. Uncommented line i contains a single integer which is the set to which vertex i is assigned. Note that set assignment numbers start at zero.

The standard format can be inconvenient for parallel computing applications since the vertices owned by a particular processor can be scattered throughout the file. It can be useful to invert the standard format, having all the vertices assigned to processor 0 first, followed by all the vertices assigned to processor 1, etc. This input format can be selected by setting the parameter `IN_ASSIGN_INV` to be `TRUE` (nonzero), as described in §6.1. With this format, the file again begins with an arbitrary number of comment lines beginning with '%' or '#'. The next line contains a single value n_0 which is the number of vertices in set 0. The following n_0 lines list the vertices in set 0. This is followed by a line containing n_1, the number of vertices in set 1, and so on.

5.4. Operating the code. To operate the code you must answer a sequence of questions. With a basic understanding of the code structure and the methods described in §3, these questions should be mostly self–explanatory. A brief outline and a few notes are, however, in order.

First you will be asked to provide the name of the graph input file. If the `OUTPUT_ASSIGN` or `ECHO` parameters from §6.1 are set appropriately, you will also be asked for the names of output files. (If the text output file controlled by `OUTPUT_ASSIGN` already exists, the new output is appended to the existing file.) You will then select global and local partitioning methods from those described in §3. (Since multilevel–KL automatically performs KL, you aren't asked to specify a local method with this global option.) The global method options are (1) Multilevel–KL, (2) Spectral, (3) Inertial, (4) Linear, (5) Random, (6) Scattered and (7) Read–from–file. Option (7) is discussed in detail in §4.4. The local method options are currently (1) Kernighan–Lin and (2) None.

Depending upon your method selections, you may need to answer a few additional questions. If you choose a spectral method you will need to choose between the multilevel RQI/Symmlq eigensolver and Lanczos. If you select the inertial method you will need to specify the name of a coordinate input file. And if you ask for multilevel–KL or the multilevel eigensolver you will need to say how many vertices you want in the coarsest graph. (We generally use values in the range 50 to 500 for this parameter.) Note that because quadrisection and octasection make use of higher frequency information, they may need a slightly larger coarsest graph to resolve things as well as bisection does.

Chaco will then ask you for the size of the parallel machine for which you are partitioning and compute the appropriate number of sets. **Chaco** knows about the topology of hypercube and mesh parallel machines; you select between them by using the `ARCHITECTURE` parameter discussed in §6.7. The code makes an effort to assign sets to processors in a way that improves data locality on the selected architecture. Although the mapping to processors will be best for the architectures the code understands, it is important to note that **Chaco** generates *partitions* that are appropriate for any application. If mapping isn't important in your application, you can use `ARCHITECTURE` to specify a one-dimensional mesh and simply enter the number of sets you require.

Finally you will choose whether to apply the partitioning method in bisection, quadrisection or octasection form. Note that if you choose quadrisection or octasection and an integral number of steps will not produce the specified total number of sets, **Chaco** will automatically change to either quadrisection or bisection at the end of the recursion so as to generate the required number of sets.

Chaco will now go off and do the requested calculation, printing results to the screen and/or files. Afterwards, it will ask you whether you wish to run another problem.

5.5. Output formats. **Chaco** has various output options which are controlled by parameters described in §6.1 and §6.9. As the values of these parameters are increased, more detailed information is printed. If they are all set to zero, no output is produced under normal circumstances. There are, however, a few unrecoverable error messages which have authority to override this. The parameter `OUTPUT_METRICS` controls the calculation and printing of several partition metrics. These metrics can be displayed in a summary form with maximum, minimum and total number indicated, or they may be displayed in a detailed, set by set manner. The metrics of partition quality recorded are:

Set Size: The total weight of the vertices in a set. In a balanced decomposition these values should be as close as possible.

Edge Cuts: The weight of edges which connect a vertex in a set to vertices in a different set.

Hypercube Hops: A measure in which each cut edge is multiplied by the architectural distance between the two processors owning the end vertices. This metric often models communication time better than cuts does because it takes into account network congestion.

Boundary Vertices: The weight of vertices which have edges connecting them to a different set. For example, if an unweighted vertex in set 1 has three edges connecting it to set 4, its contribution to the boundary vertices total is one. If it also had an edge to set 7, its contribution would be two. This is useful in modeling applications like parallel matrix–vector multiplication in which the value associated with a vertex may be communicated to another set just once and used multiple times.

Boundary Vertex Hops: Boundary vertices weighted by the number of wires a message must traverse between corresponding processors. This adjusts the boundary vertices metric to account for congestion.

Adjacent Sets: The vertices in a particular set will have edges connecting them to some number of other sets. This metric counts the number of those other sets. This value corresponds to the number of messages the corresponding processor will have to send.

Internal Vertices: The total weight of all the vertices in a set which have no edges connecting them to vertices in other sets. As discussed in §6.6, the presense of such vertices may allow for overlapping communication with computation.

Assorted timing information is displayed under control of `OUTPUT_TIME`. This information, along with the input values and the settings for all the relevant parameters can be written to either the screen or both the screen and a designated file under control of the `ECHO` parameter.

Normally **Chaco** asks questions interactively, but if you are piping a file as input, you may want to switch the prompts off. You can do so by setting the `PROMPT` parameter to `FALSE`.

Chaco can also write an output file containing the partition assignments. Whether or not a file is generated is controlled by the parameter `OUTPUT_ASSIGN`, as discussed in §6.1. There are two assignment file formats which are the same as the input formats described above in §5.3. In the standard output format, line i contains a single number indicating the set to which vertex i is assigned. (The set numbers begin at zero.) In the inverted format, the first line of the file contains n_0, the number of vertices in set 0. The following n_0 lines contain the vertices assigned to set 0. The next line has n_1, the number of vertices assigned to set 1, and so on. This inverted format can be useful in parallel computing applications because the vertices owned by a particular processor can be read without having to scan the entire assignment file. If you prefer this inverted format, set the parameter `OUT_ASSIGN_INV` to `TRUE` (nonzero) as discussed in §6.1.

6. User–modifiable parameters. We have collected most of the internal parameters which control the operation of **Chaco** into the file "user_params.c" in the directory "code/main". If you wish to modify some of these parameters you have two options. You can edit the file "user_params.c" and recompile the code, effectively changing the default values[6]. Alternately, you can modify the values at run time as described in §6.10.

There are three basic types of parameters, those that control output type and quantity, those that select among different algorithmic variants and those which turn on and off additional functionality. The default values for the debugging parameters generate a minimal amount of output. This can be increased or in some cases decreased as desired. The defaults for the execution parameters were selected to provide a reasonable balance between run time and quality of the solution, but we make no claim to having selected them optimally for your problem. The default setting for the extended functionality parameters is off. The parameters and their functions are described in the sections below.

6.1. Input and output control parameters.

`CHECK_INPUT` If `TRUE` (nonzero), the graph and input parameters are checked for errors. Although checking the graph can take a few seconds for large problems, this feature should probably be left active (the default) for robustness. (The time spent checking will be printed out if you set the parameter `OUTPUT_TIME` to be greater than zero.)

