ELSEVIER

Parallel Computing 26 (2000) 1519–1534

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Graph partitioning models for parallel computing [☆]

Bruce Hendrickson [a,*], Tamara G. Kolda [b]

[a] *Parallel Computing Sciences Department, Sandia National Labs, Albuquerque, NM 87185-1110, USA*

[b] *Computational Sciences and Mathematics Research Department, Sandia National Labs, Livermore, CA 94551-9214, USA*

Received 12 February 1999; received in revised form 28 September 1999; accepted 28 September 1999

Abstract

Calculations can naturally be described as graphs in which vertices represent computation and edges reflect data dependencies. By partitioning the vertices of a graph, the calculation can be divided among processors of a parallel computer. However, the standard methodology for graph partitioning minimizes the wrong metric and lacks expressibility. We survey several recently proposed alternatives and discuss their relative merits. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Graph partitioning; Hypergraph partitioning; Parallel computing

1. Introduction

Graphs are widely used to describe the data dependencies within a computation. Recall that a *graph*, $G = (V, E)$, consists of a set of *vertices*, $V = \{v_1, v_2, \ldots, v_n\}$, and a set of pairwise relationships, $E \subset V \times V$, called *edges*. If $(v_i, v_j) \in E$, then we say that vertices v_i and v_j are *neighbors*. For our purposes, the vertices of the graph represent units of computation, and the edges encode data dependencies. Sometimes

[☆] This work was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, US Department of Energy, under contracts DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation and DE-AC04-94AL85000 with Sandia Corporation.

[*] Corresponding author.

E-mail addresses: bah@cs.sandia.gov (B. Hendrickson), tgkolda@sandia.gov (T.G. Kolda).

PII: S 0 1 6 7 - 8 1 9 1 (0 0) 0 0 0 4 8 - X

it is appropriate to associate *weights* with the nodes and/or edges of the graph to indicate the amount of work and/or data, respectively.

For example, differential equations are usually solved numerically on a grid. During each iteration in the process towards a solution, all the grid points are updated using neighboring values in the mesh. In Fig. 1 we show the mesh and an associated data dependency graph for a symmetric seven-point stencil. Here, each vertex in the graph at right represents the computation to update the associated point on the grid. Each vertex has edges connecting it to the vertices from which it needs information. Outputs from one iteration serve as inputs for the next.

Once we have a graph model of a computation, *graph partitioning* can be used to determine how to divide the work and data for an efficient parallel computation. Our objectives, stated loosely, are to evenly distribute the computations over p processors by partitioning the vertices into p equally weighted sets while minimizing interprocessor communication which is represented by edges crossing between partitions.

It is this simple relationship between graphs and computations that explains the ubiquity of graph partitioning in parallel computing. Graph partitioning is universally employed in the parallelization of calculations on unstructured grids including finite element, finite difference, and finite volume techniques using both explicit and implicit methods. It is used in the parallelization of matrix–vector multiplication for all types of iterative solvers. It is also used to parallelize neural net simulations, particle calculations, circuit simulations, and a variety of other computations.

Until recently only the *standard graph partitioning approach* has been employed. The standard approach is to model the problem using a graph as described above and partition the vertices of the graph into equally weighted sets so that the weight of the edges crossing between sets is minimized. Well-known software packages such as Chaco [13] and METIS [20] can be used for this purpose. Note that the graph partitioning problem is NP-hard [9], so these tools merely apply heuristics to generate approximate solutions.

Unfortunately, the standard graph partitioning approach has several significant shortcomings that are discussed in detail in Section 2. The edge cut metric that it tries to minimize is, at best, an imperfect model of communication in a parallel computation. The model also suffers from a lack of expressibility that limits the applications it can address.

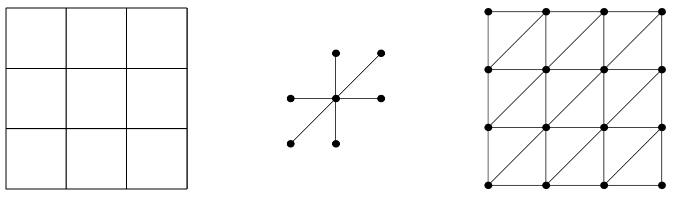

Fig. 1. Grid, stencil, and graph.

This paper extends and elaborates upon Hendrickson's critique of the standard partitioning model in [10]. Whereas Hendrickson restricted his concerns to matrix–vector products, in the current paper we show that the same issues plague virtually all applications of graph partitioning to parallel computation. In Section 3, we survey some recent work on alternative models that address some of the limitations of the standard approach. We follow with a brief discussion of algorithms in Section 4, and suggest some fertile areas for further research in Section 5.

2. Shortcomings of the standard graph partitioning approach

We discuss several shortcomings of the standard graph partitioning approach. We begin with flaws associated with using the edge cut metric (Section 2.1) and continue with limitations of the standard graph model (Section 2.2).

