Graph Partitioning and Parallel Solvers: Has the Emperor No Clothes?*
(Extended Abstract)

Bruce Hendrickson

Sandia National Labs, Albuquerque, NM 87185–1110

Abstract. Sparse matrix-vector multiplication is the kernel for many scientific computations. Parallelizing this operation requires the matrix to be divided among processors. This division is commonly phrased in terms of graph partitioning. Although this abstraction has proved to be very useful, it has significant flaws and limitations. The cost model implicit in this abstraction is only a weak approximation to the true cost of the parallel matrix-vector multiplication. And the graph model is unnecessarily restrictive. This paper will detail the shortcomings of the current paradigm and suggest directions for improvement and further research.

1 Introduction

Over the past several years, a comfortable consensus has settled upon the community concerning the applicability of graph partitioning to the parallelization of explicit methods and iterative solvers. This consensus is largely the result of two happy occurrences. First is the availability of good algorithms and software for partitioning graphs, motivated by parallel computing applications (eg. Chaco [2], METIS [5], JOSTLE [11], PARTY [10] and SCOTCH [9]). Second is the excellent parallel efficiencies which can be obtained when solving differential equations. Clearly, the latter is a consequence of the former, right?

Yes, and no. Although the Emperor may not be completely naked, he is wearing little more than his underwear. The graph partitioning abstraction is seriously flawed, and its uncritical adoption by the community has limited investigation of alternatives. The standard partitioning approach suffers from two kinds of shortcomings. First, it optimizes the wrong metric. And second, it is unnecessarily limiting. These shortcomings will be discussed in detail below. The purpose of this paper is to elucidate these failings, and to suggest some improvements and directions for further research. More generally, I hope to stimulate renewed interest in a set of problems that has been erroneously considered to be solved.

* In Proc. Irregular '98, ©Springer-Verlag.This work was funded by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research and performed at Sandia National Laboratories, operated for the U.S. DOE under contract number DE-AC04-76DP00789.

2 Matrices, Graphs and Grids

Before discussing the problems with graph partitioning in detail, it will be helpful to review the relationship between grids, matrices and graphs. A grid or mesh is the scaffolding upon which a function is decomposed into simple pieces. Standard techniques for solving differential equations represent their solutions as a union of simple functions on the pieces the grid. The grid determines the nonzero structure of the corresponding sparse matrix, but the relationship between the grid and the matrix can be complex. The solution values can be associated with grid points, or with grid regions. Multiple unknowns per grid point lead to block structure in the matrix. A nonzero value in the matrix might occur for each pair of grid points which are connected by an edge. Alternatively, a nonzero might occur if two grid points share a region. More complicated relationships are also possible. But an important point is that the matrices associated with computational grids are usually structurally symmetric. That is, if element $[i, j]$ is nonzero, then so is element $[j, i]$.

The standard graph of a symmetric $n \times n$ matrix has n vertices, and an edge connects vertices i and j if matrix element $[i, j]$ is nonzero. Note that this graph is generally not identical to the grid, although the two are related. Also note that every symmetric matrix has a graph, whether or not it has a corresponding grid.

For parallelizing iterative solvers, three graphs are commonly used.

1. The standard graph of the (symmetric) matrix.
2. The grid.
3. The dual of the grid. That is, each region becomes a vertex and two vertices are connected if their corresponding regions are adjacent to each other

Each of these graphs has its advantages and disadvantages. The graph of the matrix is unambiguous and always available for symmetric matrices. But if the matrix is never explicitly formed, the graph may be difficult to construct. And if there are multiple unknowns per grid point, then the graph of the matrix may be unnecessarily large.

The grid does not suffer from the problem of multiple unknowns, and so can be a smaller representation of the basic structure of the matrix. However, as discussed above, it only approximates the structure of the matrix. Also, if the matrix doesn't come from a grid-based differential equation, then there may be no grid to use.

The dual of the grid has the same advantages and disadvantages as the grid, and its construction requires some nontrivial effort on the part of the user. However, as will be discussed in §4, when the standard graph partitioning approach is used, the dual is often a better model of communication requirements than the grid itself.