`ECHO` This parameter controls the printing of the input values and parameters, as well as whether to copy these values to a file. A value of 0 induces no echoing. If `ECHO` is 1 (or -1), the input selections will be echoed to the screen. If it is 2 (or -2), then the relevant user parameters will also be echoed. If the value is less than zero, you will be asked for the name of a file in which to record the results of a run. This file will contain the same input selections and parameters that are copied to the screen, along with partition metrics (controlled by `OUTPUT_METRICS`), a run time breakdown a run time breakdown (controlled by `OUTPUT_TIME`) and any warning or error messages generated by the code. Saving these results in a file can be useful if you are doing a sequence of runs for later analysis. The default value is 2.

`OUTPUT_METRICS` This parameter controls how much information about the quality of the partition will be computed and printed out. A zero value means that no evaluation will be performed or printed. A negative value generates output about each set instead of just a summary of minimum and maximum values over

[6] It might be prudent to save a copy of the original file so that you can return to the "factory settings" easily. The default values quoted in the text assume no changes have been made to this file.

all sets. A value of 1 (or -1) produces information about the final partition. If you are partitioning for a hypercube, a value of 2 (or -2) generates data about all the intermediate partitions for smaller hypercubes that were implicitly generated in the process. The meanings of the various output metrics are described in §5.5. The default value is 2.

`OUTPUT_TIME` This value determines how much information gets printed about the run-time of **Chaco**. A value of 0 means that nothing is printed, and values of 1 or 2 allow for increasingly detailed timing output. The default value is 2.

`OUTPUT_ASSIGN` If this value is `TRUE`, you will be prompted for the name of a file in which the vertex assignment will be printed. A description of the format of this output file can be found in §5.5. The default for this parameter is `FALSE` (zero).

`OUT_ASSIGN_INV` If `OUTPUT_ASSIGN` is `TRUE` so you are writing an assignment file, then this parameter controls the format of that file. In the standard output format, line i of the file contains the set to which vertex i is assigned. In some settings it is preferable to use an inverted format in which all the vertices in set 0 come first, followed by all those in set 1, etc. If you prefer this inverted format, you should set `OUT_ASSIGN_INV` to be `TRUE`, in which case the assignment will be printed in the format described at the end of §5.5. The default value is `FALSE`, corresponding to the standard format.

`IN_ASSIGN_INV` If you are reading an assignment from a file, then the file should be in one of the two formats described in §5.3. In the standard format, the ith uncommented line contains the set to which vertex i is assigned. In the inverted format, all the vertices in set 0 are specified first, followed by those in set 1, and so on. If `IN_ASSIGN_INV` is `FALSE` (the default) then the standard format is assumed. If set to `TRUE` the the inverted format is expected.

`PROMPT` **Chaco** assumes you are answering the input questions interactively. However, if you are piping a file into **Chaco**, it may be more aesthetic to skip the input questions. Setting `PROMPT` to `FALSE` keeps the code from explicitly asking for inputs. The default is `TRUE`.

`PRINT_HEADERS` This parameter controls whether or not titles are printed for the different sections of output. The default value is `TRUE`.

6.2. Eigenvector calculation parameters.

`LANCZOS_TYPE` If you are using a spectral partitioning method or the multilevel–KL method, Lanczos is used at some point as an eigen solver. (The multilevel–KL method uses Lanczos to generate a spectral partitioning of the coarsest grid, and the RQI/Symmlq eigen solver also uses Lanczos on the the coarsest graph.) A discussion of the relative merits of the different methods can be found in §3.3. A value of 1 selects full orthogonalization, a value of 2 chooses full orthogonalization with the inverse operator, and a value of 3 selects selective orthogonalization The default value is 3.

`EIGEN_TOLERANCE` This one probably deserves its own short paper. All we can do here is make a few general remarks and urge caution. If you are using a pure spectral method or the multilevel–KL partitioning method then you need to calculate eigenvectors. This parameter controls how accurately you compute them. If you are using one of the Lanczos methods and LANCZOS_CONVERGENCE_MODE is set to 0, then EIGEN_TOLERANCE is a tolerance on the eigen residual $||Au - \lambda u||$ where (λ, u) is the eigen pair of A in question. Similarly, if you are using the multilevel RQI/Symmlq method to compute eigenvectors and RQI_CONVERGENCE_MODE is set to 0, the eigen residual is used in the convergence test. If a convergence mode flag is set to 1 then the convergence of the corresponding iterative method is instead monitored with respect to the *partition* residual. That is, the iteration pauses periodically and a partition is computed based on the current approximation to the eigenvector. When the change in partition cut size since the last pause is less than EIGEN_TOLERANCE times the number of graph vertices, the eigenvector computation terminates. These latter modes provide the ability to automatically choose the accuracy of the eigenvector computation to achieve any level of stability in partition quality.

An extremely accurate eigenvector computation is expensive, and probably unnecessary, particularly if you are using Kernighan–Lin to refine the spectral partition. However, in general the quality of the partition gradually degrades as the accuracy is reduced below some critical point. This can be a result of inaccuracy in the eigenvector, or it may be because the eigen solver has converged to an entirely wrong eigen pair. This latter phenomenon of misconvergence occurs quite frequently if you use too large an eigen tolerance because there are many eigenvalues in any interval of that width. So to be really correct one should probably relate the eigen tolerance to the expected gap between eigenvalues in the relevant portion of the spectrum using, for example, the graph size. But, as discussed earlier in §3, slight misconvergence is not a grave problem since misconverged eigenvectors often give good partitions. The multidimensional spectral methods do in general require somewhat higher accuracy than spectral bisection to perform at their best. Apart from this, however, the question of the appropriate eigen tolerance and risk of misconvergence is more a question of being able to reproduce partitions reliably and of having a fair basis on which to compare eigen solvers. **Chaco**'s design philosophy here is that you should get the accuracy you request, and, failing that, you should be warned and told the accuracy you did get. We feel the largest value of EIGEN_TOLERANCE that is advisable for general use is about 10^{-3}, and that is what we ship the code with. If you are really pressed for speed and are using a local clean–up phase, a value of 10^{-2} might be reasonable. At the other extreme, a value of 10^{-6} should prove acceptably tight in most situations — if you're working on a graph large enough to require higher accuracy, you should probably switch to the multilevel–KL partitioning method, which for large problems generally gives better answers in less time.

SRESTOL If this parameter is non-negative and the residual encountered at the end of the recurrence used to compute the eigenvector of the tridiagonal matrix in

Lanczos is greater than it, a corresponding set of warning conditions is flagged. (See discussion of `WARNING_EVECS`.) If this parameter is negative, the residual tolerance for the eigenvector of the tridiagonal matrix is automatically set to the square of `EIGEN_TOLERANCE`. The default value is -1, so the tolerance is set automatically. If you are frequently warned that the tolerance on this computation is not achieved and you are not getting the overall Lanczos accuracy you have requested, try increasing `BISECTION_SAFETY`. If you get frequent warnings about `SRESTOL` and you are achieving the Lanczos accuracy you want, either specify a value of SRESTOL which is looser (bigger) than the square of the eigen tolerance, or (if the warnings bother you) reduce the value of `WARNING_EVECS` appropriately.