2.1. Flaws of the edge cut metric

Minimizing edge cuts has several major flaws. First, although it is not widely acknowledged, edge cuts are *not* proportional to the total communication volume, as illustrated in Fig. 2. The ovals correspond to different processors among which the vertices of the graph are partitioned. Assume that each edge has a weight of two corresponding to one unit of data being communicated in each direction, so the weight of the cut edges is 10. However, observe that the data from node v_2 on processor P_1 need only be communicated *once* to processor P_2; similarly with nodes v_4 and v_7. Thus, the actual communication volume is only seven. In general, the edge cut metric does not recognize that two or more edges may represent the same information flow, so it overcounts the true volume of communication.

Second, the time to send a message on a parallel computer is a function of the *latency* (or *start-up time*) as well as the size of the message. As has been observed by a number of researchers, graph partitioning approaches try to (approximately) minimize the total volume but not the total number of messages. Depending on the

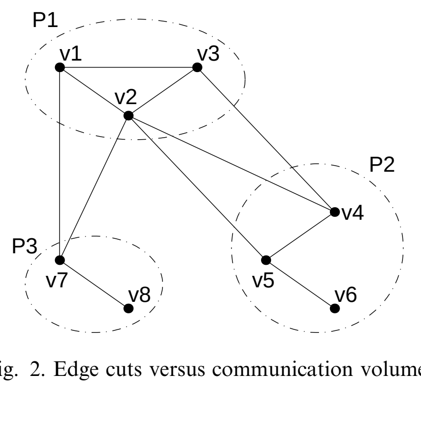

Fig. 2. Edge cuts versus communication volume.

machine architecture and problem size, message latencies can be more important than message volume.

Third, the performance of a parallel application is generally limited by the slowest processor. Even if the computational work is well balanced, the communication effort might not be. Rather than minimizing the total communication volume or even the total number of messages, we may instead wish to minimize the maximum volume and/or number of messages handled by any *single* processor. As several researchers have noted, the standard edge cuts measure does not encapsulate this type of objective.

Last, on many architectures the time to send a message depends upon the distance between the sending and receiving processors. Geographic distance is not the issue here, but rather the number of switches the message is routed through. Although most modern machines have some form of cut-through or worm hole routing that enables a single message to travel quickly between distant processors, the communication network is usually handling many messages simultaneously. A message between distant processors ties up many wires that cannot be used by other messages. To avoid message contention and improve the overall throughput of the message traffic, it is preferable to have communication restricted to nearby processors. For the problem illustrated in Fig. 2, on a one-dimensional row of processors, the layout P_3–P_1–P_2 would be preferable to P_1–P_2–P_3.

In actuality, we are interested in all of these metrics to varying degrees, depending on how they affect the overall speed of the application. We likely want to minimize an objective function with several components (e.g., total volume *and* total number of messages), weighted to reflect the importance of each measure. In even more complicated settings, we may wish to balance the sum of the computational and communications work on each processor while minimizing these combined objectives.

Despite the limitations of the edge cut measure, the standard partitioning approach has proved successful for the parallel solution of differential equations and grid-based problems in general for several reasons. First, grid points usually have only a small number of neighbors, so the number of edge cuts is within a small multiple of the actual communication volume. This is not true of more general problems with more complex data dependencies. Second, computational grids generally exhibit a high degree of geometric locality which ensures that good partitions exist [29]. If the grid size, n, is increased while the number of processors is held fixed, the communication volume grows only as $n^{2/3}$ in three dimensions and $n^{1/2}$ in two dimensions. Similarly, geometric locality tends to limit the number of messages each processor needs to send. Last, the communication volume per processor is fairly evenly distributed since there usually is not an enormous difference in the size of the boundary of each piece of the grid. For all these reasons, large grid computations tend to be limited by computational performance, so the details of the communication (and hence the partition) are not critical. For other applications with more complex dependency patterns, the quality of the partition can have a much more dramatic impact on overall performance.

2.2. Limitations of the standard graph model

Besides minimizing the wrong objective function, the standard graph partitioning approach suffers from limitations due to the lack of expressibility in the model.

One limitation of the undirected graph model is that it can only express symmetric data dependencies. For example, the graph associated with a symmetric matrix is shown in Fig. 3. For the computation $y = Ax$, vertex v_i is associated with the computation of the inner product between row i of the matrix A and the vector x. Observe that the edge between node v_1 and v_2 symbolizes a symmetric dependency: v_1 needs x_2, and v_2 needs x_1.