3 Parallel Matrix-Vector Multiplication

For sparse matrices, the standard approach to parallelizing matrix-vector multiplication involves dividing the rows of the matrix among the processors. Each processor is responsible for computing the contributions from the matrix values it owns. Consider forming the product $y = Ax$ on p processors. The rows and columns of A are divided into p sets, and the elements of x and y are divided in the same way. As we will discuss below, graph partitioning is only applicable to structurally symmetric matrices, so for the remainder of this section we will assume that A is structurally symmetric. Conceptually, the partitioning can be thought of as a symmetric reordering of the rows and columns of A, and also of x and y, followed by an organization of A into block structure as illustrated below.

$$A = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix}, \tag{1}$$

Processor i begins with segment i of x and computes segment i of y. It has all the necessary elements of x to compute the contribution from its diagonal block A_{ii}. To compute the contribution of off-diagonal blocks, processor i needs some x values from other processors. Specifically, if $A(k,l) \in A_{ij}$ is nonzero, then processor i needs $x(l)$ from processor j. This exchange of elements of x is all the communication required to form the matrix-vector product.

Several important observations are in order. First, the block structure of A can be interpreted in terms of the standard graph of the matrix of A. Since each row of A is a vertex in the graph, the block structure induces a partitioning of the vertices into p sets. Second, the total number of nonzero values in A_{ij} is equal to the number of edges in the graph which connect vertices in partition i to those in partition j. Third, the number of x values that processor j must send to processor i is equal to the number of vertices in partition j which have neighbors in partition i. We will call vertices which have neighbors on other processors *boundary vertices.*

Explicit solvers have exactly the same communication pattern as matrix-vector multiplication. Consequently, all the issues discussed here are relevant to them as well.

It is worth noting that the same row-wise partitioning of the matrix can also be used to efficiently compute $z = A^T w$. This is useful for nonsymmetric matrices since many iterative methods for such matrices require both matrix-vector and matrix-transpose-vector products. Details can be found in [3].

4 Problems with Graph Partitioning

The standard graph partitioning problem addressed by codes like Chaco [2] and METIS [5] is the following. Divide the vertices of the graph into equally sized sets in such a way that the number of edges crossing between sets is minimized. (This formulation can be generalized to consider weights on the vertices and edges, but details are beyond the scope of this paper.) The construction of an appropriate graph is the responsibility of the user, and typically one of the three graphs described in §2 is used. By limiting the number of edges crossing between sets, the off-diagonal blocks of the matrix have few nonzeros. Thus, in matrix terms, graph partitioning methods try to put most of the nonzero values in the diagonal block. This is clearly advantageous for limiting the communication, but unfortunately, the number of cut edges is not the most important thing to minimize. The inappropriateness of this metric are discussed in §4.1, while more general limitations of the graph partitioning model are described in §4.2.

4.1 Problems With Edge Cuts

Minimizing edge cuts has three major flaws.

A. Although widely assumed to be the case, edge cuts are not proportional to the total communication volume. Obviously, this depends on the graph presented to the partitioner. If the graph is the standard graph of the matrix then, as discussed in 3, the communication volume is instead proportional to the number of boundary vertices. It is for this reason that the dual of the grid is often used as the graph to partition. Edges in the dual roughly correspond to pairs of boundary vertices in the grid. But the dual may be difficult to construct, and is not defined for general matrices. It would be better to have a partitioner that works directly with the structure of the matrix and tries to minimize an accurate model of the communication cost.

B. Sending a message on a parallel computer requires some startup time (or latency) and some time proportional to the length of the message. Graph partitioning approaches try to (approximately) minimize the total volume, but not the total number of messages. Unfortunately, in all but the largest problems, latencies can be more important than volume.

C. The performance of a parallel application is limited by the slowest processor. Even if all the computational work is well balanced, the communication effort might not be. Graph partitioning algorithms try to minimize the total communication cost. However, optimal performance will be obtained by instead minimizing the maximum communication cost among all processors. In some applications it is best to minimize the sum of the computation and communication time, so imbalance in one can be offset in the other. The standard graph partitioning approaches don't address these issues.

With these problems, why has the standard graph partitioning approach proved so successful for the parallel solution of differential equations? There are two reasons. First, grid points generally have only a small number of neighbors. So cutting a minimal number of edges ensures that the number of boundary vertices is within a small multiple of the optimal. This isn't true of more general matrices. Also, well structured grids have good partitions. If the number of processors is kept fixed while the problem size increases, latencies become unimportant and the computational work for a matrix-vector multiplication should grow linearly with n, while the communication volume should only grow as $n^{2/3}$ in 3D and $n^{1/2}$ in 2D. Thus, very large problems should exhibit excellent scalability, even if the partition is nonoptimal. For more general matrices, this inherent scalability doesn't hold true, and the quality of the partition matters much more.