`LANCZOS_SO_INTERVAL` If you are using the selective orthogonalization variant of Lanczos, then the convergence of the process is checked indirectly through the Ritz pairs every few steps. The number of Lanczos iterations between checks is set by the value of this parameter. Choosing a large value will generally make the computation run marginally faster, but increases the risk of degraded accuracy or misconvergence and may therefore actually increase run time. A smaller value is more robust since numerical breakdown due to the convergence of Ritz pairs will be detected sooner. If you encounter convergence problems while using selective orthogonalization, try reducing this parameter. Due to the details of the orthogonalization procedure, a value of 1 will cause redundant work, so the minimum sensible value is 2; the default is 10.

`LANCZOS_MAXITNS` If this parameter is set to a non-negative integer, Lanczos will terminate at that number of iterations. If it has a negative value, the maximum number of Lanczos iterations will be set automatically to twice the number of vertices in the graph, *i.e.* it will be $2n$, where n is the order of the matrix in the eigen system. Except in rare circumstances Lanczos will converge before n iterations, so this autoset feature in practice means that Lanczos will iterate until it converges to tolerance. The default is -1 for autoset.

`BISECTION_SAFETY` In Lanczos some of the extremal eigenvalues of the tridiagonal matrix must be found periodically. If the number of eigenvalues to be found is small, a bisection algorithm is used to find roots of the Sturm sequence which correspond to the eigenvalues. This parameter amplifies or shrinks the convergence tolerance on the bisection algorithm. A higher value specifies a tighter (smaller) tolerance and results in more accurate computation of these eigenvalues, but a slightly longer run time. If the code encounters numerical accuracy problems it thinks are related to accuracy of the eigenvalues of the tridiagonal, it will dynamically increase the amplification of the convergence tolerance for the bisection computation by some multiplicative factor. The next time Lanczos is invoked the amplification is reset to `BISECTION_SAFETY`, which has a default value of 10.

`LANCZOS_CONVERGENCE_MODE` If the code is performing spectral bisection and this parameter is set to 1, the convergence of the Lanczos iteration is determined by monitoring convergence of the partition rather than the eigen residual. At each Lanczos pause an approximate eigenvector is computed and used to generate the current partition. If the partition has changed less than `EIGEN_TOLERANCE` times the number of vertices, the iteration is considered converged. This is useful if you want to determine the accuracy of the eigenvector in an adaptive way. For example, you may want to iterate until the point at which further iteration will not change the partition. Computing eigenvector approximations frequently within Lanczos is, however, very expensive because it requires a sum across all the current Lanczos basis vectors. We therefore recommend that you generally leave this parameter in its default state of 0 so that convergence will be evaluated in the normal way by comparing the eigen residual against the eigen tolerance. *Note that when using spectral quadrisection or octasection there is no choice — convergence mode 0 will be used.*

`RQI_CONVERGENCE_MODE` This parameter plays the same role as `LANCZOS_CONVERGENCE_MODE`, but in the RQI/Symmlq context. If it is set to 0, RQI convergence happens when the eigen residual is less than `EIGEN_TOLERANCE`. If the parameter is set to 1 an additional check is invoked based on whether the partition has changed since the last step by less than `EIGEN_TOLERANCE` times the number of vertices. Since RQI is computing a new approximation to the eigenvector on each step, this additional convergence check is relatively economical. And, since the partition often converges to reasonable accuracy before the eigenvector does, we have made convergence mode 1 the default. *If you are comparing run times of Lanczos and RQI/Symmlq you should, to be fair, use the same convergence modes for both.*

`LANCZOS_SO_PRECISION` The selective orthogonalization version of Lanczos performs its dominant computations (sparse matrix-vector multiplication and blas-type operations) on data of type `float` when this parameter is set to 1; when it is set to 2, type `double` is used. Computations of the type in question are less accurate if performed on data of type `float` than if performed on type `double` because the result is stored in lower precision. On some machines and using some C compilers, floating point operations performed on `float` data are faster than those performed on `double` data. But they may be (and often are) actually *slower*. You can test this for your computing environment by setting the `TIME_KERNELS` to `TRUE`. Using type `float` can however lead to significant memory savings in this context because the Lanczos basis, which generally dominates the storage requirements, occupies half as much memory. The default value of this parameter is 2, and we recommend that you generally use this value unless you are running out of memory since the compute–time saving (if any) is rarely significant.

`WARNING_EVECS` If this parameter has a value greater than 0, the occurrence of a variety of possible numerical or storage-related problems in the eigen solvers is reported. When using RQI/Symmlq, a value above 0 means you will be notified if the eigen residual is not converging monotonically, an indication of possible misconvergence. When using Lanczos, a value above 0 means you will be warned if the requested eigen tolerance was not achieved, if there has been a minor or severe loss of orthogonality in the computation, if the maximum number of Lanczos iterations was reached and if the code needed to switch tridiagonal solvers to accurately compute the Ritz values. You will also be notified if the code has run out of memory and is recovering by computing the best available approximation to the eigenvector. A value above 1 means that if any of the preceding warning conditions occur, you will be notified of the eigenvalues and predicted and actual eigen residual tolerances. A value above 2 means you will be notified when the computation of the eigenvector of the tridiagonal matrix has been problematic and if the back-up iteration was used (and how many times) for this computation. If the extended eigen solver is used, not all this warning information is provided. Various warnings are reported when the extended eigen problem is not well posed and this parameter is set greater than 0. The default value is 2.

`WARNING_ORTHTOL` This parameter determines the level of loss of orthogonality in Lanczos which is considered minor but worth reporting. If the ratio between the estimate of the eigen residual and the computed eigen residual is above this value, the minor loss of orthogonality condition is triggered. To avoid generating insignificant messages, warnings are not printed if the actual eigen residual is significantly lower than the eigen tolerance. The default value is 2. Refer to the discussion on `WARNING_EVECS`.

`WARNING_MISTOL` Same as `WARNING_ORTHTOL`, but this value indicates a more serious loss of orthogonality. In some cases this may indicate misconvergence, hence the name. The default is 100.

`LANCZOS_TIME` A detailed breakdown of the time spent in different stages of the Lanczos eigen solver is provided when this parameter is set to `TRUE`. Lanczos will run ever so slightly faster if you leave this value at `FALSE` (the default), since many fewer calls to the timing function will be made.

`TIME_KERNELS` If this parameter is set to `TRUE` a table is printed out comparing various kernel operations in single precision (data type `float`) and double precision (data type `double`). The kernel operations are basic linear algebra primitives and multiplication of a dense vector by the weighted Laplacian matrix of the graph. The comparison is with respect to numerical result and execution time. The number of loops of the kernel operations performed is chosen so that the time of the standard 2–norm operation is approximately 1 second, hence on very large and dense graphs the sparse matrix multiplication kernel timing test may require significant time. The default value is `FALSE`.

6.3. Other parameters for spectral methods.

`MAKE_CONNECTED` Spectral methods can break down if the graph is disconnected. Even if the original graph is connected, disconnected graphs can be generated in the recursion. To avoid any associated problems, we use a breadth-first-search algorithm to find connected components and add a minimal number of edges to make the graph connected. If `MAKE_CONNECTED` is `TRUE` (the default), then this connectivity check will be invoked whenever a spectral option is selected. You should only change this parameter if you plan to use a spectral method and you are certain that you will only operate on connected graphs (*i.e.* if you aren't recursing).