However, if the matrix is square but unsymmetric, then the dependencies are unsymmetric as well: v_1 might need x_2, while v_2 does not need x_1. This situation can be easily represented in a *directed* graph, but not in the standard model. In a directed graph, edges are directed *from* the data producing vertex *to* the data consuming vertex. There are two work-arounds to make the standard model 'fit' unsymmetric dependencies. The first is to convert the directed edges to undirected edges. The second is a slight extension of the first; an edge that represents only a one-way communication gets a weight of one, and an edge that represents two-way communication gets a weight of two.

Unsymmetric dependencies show up in other settings as well. For example, flow calculations often involve unsymmetric stencils as depicted in Fig. 4.

Another limitation of the symmetric model is that it forces the partition of the input and output data to be identical. This is often desirable, particularly when the

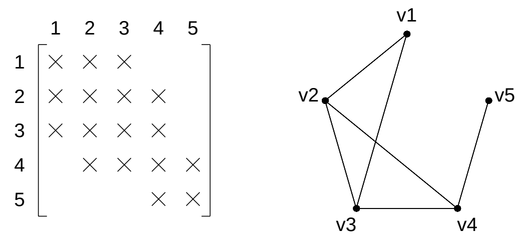

Fig. 3. Graph of a symmetric matrix.

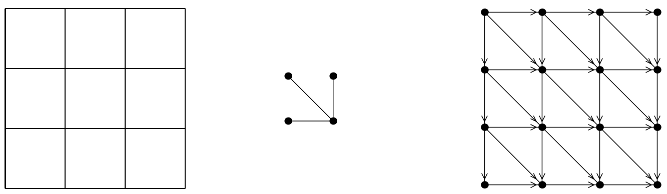

Fig. 4. Grid, stencil, and directed graph.

same calculation is performed over and over and the output from one computation is the input to the next. But in many situations this is an unnecessary restriction. For instance, the standard model generates identical partitions of x and y when computing $y = Ax$ for a square matrix. For unsymmetric matrices, communication may be reduced by allowing the two partitions to differ. For example, the input x may be the result of a previous two-part operation that first computes $y = Ax$ and then $z = A^{T}y$; this effectively maps from x-space to y-space and back to x-space. (The data layout and communication for application of A and A^{T} is identical; see [12]).

A third limitation of the standard model is that it assumes that the input and output of the calculation are the same size, which may not be the case. For example, when A is rectangular in the calculation of $y = Ax$, the x- and y-spaces are of different dimensions. The standard model handles symmetric matrix–vector multiplication ($y = Ax$) by having a single vertex v_i represent *both* x_i and y_i. When the matrix is not square, x and y are of different lengths, and the standard model is inapplicable.

Finally, even within the general framework of calculations that are repeated over and over again, it is common for the calculation to consist of several distinct phases. Examples include the application of a matrix *and* a preconditioner in an iterative method, solving a differential equation *and* applying boundary conditions, simulating different phenomena in a multi-physics calculation, and advancing a grid *and* detecting contacts in a transient dynamics computation. The union of multiple phases cannot generally be described via an undirected graph. As we see in the next section, some alternatives to the standard model retain its basic simplicity while handling some of these more complex situations.

3. Alternative graph partitioning models

Some of the shortcomings of the standard graph partitioning model can be addressed by using recently developed alternatives. We describe four such non-standard models below.

3.1. A bipartite graph model

As we noted in Section 2.2, the standard model using an undirected graph can only encode symmetric data dependencies and symmetric partitions. These limitations are a particular problem for iterative solvers on unsymmetric or non-square matrices. When using a preconditioner, the inability of the standard model to capture multiple phase calculations is also a problem. In [11,12,24], Hendrickson and Kolda propose a bipartite graph model for describing matrix–vector multiplication that addresses some of these shortcomings. The bipartite model can also be applied to other problems involving unsymmetric dependencies and multiple phases.

A *bipartite* graph, $G = (V_1, V_2, E)$, is a special type of graph in which the vertices are divided into two disjoint subsets, V_1 and V_2, and $E \subset V_1 \times V_2$. So, no edge connects two vertices in the same subset; instead, all the edges cross between V_1 and V_2.

This bipartite graph representation is most useful when the initial tasks are logically distinct from the final tasks, such as in the transfer between phases of the multi-phase calculations described in Section 2.2. An important example is matrix–vector multiplication with non-square matrices. Fig. 5 shows the bipartite graph representation of a rectangular matrix. The sets V_1 and V_2 correspond to the row and column vertices, respectively. Each row vertex in V_1 is weighted with the number of non-zeros in its row; e.g., row vertex r_4 has a weight of one. This weighting reflects the computational work required in the matrix–vector product. Whichever processor owns vertex r_i will own the piece y_i of the resulting solution vector $y = Ax$. The partitioning of the column vertices (V_2) affects the layout of the input vector, x. The column vertices may be left unweighted so that x may be partitioned in the optimal way to minimize edge cuts. Better yet, the column vertices may be weighted to distribute the computation of another operation on the input data such as level-1 BLAS operations or multiplication by another matrix such as a preconditioner in an iterative method.