4.2 Limitations of Graph partitioning

Besides minimizing the wrong objective function, the standard graph partitioning approach suffers from several limitations, all reflective of a lack of expressibility of the model.

A. Only square, symmetric matrices. In the standard graph model, a single vertex represents both a row and a column of a matrix. If the matrix is not square, then the model doesn't apply. Also, in the standard model the single edge between i and j can't distinguish a nonzero in location $(i.j)$ from one in location (j, i). So even square matrices aren't handled well if they are not symmetric.

B. Only symmetric partitions. Since a single vertex represents both a row and a column, the partition of the rows is forced to be identical to the partition of the columns. This is equivalent to forcing the partition of y to be identical to the partition of x in $y = Ax$. If the matrix is symmetric, then this restriction simplifies the operation of an iterative solver. However, for nonsymmetric solvers, this restriction is unnecessary.

C. Ignores the application of the preconditioner. Preconditioning can dramatically improve the performance and robustness of iterative solvers. In a preconditioned iterative method, the matrix-vector product is only one part of a larger operation. For some preconditioners (eg. Jacobi) optimal performance of $y = Ax$ will lead to good overall performance. But for more complex preconditioners it is better to consider the full calculation instead of just the matrix-vector product. Unfortunately, the standard graph model has no capacity to consider the larger problem.

5 A Better Combinatorial Model

Some of the limitations discussed in §4.2 can be addressed by a more expressive model proposed by Kolda and Hendrickson [3, 4, 8]. The model uses a *bipartite* graph to represent the matrix. In this graph, there is a vertex for each row and another vertex for each column. An edge connects row vertex i to column vertex j if there is a nonzero in matrix location $[i, j]$. The structure of nonsymmetric and nonsquare matrices can be unambiguously described in this way, which addresses limitation (**A**).

The row vertices and column vertices can now each be partitioned into p sets, which corresponds to finding a $p \times p$ block structure of the matrix. An edge between a row vertex in partition i and a column vertex in partition j corresponds to a nonzero value in off-diagonal block A_{ij}. By minimizing such edges, most of the nonzero values will be in the diagonal blocks. The same matrix-vector multiplication algorithm sketched above can now be applied. The partition of x will correspond to the column partition of A, while the partition of y reflects A's row partition. The freedom to have different row and column partitions removes limitation (**B**).

The freedom to decouple row and column partitions has an additional advantage. Consider the case where two matrices are involved, so $y = BAx$. In [3], Hendrickson and Kolda show that the row partition can be used to get good performance in the application of A, while the column partition can optimize the application of B. So if a preconditioner is explicitly formed as a matrix (eg. approximate inverse preconditioners), then the bipartite partitioning problem can model the full iteration. This is a partial resolution of limitation (**C**).

Although the bipartite model is more expressive than the standard model, the algorithms in [3] still optimize the flawed metric of edge cuts.

6 Conclusions and Directions for Further Research

Despite the success of the standard graph partitioning approach to many parallel computing applications, the methodology is flawed. Graph partitioning minimizes the wrong metric, and the graph model of matrices is unnecessarily limiting. A bipartite graph representation resolves some of the limitations, but a number of important open questions remain.

1. **More accurate metric.** The principle shortcoming of standard graph partitioning is that the quantity being minimized does not directly reflect the communication cost in the application. Unfortunately, minimizing a more appropriate metric is challenging. Boundary vertices, the correct measure of communication volume, are harder to optimize than edge cuts. A good refinement algorithm which works on this quantity would be a significant advance. More generally, the true communication cost in a parallel matrix-vector multiplication operation is more complex than just total volume. The number of messages can be at least as important as the volume, but graph partitioning algorithms seem ill suited to addressing this quantity directly. Similarly, since the processors tend to stay synchronized in an iterative solver, the maximum communication load over processors is more important than the total. These issues have received insufficient attention.
2. **Matrix and preconditioner**. As discussed in §4.2, the matrix-vector multiplication is generally only a piece of a larger operation. Methodologies that optimize a full step of the iterative solver would be better. The bipartite model described in §5 is an improvement, but much work remains to be done.
3. **Multicriteria partitioning**. In many parallel applications there are several computational kernels with different load balancing needs. In many of these

situations, the individual kernels are synchronized so for peak performance each kernel must be load balanced. A simple example is the application of boundary conditions which only occurs on the surface of a mesh. A good partition for such a problem will divide the problem in such a way that each kernel is balanced. In the boundary condition example, each processor should own a portion of the surface of the mesh, and also of the full volumetric mesh. This important problem has not yet received sufficient attention, but some researchers are beginning to address it [7].