`PERTURB` Spectral methods can encounter problems if the graph has symmetry since its eigenvalues can then have multiplicity greater than 1. For spectral bisection, all you can hope for is selecting some vector (which depends on the starting Lanczos vector) in the subspace of second lowest eigenvectors. However, since they work within a subspace of 2 and 3 vectors respectively, spectral quadrisection and octasection can handle two or three degrees of multiplicity respectively. Unfortunately, Lanczos can't easily identify this multiplicity. We can, however, avoid the issue by randomly perturbing the matrix. If you are invoking bisection, then the matrix is not perturbed, but in quadrisection or octasection mode the parameter `PERTURB` controls whether or not this perturbation is invoked. Using this option helps avoid problems in some degenerate cases like the square grid graph, at the cost of a very slight increase in run time. We recommend that you leave this feature actived (the default) unless you are sure you don't need it.

`NPERTURB` If the `PERTURB` option is being used, this parameter indicates how many random edges are added to the graph to break the symmetry. The default is 2.

`PERTURB_MAX` If the `PERTURB` option is being used, this parameter is the maximum value of an edge weight for one of the randomly added edges. A small value will perturb the eigenvectors a small amount, but if the perturbation is too small, then Lanczos may not be able to separate the multiple eigenvectors. This value should probably be a small multiple of `EIGEN_TOLERANCE`. The default is .003.

`MAPPING_TYPE` We have implemented several methods for generating a partition from eigenvectors, and decided to retain two of them. If this value is 0, then the the partitions are determined by the signs of the values in the eigenvector(s). Note that this will generally produce a somewhat imbalanced partition (which can be balanced by KL). In the bisection case, this option reduces to dividing at a value of zero. If `MAPPING_TYPE` is 1, then the code uses the minimum cost assignment algorithm described in [12], which generates balanced sets. In the bisection case, this latter option reduces to dividing at the median. Since we consider this second approach superior, the default value is 1.

`COARSE_NLEVEL_RQI` This parameter applies if you are using the spectral method with the RQI/Symmlq eigen solver option. As you work back through the intermediate graphs, the approximation to the eigenvector is refined with Rayleigh Quotient Iteration every few levels. This parameter indicates how many levels occur between these refinements. A small value for this parameter is more robust, but a large value will reduce execution time. The default value is 2.

`OPT3D_NTRIES` If you are using spectral octasection, then when mapping back to a discrete solution you need to solve a constrained, global optimization problem as described in [12]. In our experience, this problem usually has a small number of local minimizers, so we solve it using local minimization techniques from random starting points. This parameter controls the number of local minimizations, and should only be modified by sophisticated users. The default value is 5.

6.4. Kernighan–Lin parameters.

`KL_METRIC` When dividing into more than 2 sets at once, our implementation of Kernighan–Lin can try to minimize any inter–set metric. Two are currently built into the code and are controlled by this parameter. If the value of `KL_METRIC` is one, then all edges crossing between two sets are treated the same. If the value is two, then edges are weighted by a metric that corresponds to their architectural distance in the target parallel architecture. (Note that the spectral quadrisection and octasection algorithms automatically use a hypercube hop metric.) Also note that in bisection the choice of metrics doesn't matter. If you wish to use a different metric than cuts or hops, you can tinker with the appropriate code in "code/submain/submain.c". The default value is 2.

`KL_RANDOM` This flag turns on and off the randomness in the Kernighan–Lin routines. We recommend that you leave this parameter in the default setting of `TRUE` since it increases the quality and robustness of Kernighan–Lin for a tiny increase in run time.

`KL_BAD_MOVES` Our version of Kernighan–Lin can exit a pass early if it doesn't seem to be making any progress. This parameter controls how quickly KL will hit this cutoff. A large value may make KL more effective, but will also increase the run time. The default is 20.

`KL_NTRIES_BAD` This parameter controls the speed at which the Kernighan–Lin code is exited. The KL routine will exit after `KL_NTRIES_BAD` passes in which no improvement is detected. We have designed some randomness into this algorithm, so a pass with no improvement can be followed by one that finds a better partitioning. However, if you set `KL_RANDOM` to `FALSE`, then you should set `KL_NTRIES_BAD` to 1. If `KL_NTRIES_BAD` is set to zero, then the code will run a single pass of KL and exit whether or not the partition is improved. Since the first pass is usually responsible for the bulk of the improvement, this is a reasonable choice if run time is critical. A large value for this parameter should produce better results, but will cause the code to run longer. The default is 1.

`KL_UNDO_LIST` This parameter turns on an optimization that dramatically reduces the run time of Kernighan–Lin for large graphs. Instead of bucket sorting the entire set of possible vertex moves before each pass, this option preserves the moves that haven't been changed; typically the vast majority. This leads to a dramatic increase in speed, with no perceptible change in quality. We strongly encourage you to leave this parameter in its default setting of `TRUE`.

`KL_IMBALANCE` **Chaco** generally tries to keep the vertex weight sums in the sets it generates as nearly equal as possible. Specifically, when a single division step is performed, the difference in vertex weight sum between any two of the subsets is at most the weight of the heaviest vertex. For weighted graphs, after several levels of recursive partitioning, the set sizes may deviate by more than the weight of a single vertex. But for unweighted graphs set sizes should vary by at most one. This is the default operation of the code, associated with a value for `KL_IMBALANCE` of 0. However, there are settings in which sets needn't be perfectly matched in size, and an imbalanced partition with fewer crossing edges is preferable. Such partitions can be found by **Chaco**'s implementation of KL and multilevel–KL. If `KL_IMBALANCE` is set to some value q between 0 and 1, then KL and multilevel KL will look for partitions in which the the fractional imbalance is no more than q. Specifically, the difference between any two sets is bounded by q times the average set size.

6.5. Parameters for multilevel algorithms.

`COARSEN_RATIO_MIN` This value is employed if you are using either the RQI/Symmlq eigen solver or the multilevel–KL partitioning algorithm. It should have a value between .5 and 1.0, representing the minimal acceptable reduction in number of vertices associated with a coarsening step. If a step fails to achieve this reduction, the coarsening algorithm exits prematurely, and the resulting calculations will be performed on a larger graph than expected. The coarsening algorithm cannot reduce the number of vertices by more than half, so this value should always be greater than .5; the default is .7.

`COARSE_NLEVEL_KL` If you are using the multilevel–KL partitioning algorithm, then Kernighan–Lin gets invoked periodically on successively finer graphs. This parameter indicates how many levels occur between these invocations. A small value for `COARSE_NLEVEL_KL` will generally lead to better partitions, while a large value will reduce execution time. The default is 2.

`COARSE_NLEVEL_RQI` See discussion in §6.3.

`MATCH_TYPE` The first step in coarsening is the generation of a maximal matching in the graph. We have three matching codes to choose from. The default value for `MATCH_TYPE` is 1, which selects a fast algorithm based on a breadth first search. Increasingly time consuming but more truly random algorithms are invoked by larger values up to a maximum of 4.