The bipartite graph model is useful principally where the standard model fails to be a good representation, and it has three main advantages. First, it can encode a class of problems that the standard graph model cannot. Specifically, the initial (or input) vertices can be different from the final (or output) vertices. Second, even if the initial vertices are identical to the final vertices, the bipartite model allows the initial partition to differ from the final partition. It achieves this by representing each vertex twice, once as an initial vertex and once as a final vertex. This freedom can allow for a reduction in communication. However, in many applications a symmetric partition is preferable, and this model cannot naturally provide that. Third, by partitioning both the initial and the final vertices, it can ensure load balance in two separate operations, as mentioned above.

Although the bipartite model has expressibility that the standard model lacks, the algorithms in [12] still optimize the flawed metric of edge cuts (as well as sharing the other problems of the standard model described in Section 2.1). As we see in the Section 3.2, this problem can be addressed by optimizing a graph quantity other than cut edges.

Although the bipartite model is good for describing two computational operations, it is not able to accurately encode more. One possible generalization is to use a k-partite graph in which the first set of vertices is connected to a second set, which is

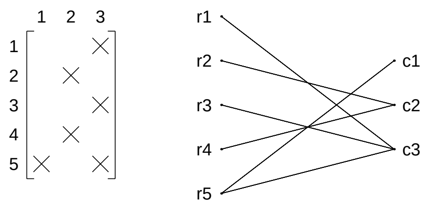

Fig. 5. Rectangular matrix and bipartite graph.

connected to a third set, and so on. An alternative is the multi-constraint methodology described in Section 3.3.

3.2. A hypergraph model

Edge cuts are not equal to communication volume, as illustrated in Fig. 2. In the figure, vertex v_2 on processor P_1, for example, has two edges connecting to vertices on processor P_2, but v_2 need only be communicated once. The true communication volume is not a function of the number of edges being cut, but rather of the sum of the number of processors to which each vertex has connections. More formally, the total communication volume is $\sum_i b_i$, where b_i is the number of external partitions in which vertex v_i has neighbors. We call this quantity the *boundary cut* of a partition. The observation that boundary cuts are the more appropriate metric was made in [10]. Minimizing boundary cuts is a non-traditional graph partitioning problem, but it can be addressed using the same basic algorithmic tools that have been developed for other partitioning variants. These ideas have been instantiated in the current version of METIS [18]. Boundary cuts can also be employed in the bipartite graph model from Section 3.1.

A more elegant expression of this metric is in the hypergraph model proposed by Çatalyürek, Aykanat, Pinar, and Pinar [3,5,26]. A *hypergraph* is a generalization of a graph in which edges can include more than two vertices. A hypergraph, $G = (V, H)$, consists of a set of vertices, V, and a set of *hyperedges*, H. Each hyperedge comprises a subset of vertices. Note that graphs are special cases of hypergraphs in which each hyperedge contains only two vertices. Hyperedges provide an alternative representation of the data dependencies. The partitioning problem is now to divide the vertices into equally weighted sets so that few hyperedges cross between partitions.

The hypergraph model has broader applicability than the standard approach. But even for problems that can be described with the standard model, the hypergraph model is preferable since it correctly minimizes the communication volume. To see this, consider a computation like the one in Fig. 2 that can be described by a standard undirected graph $G = (V, E)$. Now construct an equivalent hypergraph (V, H) with $|V|$ hyperedges. Each vertex, $v_i \in G$, corresponds to a hyperedge h_i consisting of v_i and all its neighbors in G. A hyperedge represents the entities that either produce or that consume a piece of data. When the vertices are partitioned among processors, that piece of data must be communicated from the processor that produced it to all those that consume it. Thus, the communication associated with a hyperedge is one less than the number of processors its constituent vertices are partitioned among. (This corresponds to the boundary cut value from the discussion above.) By partitioning the hypergraph so that hyperedges are split among as few processors as possible, the model correctly minimizes communication volume.

In [5], Çatalyürek and Aykanat apply this model to symmetric matrix–vector multiplication. For a set of highly unstructured matrices from linear programming problems, they report that the hypergraph model reduces communication by over

30% on an average over the standard partitioning approach. However, for reasons discussed in Section 2.1, the gains are more modest for matrices from grid calculations, generally less than 10% [1].

In addition to resolving the principle problem of the edge cut metric, the hypergraph approach is more expressive than the standard model. It can encode problems with unsymmetric dependencies and even problems in which the initial vertices differ from the final vertices. In Fig. 6, we show two different sketches of a hypergraph relating the data dependencies for the rectangular matrix–vector multiply from Fig. 5. For example, hyperedge h_2 contains all the vertices that need x_2, i.e., $\{v_2, v_4\}$. In the left figure, the hyperedges are illustrated by the ovals. In the right figure, the vertices are on one side and hyperedges on the other, and each hyperedge is connected to the vertices that comprise it. We include this second hypergraph representation to underscore the one-to-one relationship between hypergraphs and bipartite graphs. The hypergraph partitioning model is closely related to the bipartite model from Section 3.1, but the partitioning objectives are different.