4. **Partitioning for domain decomposition**. Domain decomposition is a numerical technique in which a large grid is broken into smaller pieces. The solver works on individual subdomains first, and then couples them together. The properties of a good decomposition are not entirely clear, and they depend upon the details of the solution technique. But they are almost certainly not identical to the criteria for matrix-vector multiplication. For instance, Farhat, et al. [1] argue that the domains must have good aspect ratios (eg. not be long and skinny). It can also be important that subdomains are connected, even though the best partitions for matrix-vector multiplication needn't be. For the most part, practitioners of domain decomposition have made due with partitioning algorithms developed for other purposes, with perhaps some minor perturbations at the end. But a concerted effort to devise schemes which meet the need of this community could lead to significant advances.
5. **Parallel partitioning**. Most of the work on parallel partitioning has been done in the context of dynamic load balancing. But several trends are increasing the need for this capability. First is the interest in very large meshes, which won't easily fit on a sequential machine and so must be partitioned in parallel. Second, for a more subtle reason, is the growing interest in heterogeneous parallel architectures. Generally, partitioning is performed as a preprocessing step in which the user specifies the number of processors the problem will run on. With heterogeneous parallel machines, the number of processors is insufficient – the partitioner should also know their relative speeds and memory sizes. A user will want to run on whatever processors happen to be idle when the job is ready, so it is impossible to provide this information to a partitioner in advance. A better solution is to partition on the parallel machine when the job is initiated. A number of parallel partitioners have been implemented including Jostle [11] and ParMETIS [6]. This is an active area of research

Despite the general feeling that partitioning is a mature area, there are a number of open problems and many opportunities for significant advances in the state of the art.

Acknowledgements

The ideas in this paper have been influenced by many discussions with Rob Leland and Tammy Kolda. I have also benefited from conversations with Ray

Tuminaro, David Day, Alex Pothen and Michele Benzi.

References

1. C. Farhat, N. Maman and G. Brown, *Mesh partitioning for implicit computation via domain decomposition: impact and optimization of the subdomain aspect ratio*, Int. J. Num. Meth. Engrg. 38 (1995), pp. 989–1000.
2. B. Hendrickson and R. Leland, *The Chaco user's guide, version 2.0*, Tech. Rep. SAND95-2344, Sandia Natl. Lab., Albuquerque, NM, 87185, 1995.
3. B. Hendrickson and T. G. Kolda, *Parallel algorithms for nonsymmetric iterative solvers*. In preparation.
4. ———, *Partitioning Sparse Rectangular Matrices for Parallel Computation of Ax and $A^T v$*. In *Lecture Notes in Computer Science*, Springer-Verlag, 1998. Proc. PARA'98. To appear.
5. G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Rep. 95-035, Dept. Computer Science, Univ. Minnesota, Minneapolis, MN 55455, 1995.
6. ———, *Parallel multilevel graph partitioning*, Tech. Rep. 95-036, Dept. Computer Science, Univ. Minnesota, Minneapolis, MN 55455, 1995.
7. ———, *Multilevel algorithms for multi-constraint graph partitioning*, Tech. Rep. 98-019, Dept. Computer Science, Univ. Minnesota, Minneapolis, MN 55455, 1998.
8. T. G. Kolda, *Partitioning sparse rectangular matrices for parallel processing*. In *Proc. Irregular'98*, 1998.
9. F. Pellegrini, *SCOTCH 3.1 user's guide*, Tech. Rep. 1137-96, Laboratoire Bordelais de Recherche en Informatique, Universite Bordeaux, France, 1996.
10. R. Preis, R. Diekmann, *The PARTY Partitioning-Library, User Guide - Version 1.1*, Tech. Rep. tr-rsfb-96-024, University of Paderborn, Paderborn, Germany, 1996.
11. C. Walshaw, M. Cross, and M. Everett, *Mesh partitioning and load-balancing for distributed memory parallel systems*, in Proc. Parallel & Distributed Computing for Computational Mechanics, Lochinver, Scotland, 1997, B. Topping, ed., 1998.