`HEAVY_MATCH` Karypis and Kumar [16] have reported that the multilevel–KL algorithm is improved by selecting matching edges that have high weights. If this parameter is set to `TRUE` then the matching algorithms for either the multilevel eigen solver or multilevel–KL will try to generate heavy weight matchings. The default for this parameter is `FALSE`.

`COARSE_KL_BOTTOM` Kernighan–Lin refinement is invoked every `COARSE_NLEVEL_KL` levels starting with the finest graph. To ensure that partition of the coarsest graph is as good as possible, it makes sense to invoke KL on this graph. If `COARSE_KL_BOTTOM` is `TRUE` (the default), KL will always be invoked on the coarsest graph. This generally improves the quality of partitions, and since the coarsest graph is small, the time for this process is negligible.

COARSEN_VWGTS For both the multilevel–KL algorithm and the RQI/Symmlq eigen solver, we construct coarse graphs via a sequence of edge contractions. These contractions combine two vertices into one. Balance constraints are preserved if the weight of the combined vertex is the sum of the weights of its two constituents. If COARSEN_VWGTS is TRUE (the default) then the new vertex is weighted in this manner. However, as discussed in §3.6, Bui and Jones proposed an algorithm where the new vertex has unit weight. This can be implemented by setting COARSEN_VWGTS to FALSE.

COARSEN_EWGTS Edge contraction can also cause edges in the graph to fall on top of each other. If so, the cost of a partition can be preserved by making the weight of the resulting edge equal to the sum of the weights of those that comprise it. This weighting is performed if COARSEN_EWGTS is TRUE (the default). In the algorithm by Bui and Jones the resulting edge is instead given a unit weight, which will happen if COARSEN_EWGTS is FALSE.

KL_ONLY_BNDY If the vertices on the boundary are known, then it is more efficient to initialize only these values when starting Kerninghan–Lin. Other values can be evaluated on an as–needed basis, but typically only a small fraction will need to be computed. Note that vertices not initially on a boundary can be handled properly; they're just not placed into the KL data structures until one of their neighbors has moved, placing them on the new boundary. In the multilevel–KL algorithm, the boundary can be easily propagated between levels, so this efficiency can be realized. This significantly improves the speed of the algorithm without affecting its quality. When runing multilevel–KL, if KL_ONLY_BNDY is set to TRUE (the default) then only those vertices on the boundary between sets are initialized.

KL_IMBALANCE See discussion in §6.4.

6.6. Parameters for post–processing options.

REFINE_PARTITION This parameter controls how many sweeps are made through the pairs of sets with a nonzero boundary in an effort to improve the partition via an invocation of Kernighan–Lin. A discussion of this approach can be found in §4.3.1. The default value is zero since the process is fairly expensive.

INTERNAL_VERTICES In many parallel computing applications, vertices with no edges to other sets require only local data. This may allow for overlap of communication and computation. If TRUE, the INTERNAL_VERTICES parameter activates code to try to increase the minimal number of internal vertices in a set. The algorithm for this task is sketched in §4.3.2. The default value for this parameter is FALSE since this fairly specialized functionality is probably not required for most applications. We also caution that this can be a time consuming process if the partition submitted is of low quality.

REFINE_MAP If this parameter is TRUE the mapping of sets to processors is modified in a greedy manner to improve locality. Note that this doesn't change the composition of the sets, just which processor each set is assigned to. The algorithm for this process is described in §4.3.3. Since **Chaco** is used for many applications other than parallel computing, the default for this parameter is `FALSE`. But if you are really interested in the quality of your mapping to processors you should set it to `TRUE`.

6.7. Architecture parameters.

`ARCHITECTURE` In addition to partitioning the graph, **Chaco** tries to assign subgraphs generated to processors of a parallel computer in an intelligent manner. This parameter specifies the topology of the parallel computer. A value of 0 (the default) specifies a hypercube, while values of 1, 2 or 3 indicate meshes of dimensionality one, two or three respectively. If your application doesn't care about how pieces get assigned to sets, then you should simply set this value to 1 and input the number of sets you require when prompted. This might be the case, for example, if you were partitioning for a heterogeneous network of computers coupled by a slow or unpredictable network.

6.8. Miscellaneous parameters.

`TERM_PROP` This parameter determines whether or not terminal propagation is invoked in spectral bisection, Kernighan–Lin and multilevel–KL. Details concerning terminal propagation, a method for generating better mappings to processors, are given in §4.2. Currently, spectral terminal propagation only works in bisection mode, but with KL and multilevel–KL the method can handle an arbitrary number of sets. The default value is `FALSE`.

`CUT_TO_HOP_COST` When performing terminal propagation, this value controls the relative importance of generating a new cut edge versus increasing the inter–processor distance associated with an existing cut edge. This parameter thus allows the user to tradeoff the importance of communication volume to communication locality. The default value is 1.0.

`SEQUENCE` If this parameter is set to `TRUE`, **Chaco** computes and sorts the Fiedler vector of the graph and places the result in a file named by `SEQ_FILENAME`. The other operations associated with partitioning are not performed. This functionality is provided to assist development of spectral graph algorithms, and is discussed in §4.1. The default value is `FALSE`.

`SEQ_FILENAME` This parameter is a character string which specifies the name of the file to which the sorted Fiedler is printed if `SEQUENCE` is nonzero. The default is "`Sequence.out`"

`RANDOM_SEED` This is the seed for the random number generator "rand()".

`NSQRTS` If you are using either multilevel–KL or the RQI/Symmlq eigen solver, then coarse versions of the graph are created with vertex weights. The square roots of these vertex weights are also needed. Since these are typically integers, **Chaco** avoids redundant computation by computing the root of each distinct integer once and storing it in the array `SQRTS`. The value of `NSQRTS` is the length of this array, and for best performance it should be somewhat larger than the number of vertices in the original graph divided by the number of vertices in the coarsest graph. A large value may use a small amount of unnecessary space, while a small value may lead to a slight excess of computation. The default is 1000.

MAKE_VWGTS In matrix–vector multiplication, the cost associated with a row is proportional to the number of nonzeros in that row. If vertices of your graph represent rows of a matrix, MAKE_VWGTS allows you to automatically weight them in this way. Note that if MAKE_VWGTS is TRUE then any weights in your graph file are ignored. The default for this parameter is FALSE, meaning that the option isn't invoked.

FREE_GRAPH **Chaco** first reads a graph into a simple format before converting it into a more complex data structure. This simple format is used to allow the code to be called from Fortran as described in §7. Once the graph has been reformatted the space used by the simple format is deleted if FREE_GRAPH is TRUE, which is the default. However, if you are calling the code from other software, you may wish to save the simple graph structure for other purposes, or you may not have generated it via C `malloc()` calls. In these cases, you should set FREE_GRAPH to FALSE, to disable this feature.

PARAMS_FILENAME This parameter defines the name of the file from which the code reads parameter modifications as described in §6.10. The default is "User_Params" in the executable directory. This is the only parameter that cannot be changed at run time. To change this file name you must edit the file "code/main/user_params.c" and recompile.