The guiding principle in the construction of a hypergraph is that each hyperedge contains the set of vertices that generate or need some particular piece of data. This principle applies equally well to the case when dependencies are uni-directional, and it continues to correctly model the communication volume. However, there is a subtle requirement that the data are produced by one of the vertices that depends on it. For example, in Fig. 6 we assume that the data associated with hyperedge h_1 live on the processor that owns vertex v_5. If that is not the case for some reason, e.g., h_1 is the output of v_1, then h_1 should also include its producing vertex, e.g., v_1.

In this way, the hypergraph model can be used even in cases where the input and output data partitions are not identical, although it is perhaps not as natural as the bipartite model in this case. We simply find the best partition for the computation nodes using a hypergraph partitioner, and this yields a partition of the output data. Then, rather than assuming the input data have the same partition as the output data, we can calculate the optimal input data partitioning as an assignment problem. Therefore, the hypergraph model can be an alternative to the bipartite model when we are only encoding one operation; however, the bipartite (or k-partite) models are still best when encoding multi-step operations.

In summary, we find the hypergraph model to be uniformly superior to the standard model. It is also an attractive alternative to the bipartite model for

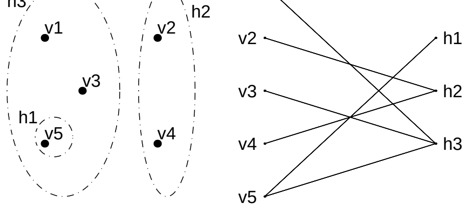

Fig. 6. Two hypergraph representations.

unsymmetric problems when only one operation is being encoded. However, the bipartite (or k-partite) models are still more powerful when encoding multi-phase operations. This is particularly true when the bipartite model minimizes the boundary cut value as discussed above.

3.3. Multi-constraint and multi-objective partitioning

The bipartite model from Section 3.1 is able to describe some types of multi-phase calculations. An alternative approach is the multi-constraint partitioning model of Karypis and Kumar [22]. Strictly speaking, the multi-constraint approach is not an alternative to other models but rather an augmentation. In the multi-constraint model, each vertex is assigned a vector of k weights that represent the work associated with that vertex in each of k computational phases. The goal is now to partition the vertices of that graph in such a way that communication is minimized and that each of the k weights is balanced. In this way, each phase of the computation will be load balanced. The edges in the graph represent data dependencies in all the computational phases.

This is a fairly general model. For instance, when solving a differential equation *and* also applying boundary conditions, each vertex can have two weights. The first weight reflects the work required by a grid point in the solver, and the second can encode the work required for applying the boundary condition. For vertices not on the boundary, the value of the second weight is zero. A successful partitioning for this problem ensures that the equation solver is balanced in such a way that each processor has an equal fraction of the work associated with the boundary condition. An example of the utility of this model is provided by Stiller, Boryczko and Nagle [28]. They use the multi-objective model to obtain a partition that balances the work associated with each of the grids in a multi-grid solver which leads to significant improvement in the parallel performance of their multi-grid code.

The multi-constraint model can encode the bipartite (and k-partite) approaches as a special case. Given a bipartite graph $G = (V_1, V_2, E)$, an equivalent multi-constraint model would have a set of vertices $V = V_1 \cup V_2$, and edges identical to those in the bipartite graph. Each vertex would be assigned two weights, one for the phase modeled by V_1 and the second for the phase modeled by V_2. Hence, each vertex would have one of its weights set to zero. More generally, the multi-constraint model can encode multiple phases with distinct vertices via a model in which it includes the union of all vertices in all phases.

As originally proposed by Karypis and Kumar, the multi-constraint model attempts to minimize edge cuts, but this is an unnecessary restriction. Hyperedges could be used or, equivalently, the boundary cut value from Section 3.2.

A related model is the multi-objective approach of Schloegel, Karypis and Kumar [27]. This model attempts to address the common situation in which a partition should simultaneously minimize several different cost functions. To achieve this each edge is given a vector of j weights, each of which reflects one of the j different cost functions. The goal of the partitioning is to balance the vertex weights in such a way that each of the cost functions is kept small. This model is well suited to multi-phase computations in which different communication patterns arise in different phases. This multi-objective approach can be combined with the multi-criteria methodology to produce a still more encompassing model.

An important limitation of the multi-objective model is that its objective functions are sums of edge weights. As discussed above, edge cuts are only a crude model of communication cost. And other desirable goals, like minimizing the total number of messages, are not amenable to this formalism.

Although the generality of the multi-objective and multi-criteria models is attractive, the associated partitioning problems are quite difficult. When simpler models can be applied, they are usually easier to work with.

3.4. Skewed partitioning

Yet another alternative to the standard partitioning model is the *skewed* partitioning approach developed by Pellegrini [25] and Hendrickson et al. [15]. As with the multi-constraint model, skewed partitioning is really an augmentation of any of the other graph partitioning models rather than a true alternative. In the skewed model, each vertex is allowed to have a set of k preference values expressing its respective desire to be in each of the k sets. When determining how to partition the vertices, these preference values are considered along with the metrics of communication cost.

Preference values can be used in several different ways to achieve different objectives. In dynamic load balancing it is desirable that the new partition be similar to the existing one to limit the amount of data that needs to be moved. This can be encoded in the preference values by giving each datum a preference to remain in its current partition [31]. The magnitude of the preference values can be adjusted to trade-off between partition quality and reduction in data movement.

Another use for preference values is to encourage communicating objects to be assigned to architecturally close processors to reduce message congestion [15,25]. Assume we are partitioning for p processors by recursive application of a k-way partitioner. After the first partition, the graph is divided into k parts that are assigned to k portions of the parallel machine, but the assignment of individual vertices within each part is not yet completed. When doing subsequent partitions, preference values can be used to guide the assignments to processors to encourage neighboring vertices to be near each other in the parallel machine. A vertex in initial partition i that has neighbors in partition j has preference to be assigned to a processor that is near the jth portion of the machine. In this way, the partitioning step is coupled with the problem of assigning partitions to processors. The result is a partition that exhibits better message locality. As before, the magnitude of the preferences can be altered to trade-off between partition quality and message locality.

This same idea was developed independently in the circuit placement community to place circuit elements on a chip with short overall wire lengths [7]. Several algorithms for this problem have been devised including multi-level and spectral approaches [15].

4. Partitioning algorithms

The different graph partitioning models reviewed in Section 3 are only viable if efficient and effective algorithms can be developed to partition them. Fortunately, the *multi-level* paradigm for partitioning has proven to be quite robust and general. The multi-level approach was devised independently by several researchers in the early 1990s [2,6,14] and popularized by the Chaco [13] and METIS [20] partitioning tools. The basic idea is quite simple. A large graph is approximated by a sequence of smaller and smaller graphs. The smallest graph is partitioned using any suitable algorithm. This partition is then propagated back through the sequence of larger and larger graphs, being refined along the way.

Adapting the multi-level approach to a particular partitioning problem requires the following tools:

1. A method for generating a sequence of smaller graphs that preserve the essential properties of the original.
2. An algorithm for partitioning the smallest graph.
3. A refinement technique for improving the partition as it is propagated back up to the original graph.

These tools are generally straightforward to devise; however, the precise details of these tools require some attention to the nature of the partitioning problem being addressed. The generation of smaller graphs is typically done with some kind of edge contraction scheme. Any existing algorithm that handles weights on edges and vertices can be used to partition the smallest graph. The refinement often involves a greedy algorithm in the spirit of Kernighan–Lin [23].

Following the multi-level paradigm, efficient and effective partitioners have been developed for partitioning graphs to minimize edge cuts [2,14], minimize vertex cuts [16], and perform multi-constraint partitioning [22]. The same approach has been successfully used to partition hypergraphs to minimize cut hyperedges [4,6,19] and to partition bipartite graphs [12]. The flexibility of the technique makes it well suited to address a range of different partitioning models and metrics.

5. Conclusions and directions for further research

In many respects, those of us working in the partitioning field have been fortunate. The dominant application for our algorithms and tools has been differential equation solvers. Whether solved implicitly or explicitly, these applications produce dependency graphs that are fairly easy to partition, and large problems are computation rather than communication bound. The applications achieved good parallel performance despite the limitations of our approaches.

Different applications are now becoming common that are much more sensitive to partition quality. Challenging partitioning problems that arise from interior point methods for linear programming, least squares problems, circuit simulation, truncated singular value computations for latent semantic indexing in information retrieval, and other applications are revealing the limitations of our traditional approaches. The standard graph partitioning methodology optimizes an inappropriate quantity, and its expressibility is too limited to address some important classes of applications.

We surveyed several alternative models that address some of the problems with the standard methodology. The bipartite model and the hypergraph model can both handle unsymmetric dependencies. The hypergraph approach correctly encodes communication volume, while the bipartite model and its k-partite generalization have the advantage of being able to represent multi-phase calculations. The multi-constraint and multi-objective approaches offer alternative ways to represent multiple phases, while the skewed partitioning model provides a mechanism for including extra information in a partitioning problem. However, these new models only start to address the problems detailed in Section 2. A number of important open problems remain, including the following:

(1) *Partitioning for alternate objectives or multi-objectives.* As discussed in Section 3.3, the work of Schloegel et al. [27] on multi-objective is promising, but limited. The objective functions are unable to express all desirable properties of a partition. For instance, models that are well suited to minimizing the number of messages or the maximum communication per processor instead of the total communication are still needed. Of further value would be hybrid models that encapsulate several metrics. New partitioning metrics may lead to new algorithmic challenges.