6.9. Parameters that control debugging output. These parameters allow you to invoke **Chaco**'s built-in debugging capabilities. The default value of these parameters, with one exception, is 0, specifying that no debugging output should be printed. (The exception is DEBUG_PARAMS, which has a default value of 2.) If no range of values is indicated, the parameter is treated as a TRUE/FALSE value and any nonzero value will activate it.

DEBUG_EVECS This parameter controls the quantity of debug output concerning calculation of eigenvectors. When set to zero, no output is generated except when an unrecoverable error condition is encountered, in which case a short message is printed before the program aborts. A value of 1 will produce a moderate amount amount of information, 2 a bit more, and so on up to a maximum value of 5.

DEBUG_KL This flag controls the output in the Kernighan–Lin routines. No debugging output is generated if the value is 0, while the improvement due to KL at each step is shown if the value is 1. Values of 2 and 3 generate large quantities of output, and should only be invoked by an expert.

DEBUG_INERTIAL If you are using the inertial method, this flag will turn on output concerning the computation of the principle axis of the graph.

DEBUG_CONNECTED If you are enforcing connectivity and using a spectral method, a value of 1 for this flag turns on a small amount of output in the routines that identify connected components. This will tell you if subgraphs have become disconnected in the course of a decomposition.

DEBUG_PERTURB A value of 1 for this flag turns on a small amount of output in the routines for randomly perturbing the matrix.

DEBUG_ASSIGN When using a spectral method, the mapping from the eigenvectors to a partition can be complicated, particularly for spectral quadrisection and octasection. This parameter turns on output in the routines that compute this mapping.

DEBUG_OPTIMIZE With spectral quadrisection or spectral octasection, part of the mapping to a partition involves a nonlinear optimization. This flag controls debugging output in the optimization subroutines.

DEBUG_BPMATCH When using spectral quadrisection or octasection, the trickiest part of the mapping from eigenvectors to a partition involves solving a minimal cost assignment problem in a bipartite graph. This flag turns on the output in the corresponding sections of the code. A value of 1 gives a moderate amount of cryptic output, while a value of 2 does more error checking and can generate a lot of output.

DEBUG_COARSEN If you invoke multilevel–KL or the RQI/Symmlq eigen solver, the code will construct a sequence of increasingly coarser approximations to the original graph. This parameter controls the output for the routines performing this process.

DEBUG_MEMORY This variable turns on some consistency checks in the allocation and freeing of memory. Unless you encounter problems you think might be memory related, this value should be left at 0.

DEBUG_INPUT If this is set to 1, a message is printed confirming that the input files have been read.

DEBUG_PARAMS This value controls how much output is generated while reading parameters from the "User_Params" file. A value of 1 means that the code will notify you of any parameter settings it does not recognize (and therefore ignores). A value of 2 prints a confirming message for each parameter value that is reset. The default value is 2.

DEBUG_INTERNAL If INTERNAL_VERTICES is nonzero, then the code will try to increase the number of entirely internal vertices in the sets with the fewest of them as described in §4.3.2. If DEBUG_INTERNAL is nonzero, debugging output will be generated in this section of the code.

DEBUG_REFINE_PART If REFINE_PARTITION is nonzero then **Chaco** tries to improve a partition by locally refining the boundaries between sets as discussed in §4.3.1. If DEBUG_REFINE_PART is nonzero, the code will generate debugging output in this operation.

DEBUG_REFINE_MAP If REFINE_MAP is TRUE, code is activated to swap sets among processors to improve locality. DEBUG_REFINE_MAP controls output in this process.

DEBUG_TRACE If this value is nonzero, messages are printed which reveal the main execution path. If the code is running into problems, this parameter may help narrow down where they are occurring.

DEBUG_MACH_PARAMS **Chaco** needs to compute a few numerical values that are machine dependent. If this flag is nonzero then the values it computes are printed out. If you are having difficulty getting the code to run on a new machine, the parameter calculation may be failing; this flag will help you detect that.

6.10. Modifying parameters at run time. You can modify the user parameters at run time by specifying the desired changes in a file called "User_Params" in the executable directory. (You can change the name of this file by modifying the value of PARAMS_FILENAME in "user_params.c".) The first thing **Chaco** does is read this file (if it exists), and make the specified changes to parameter values.

Lines of the "User_Params" file should contain a parameter name (using any combination of upper and lower case) followed by the new value. An '=' may be used to separate the parameter name and value, but is not required. Integer and real values are specified in the normal input manner, and logical parameters can be specified by strings starting with 'T' or 't' for true (one) and 'F' or 'f' for false (zero). Lines beginning with a '%' or a '#' are ignored.

If you are using a single invocation of **Chaco** to perform several decompositions, you can vary parameters between problems by adding a line to the "User_Params" file consisting of the string STOP. When **Chaco** encounters a STOP it quits reading parameters and partitions or sequences the graph. When the second problem is begun **Chaco** continues reading new parameter values from where it left off until it encounters another STOP or the end of the "User_Params" file. Any number of stop commands may be used. *Note that the parameter changes made for the first problem are still in effect unless later lines provide newer values.* Consider the following sample file.

```
%This is a sample run-time parameter modification file.
Architecture 3
term_prop True
#debug_memory = 1
Stop
TERM_PROP = f
```

In the first problem, the architecture is set to be a three-dimensional mesh, and terminal propagation is enabled. In the second problem, the architecture remains a three-dimensional mesh, but terminal propagation is now disabled.

œ v

7. Calling Chaco from other programs. Throughout this document we have assumed that **Chaco** is being used as a stand–alone program. However, this needn't be the case. We designed version 2.0 to allow for easy interface with other codes written in either C or Fortran. The mechanism for this interface is described below. Some familiarity with the remainder of this document is assumed.

The `interface()` routine and can be found in the file "code/main/interface.c". This is the routine that **Chaco** itself invokes after prompting the user for all the necessary input. Consequently, no functionality is lost by calling `interface()` yourself. The input can still be checked for consistency, all the output options are still active and the ability to modify parameters at run time by reading a file as described in §6.10 is maintained. (The parameters `ARCHITECTURE`, `EIGEN_TOLERANCE` and `RANDOM_SEED` are made obsolete by the arguments to `interface()` as detailed below, and `DEBUG_INPUT` and `PROMPT` become irrelevant, but all other parameters remain active.) The ability to control the *goals* argument described below actually gives you greater functionality than you would have in stand–alone mode.

The interface routine returns 0 if the partitioning is successful, and 1 otherwise. Typically, a return code of 1 indicates the detection of some inconsistencies in the input arguments. The arguments to `interface()` describe the graph, input and output files and arrays, properties of the desired decomposition and the requested partitioning algorithm. The arguments are described below in the order in which they occur.