(2) *Partitioning for alternative architectures.* Most of the work in partitioning techniques has been motivated by distributed memory architectures and has tried to minimize interprocessor communication. Similar, but not identical, issues arise in shared memory machines (SMPs), where it is advantageous to partition the shared memory between the processors to minimize cache coherence overhead. However, the precise objectives in the shared memory setting may differ from those for distributed memory machines. This problem deserves more attention.

Other architectural trends pose different challenges for partitioners. One important development is the growing importance of heterogeneous computing platforms. Many current parallel machines consist of a collection of shared memory nodes networked together. These machines exhibit significant network heterogeneity. Accesses within an SMP are fast, but access between SMPs is slow. There is a need for work in this area.

Another important architectural development is the growing popularity of build-it-yourself parallel computers, exemplified by Beowulf-class machines. Machines built in this way can exhibit both network and processor heterogeneity. The partitioner needs to worry about differing processor speeds and memory sizes, as well as varying access times. Appropriate machine models and partitioning approaches for heterogeneous architectures are largely an untouched area, although one step in this direction has been taken by Teresco et al. [30].

(3) *Parallel partitioning.* Most of the work on parallel partitioning has been done in the context of dynamic load balancing. Algorithmically, dynamic load balancing is more challenging than the problems we have been discussing since there is a pre-existing partition. If the new partition deviates significantly from the current one,

then a large remapping cost is incurred. This consideration does not occur in static settings and complicates the evaluation of dynamic partitioning algorithms. These issues were discussed in Section 3.4.

Independent of dynamic problems, several trends are increasing the need for parallel partitioners. First is the interest in very large meshes, that will not 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 on which the problem will run. With heterogeneous parallel machines, knowing the number of processors is insufficient – the partitioner should also know their relative speeds and memory sizes. A user wants 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 [32] and ParMETIS [21]. This is an active area of research.

It is worth noting that some of the more effective dynamic load balancers for grid problems use the grid geometry instead of the graph of data dependencies. Among the widely used algorithms in this class are recursive coordinate bisection [17] and variants of space filling curves [33]. These algorithms tend to be faster than graph methods, but produce lower quality partitions. It is likely that graph and geometric partitioners will continue to coexist since the optimal trade-off between runtime and quality varies from application to application.

(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 used to minimize parallel communication. For instance, Farhat et al. [8] argue that the domains must have good aspect ratios (e.g., not be long and skinny). It can also be important that subdomains are connected, even though the best partitions for parallel communication need not be. For the most part, practitioners of domain decomposition have used partitioning algorithms developed for other purposes, with perhaps some minor perturbations at the end. But a concerted effort to devise schemes that meet the need of this community could lead to significant advances. Progress in this area will probably require a combination of ideas from numerical analysis and graph algorithms.

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. We expect to see a continuing stream of new insights and approaches that more closely fit the different classes of applications. As the hegemony of the standard approach crumbles, we foresee a fracturing of the partitioning field as different researchers choose to work on different models and applications. This is a positive development to the extent that this more focused work leads to better tools for specific applications.

Acknowledgements

The authors would like to thank Cevdet Aykanat, Barry Peyton, Chuck Romine, and the anonymous referees for their comments.

References

[1] C. Aykanat, Private communication, 1998.
[2] T. Bui, C. Jones, A heuristic for reducing fill in sparse matrix factorization, in: Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 1993, pp. 445–452.
[3] U.V. Çatalyürek, C. Aykanat, Decomposing irregularly sparse matrices for parallel matrix–vector multiplication, in: Parallel Algorithms for Irregularly Structured Problems, Irregular '96, number 1117 in Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 75–86.
[4] U.V. Çatalyürek, C. Aykanat, PaToH a multilevel hypergraph partitioning tool for decomposing sparse matrices and partitioning VLSI circuits, Technical Report BU-CEIS-9902, Department of Computer Engineering and Information Science, Bilkent University, Turkey, 1999.
[5] U.V. Çatalyürek, C. Aykanat, Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 10 (1999) 673–693.
[6] J. Cong, M.L. Smith, A parallel bottom–up clustering algorithm with applications to circuit partitioning in VLSI design, in: Proceedings of the 30th Annual ACM/IEEE International Design Automation Conference, DAC '93, ACM, 1993, pp. 755–760.
[7] A.E. Dunlop, B.W. Kernighan, A procedure for placement of standard-cell VLSI circuits, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 4 (1985) 92–98.
[8] C. Farhat, N. Maman, G. Brown, Mesh partitioning for implicit computation via domain decomposition: impact and optimization of the subdomain aspect ratio, Int. J. Numer. Meth. Eng. 38 (1995) 989–1000.
[9] M. Garey, D. Johnson, L. Stockmeyer, Some simplified NP-complete graph problems, Theoret. Comput. Sci. 1 (1976) 237–267.
[10] B. Hendrickson, Graph partitioning and parallel solvers: has the emperor no clothes? in: Solving Irregularly Structured Problems in Parallel, Irregular '98, number 1457 in Lecture Notes in Computer Science, Springer, Berlin, 1998, 218–225.
[11] B. Hendrickson, T.G. Kolda, Partitioning sparse rectangular matrices for parallel computations of Ax and A^Tv, in: Applied Parallel Computing in Large Scale Scientific and Industrial Problems, PARA'98, number 1541 in Lecture Notes in Computer Science, Springer, Berlin, 1998, pp. 239–247.
[12] B. Hendrickson, T.G. Kolda, Partitioning nonsquare and nonsymmetric matrices for parallel processing, SIAM J. Sci. Comput. 21 (2000) 2048–2072.
[13] B. Hendrickson, R. Leland, The Chaco user's guide, version 2.0, Technical Report SAND95-2344, Sandia National Labs, Albuquerque, NM, 1995.
[14] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: Proceedings of the 1995 ACM/IEEE Conference, High Performance Networking and Computing, Supercomputing '95, ACM, New York, 1995.
[15] B. Hendrickson, R. Leland, R.V. Driessche, Skewed graph partitioning, in: Proceedings of the Eighth SIAM Conference Parallel Processing for Scientific Computing, SIAM, 1997.
[16] B. Hendrickson, E. Rothberg, Improving the runtime and quality of nested dissection ordering, SIAM J. Sci. Comput. 20 (1998) 468–489.
[17] M.T. Jones, P.E. Plassmann, Computational results for parallel unstructured mesh computations, Comput. Syst. Eng. 5 (4–6) (1994) 297–309.
[18] G. Karypis, Private communication, 1998.
[19] G. Karypis, R. Aggarwal, V. Kumar, S. Shekhar, Hypergraph partitioning using multilevel approach: application in VLSI domain, in: Proceedings of the 34th Annual ACM/IEEE International Design Automation Conference, DAC '97, ACM, 1997.

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

[21] G. Karypis, V. Kumar, Parallel multilevel graph partitioning, Technical Report 95-036, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[22] G. Karypis, V. Kumar, Multilevel algorithms for multi-constraint graph partitioning, Technical Report 98-019, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1998.

[23] B.W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell Syst. Tech. J. (1970) 291–307.

[24] T.G. Kolda, Partitioning sparse rectangular matrices for parallel processing, in: Solving Irregularly Structured Problems in Parallel, Irregular '98, number 1457 in Lecture Notes in Computer Science, Springer, Berlin, 1998, pp. 68–79.

[25] F. Pellegrini, Static mapping by dual recursive bipartitioning of process and architecture graphs, in: Proceedings of the Scalable High Performance Comput. Conference, IEEE, 1994, pp. 486–493.

[26] A. Pınar, U.V. Çatalyürek, C. Aykanat, M. Pınar, Decomposing linear programs for parallel solution, in: Applied Parallel Computing in Computations in Physics, Chemistry and Engineering Science, PARA '95, number 1041 in Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 473–482.

[27] K. Schloegel, G. Karypis, V. Kumar, A new algorithm for multi-objective graph partitioning, in: Proc. EuroPar'99, Lecture Notes in Computer Science, Springer, Berlin, 1999.

[28] J. Stiller, K. Boryczko, W. Nagel, A new approach for parallel multigrid adaption, in: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, SIAM, 1999.

[29] S.-H. Teng, Points, spheres and separators: a unified geometric approach to graph partitioning, Ph.D. Thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.

[30] J.D. Teresco, M.W. Beall, J.E. Flaherty, M.S. Shephard, A hierarchical partition model for adaptive finite element computation, Comput. Methods Appl. Mech. Engrg. 184 (2000) 269–285.

[31] R. Van Driessche, D. Roose, Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem, in: High-Performance Computing and Networking, number 919 in Lecture Notes in Computer Science, Springer, Berlin, 1995, pp. 392–397.

[32] C. Walshaw, M. Cross, M. Everett, Mesh partitioning and load-balancing for distributed memory parallel systems, in: Proceedings of Parallel & Distributed Computing for Computational Mechanics: Systems and Tools, Saxe-Coburg Publications, 1998.

[33] M.S. Warren, J.K. Salmon, A parallel hashed oct-tree *n*-body algorithm, in: Proceedings of Supercomputing '93, Portland, OR, November 1993.