A. Arguments describing the graph.

1. **nvtxs.** Type int. This is the number of vertices in the graph. Vertices are numbered from 1 to *nvtxs*.
2. **start.** Type int *. Although **Chaco** internally uses a C structure to represent the graph, a simpler representation at the start allows for interface with Fortran programs. The *start* array is of size (*nvtxs*+1). It's values are indices into the *adjacency* array. The values in *adjacency* from $start[i-1]$ to $start[i]-1$ are the vertices adjacent to vertex i in the graph. (Note that C arrays begin at zero, so in Fortran, the relevant range would be $start[i]$ to $start(i+1)-1$.)
3. **adjacency.** Type int *. As indicated in the description of *start*, this array contains a list of edges for all vertices in the graph. Note that if the `FREE_GRAPH` parameter from §6.8 is set to `TRUE`, then after converting to a new data structure, both *start* and *adjacency* are freed. If this is inappropriate for your application (*e.g.* you want to keep the graph, or you didn't dynamically allocate these arrays), then you should set `FREE_GRAPH` to `FALSE`.
4. **vwgts.** Type int *. This array of length *nvtxs* specifies weights for all the vertices. If you pass in a NULL pointer, then all vertices are given unit weight. Vertex weights should be positive.
5. **ewgts.** Type float * (Fortran type real*4). This array specifies weights for all the edges. It is of the same length as *adjacency* and is indexed in the same way. If you use Kernighan–Lin or the multilevel partitioner, these values will be rounded to the nearest integer. We suggest scaling them so they are neither very small nor very big. Edge weights should be positive.
6. **x.** Type float *. If you are using the inertial partitioner, you need to specify geometric coordinates for each vertex. This array of length *nvtxs* specifies the x coordinate for each vertex.
7. **y.** Type float *. This array specifies the y coordinate for each vertex. If it is NULL, the geometry is assumed to one-dimensional.
8. **z.** Type float *. This array specifies the z coordinate for he each vertex. If z is NULL and y is not NULL, the geometry is assumed to be two-dimensional.

B. Output file names.

9. outassignname. Type char *. If you desire the final assignment to be written to a file, this argument gives the name of that file. If this argument is NULL or if the parameter `OUTPUT_ASSIGN` is 0, then the assignment is not written to a file.

10. outfilename. Type char *. This is the name of a file in which the results of the run are printed. If it is NULL or if the parameter `ECHO` is not negative, then no file output is performed.

C. Assignment.

11. assignment. Type short *. This is the only output argument to `interface()`. It is an array of length *nvtxs* and returns the set number to which each vertex is assigned. The set number for vertex i is returned in $assignment[i-1]$ (or for Fortran, in $assignment(i)$). This can also be an input argument if *global_method*, argument 16 below, is set to 7. A description of what functionality can be used with an input assignment can be found in §4.4

D. Description of the target machine.

12. architecture. Type int. This parameter designates the topology of the parallel machine for which you are partitioning. Current capabilities include a hypercube (indicated by a value of 0), and a one-, two- or three-dimensional mesh (indicated by a value of 1, 2 or 3 respectively.) Note that this argument overrides the `ARCHITECTURE` parameter.

13. ndims_tot. Type int. If *architecture* is zero, indicating a hypercube, this value is the number of dimensions in the hypercube.

14. mesh_dims. Type int array of size 3. If *architecture* is 1, 2 or 3, indicating a mesh, the values in this array denote the size of the mesh in each dimension.

15. goal. Type double *. This optional array specifies the desired sizes of the different sets. The total number of sets is implicit in the architectural specifications provided by the preceding three parameters. If a null value is passed for *goal*, the code will try to make each set have the same vertex weight sum. If it is not null, the *goal* array should be as long as the total number of sets. The value in $goal[i]$ (or, for Fortran, $goal(i+1)$) should be the desired sum of vertex weights of vertices assigned to set i. Note that set numbers begin at zero. **Chaco** will try to get as close to this goal as possible, but may not succeed exactly. The sum of all the goals should equal the sum of all the vertex weights, and values should be nonnegative.

Although the default is to make all set sizes equal, there are applications where this may be undesirable. One example would be if you are decomposing a computation among processors of different speeds. All the code in **Chaco** handles this more general case, and should work for any consistent values in *goal*.

E. Partitioning options.

16. global_method. Type int. This argument specifies the global partitioning method and should have a value from 1 and 7. These values are the same as those on the "Global method" menu when running **Chaco** in stand–alone method, as reviewed in §5.4.

17. **local_method.** Type int. This argument specifies the local partitioning method and should have a value of 1 or 2. These values are the same as those on the "Local method" menu when running **Chaco** in stand–alone method, as reviewed in §5.4.
18. **rqi_flag.** Type int. If you requested spectral partitioning and wish to use the multilevel RQI/Symmlq eigensolver, this argument should be set to 1. If you wish instead to use Lanczos, it should be set to 0.
19. **vmax.** Type int. If you are using either the multilevel–KL partitioner, or the multilevel RQI/Symmlq eigensolver, you need to specify when the coarsest graph is small enough. When a coarse graph has no more than *vmax* vertices, the recursive coarsening is finished.
20. **ndims.** Type int. This argument should have a value of 1, 2 or 3 indicating partitioning by bisection, quadrisection or octasection.
21. **eigtol.** Type double. If you are using a spectral method or multilevel–KL, this argument specifies the tolerance you request for the eigensolver. A discussion of an appropriate choice can be found in the description of the `EIGEN_TOLERANCE` parameter in §6.2. Note that this argument overrides the value of the `EIGEN_TOLERANCE` parameter.
22. **seed.** Type long. This is a seed for the random number generator "rand()". Note that it overrides the `RANDOM_SEED` parameter.

8. Changes since Version 1. Version 2.0 of **Chaco** differs from earlier versions in a number of ways. We gratefully acknowledge the helpful suggestions of users who requested greater functionality, critiqued the interface or reported problems. In this section we briefly list the most important differences between this version and its predecessor [10], and direct the interested reader to the relevant sections of this user's guide for more details.

8.1. Enhanced functionality. Version 2.0 of **Chaco** can partition for one-, two- and three-dimensional **mesh topologies**. Earlier versions worked only for hypercubes. There are two aspects to this generalization. First, the code can now partition into an **arbitrary number of sets**. And second, the sets are assigned to processors to maximize locality on either a hypercube or a mesh. The parameter which specifies the topology is `ARCHITECTURE` and is described in §6.8. If you don't care about the mapping to processors, you can simply select a one-dimensional mesh topology and then input the number of sets you require when prompted.

We have added options to the implementation of our multilevel–KL method to include a closely related **multilevel algorithm due to Bui and Jones** [2, 15]. We have also added the ability to prefer **high weight matching** edges as advocated by Karypis and Kumar [16], and **different matching algorithms** to allow the user to tradeoff between randomness and speed. The speed of the multilevel–KL algorithm has been improved by using a **lazy evaluation** technique for initializing values. All these options are discussed in §3.6 and/or §6.5.

The ability to **relax the strict balanced requirement** has been added by the KL_IMBALANCE parameter as discussed in §6.4.

We have added a method from the circuit placement community known as **terminal propagation**. This is a technique for improving the mapping to processors by incorporating additional information in the recursion. It also allows the user to tradeoff between message congestion and communication volume. Details can be found in §4.2.

Several **post–processing algorithms for improving the partitioning and/or mapping** have been added. These techniques work to reduce the number of edges cut (§4.3.1, increase the number of vertices with no edges to other sets (§4.3.2) or improve the mapping of sets to processors (§4.3.3).

Spectral graph algorithms are becoming important tools for a surprisingly wide variety of problems. To facilitate these applications, **Chaco** can now be used to **generate and sort the Fiedler vector** of a graph (without partitioning). This is discussed in §4.1.

We have re-implemented and **improved both speed and robustness** of most of the algorithms in **Chaco**. The speed differences should be most notable in the multilevel RQI/Symmlq eigensolver and in the multilevel–KL partitioning algorithm. An important improvement on the robustness side is the inclusion of a **graceful recovery procedure for all of the Lanczos eigensolvers** in the event they run out of memory. Previously they terminated; they now compute the best readily available approximation to the eigenvector and allow the code to continue. They also contain several added layers of **defense against certain numerical problems** (see §3.3).

8.2. New and modified parameters. **Chaco** contains a variety of parameters which control the functionality, algorithmic details and input and output options. With version 1 it was necessary to recompile the code to change parameter values. This limited the ease with which parameter studies could be conducted. In version 2.0 all **parameters can be modified at runtime**, which significantly increases the usability of the parameters. Details can be found in §6.10.

A number of parameters have been added to the code to control the new functionality sketched above in §8.1. The following **additional parameters** have been added or had their meaning modified. We include a brief discussion and a pointer to the appropriate section of text.

OUTPUT_METRICS The meaning of this parameter has changed to allow for more detailed control of the output (§6.1).

TIME_KERNELS You can now time common numerical kernel operations like matrix-vector multiplication on your machine (§6.2).

PROMPT This parameter allows you to turn off the interactive queries. This makes for cleaner output when you are piping the input from a script. See §6.1.

LANCZOS_CONVERGENCE_MODE Rather than determining convergence based directly on the eigen residual, you may now choose to base convergence decisions on how the partition evolves during the Lanczos iteration (§6.2).

`LANCZOS_SO_PRECISION` To save space, you can run the selective orthogonalization variants of Lanczos in single precision (§6.2).

`RQI_CONVERGENCE_MODE` Rather than determining convergence based directly on the eigen residual, you can now base convergence decisions on how the partition is evolving during RQI.

`COARSE_KL_BOTTOM` In multilevel–KL this parameter forces the invocation of KL on the coarsest graph (§6.5).

`MATCH_TYPE` When performing coarsening this parameter allows you to choose several different maximal matching algorithms with different cost/quality tradeoffs (§6.5).

`ARCHITECTURE` This important new parameter allows you to select to partition for either a hypercube or a mesh parallel machine (§6.7).

`MAKE_VWGTS` This flag will automatically generate weights for vertices that are appropriate for applications involving **parallel matrix–vector multiplication** (§6.8).

`FREE_GRAPH` If you are calling **Chaco** from another code, this flag allows you to free the original graph storage to save space (§6.8).

8.3. Changes to input and output formats. To better handle the input of vertices of high degree, **vertex data can be spread over multiple lines** of the graph input file. The mechanism enabling this is discussed in §5.1.

Version 2.0 of **Chaco** is able to **handle unexpected graph input gracefully** in more instances. For example, self edges and edges with zero weights are now discarded instead of causing the program to halt.

The **output format of the code has been clarified and enhanced** with several additional metrics of partition quality. These are described in §5.5.

Version 2.0 of **Chaco** can **read an existing partition** from a file. It can then refine or evaluate the partition in several ways. Assignment files can now be structured in two different ways to simplify interfacing with other codes. These changes are discussed in §4.4.

Chaco can now be more **easily coupled with other programs**. The calling sequence to do this is described in §7.

Chaco can now perform **multiple runs with a single invocation**. This enables, for example, piping many calculations into a single run of the program.

The look and feel of Chaco has changed in a variety of additional ways. The intent of these changes was to **simplify the user interface** and provide additional information. Examples include the removal of some options from the user menu so as to streamline its use, changing the structure of the input prompts, and clarifying of all phases of output.

8.4. Interfaces to other codes. A *Matlab* front end for **Chaco** has been written by John Gilbert as part of the **meshpart** software (which also contains implementations of other partitioning algorithms). This public domain software can be obtained via anonymous ftp to parcftp.xerox.com in the file /pub/gilbert/meshpart.uu. Chaco has also been interfaced with a number of important scientific and engineering application codes. If you suspect this may be true of the code you are working with, check with us.

Acknowledgements. We appreciate all the constructive feedback provided by earlier users of the code. These include Patrick Ciarlet, Ralf Diekman, Gary Hennigan, Scott Hutchinson, Francoise Lamour, Steve Plimpton, Robert Preis, Padma Raghavan, Ed Rothberg, John Shadid, Barry Smith, Rafael Van Driessche and Peter Van Vleet.

The algorithms implemented in **Chaco** have been influenced by numerous people including Steve Barnard, Thang Bui, John Gilbert, John Lewis, Cindy Phillips, Steve Plimpton, Alex Pothen, Horst Simon, Ray Tuminaro and Rafael Van Driessche.

The development of **Chaco** was supported by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research. The work was performed at Sandia National Laboratories, which is operated for the U.S. Department of Energy under contract number DE-AC04-76DF00789.

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] 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.

[4] 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.

[5] M. Fiedler, *Algebraic connectivity of graphs*, Czechoslovak Math. J., 23 (1973), pp. 298–305.

[6] ———, *A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory*, Czechoslovak Math. J., 25 (1975), pp. 619–633.

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

[8] G. Golub and C. Van Loan, *Matrix Computations, Second Edition*, Johns Hopkins University Press, Baltimore, MD, 1989.

[9] S. Hammond, *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, *The Chaco user's guide, version 1.0*, Tech. Rep. SAND93–2339, Sandia National Laboratories, Albuquerque, NM, October 1993.

[11] ———, *An improved spectral load balancing method*, in Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, March 1993, pp. 953–961.

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

[13] ———, *A multilevel algorithm for partitioning graphs*, in Proc. Supercomputing '95, ACM, December 1995. To appear.

[14] B. Hendrickson, R. Leland, and R. Van Driessche, *Enhancing data locality by using terminal propagation*, in Proc. 29th Hawaii Conf. System Sciences, 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. Leland and B. Hendrickson, *An empirical study of static load balancing algorithms*, in Proc. Scalable High Perf. Comput. Conf., IEEE, May 1994, pp. 682–685.
[19] C. C. Paige and M. A. Saunders, *Solution of sparse indefinite systems of linear equations*, SIAM J. Numer. Anal., 12 (1975), pp. 617–629.
[20] B. Parlett and D. Scott, *The Lanczos algorithm with selective orthogonalization*, Math. Comp., 33 (1979), pp. 217–238.
[21] B. Parlett, H. Simon, and L. Stringer, *Estimating the largest eigenvalue with the Lanczos algorithm*, Math. Comp., 38 (1982), pp. 153–165.
[22] A. Pothen, H. Simon, and K. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal., 11 (1990), pp. 430–452.
[23] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1988.
[24] 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.
[25] 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.
[26] R. Van Driessche and D. Roose, *A spectral algorithm for constrained graph partitioning I: The bisection case*, TW Report 216, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, October 1994.
[27] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.
[28] R. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency, 3 (1991), pp. 457–481.