Load Balancing Fictions, Falsehoods and Fallacies

Bruce Hendrickson

Parallel Computing Sciences Dept., Sandia National Labs
Albuquerque, NM 87185–1110

Abstract

Effective use of a parallel computer requires that a calculation be carefully divided among the processors. This *load balancing* problem appears in many guises and has been a fervent area of research for the past decade or more. Although great progress has been made, and useful software tools developed, a number of challenges remain. It is the conviction of the author that these challenges will be easier to address if we first come to terms with some significant shortcomings in our current perspectives. This paper tries to identify several areas in which the prevailing point of view is either mistaken or insufficient. The goal is to motivate new ideas and directions for this important field.

Key words: load balancing, graph partitioning, parallel computer

1 Introduction

Among the factors contributing to the success of parallel computing in the nineties, algorithms and software for load balancing have played a prominent role. Efficient use of a parallel computer requires that the workload be evenly distributed among processors while the amount of inter-processor communication is kept small. This basic challenge presents itself differently in different applications, but the underlying issues are the same. It is quite common to recast this load balancing problem in terms of graphs. Vertices of the graph

[1] To appear in Appl. Math. Modelling. This work was supported by the Applied Mathematical Sciences program, U.S. DOE, Office of Energy Research, and was performed at Sandia National Labs, operated for the U.S. DOE under contract No. DE-AC04-94AL85000.

represent indivisible computations, while edges encode data dependencies between computations. Load balancing is then commonly phrased as a graph partitioning problem – divide the vertices into sets of equal work while cutting as few dependencies (edges) as possible.

Graph partitioning software targeted at parallel computing began to appear in the early 90's and has since become a cottage industry. Among the currently available alternatives are Chaco [1], METIS [2], Jostle [3], PARTY [4] and SCOTCH [5]). Around the same time, complex scientific computing applications began to exhibit excellent performance on parallel machines. A clear conclusion is that the partitioning tools enabled the successful parallelizations.

Although there is some truth in this assertion, there is also a significant degree of falsehood. As we will detail below in §2, the standard graph partitioning approach has serious shortcomings. In fact, it is only due to the relative simplicity of computational meshes that graph partitioners have been successful at all. More generally, it is our contention that the load balancing research community (including the author) has pursued a set of objectives which are incomplete and sometimes even ill advised. We present these concerns below as a set of load balancing fictions. In so doing, we hope to contribute to a broadening of perspective and to encourage and identify new and important research questions.

2 Fiction I: The Edge-Cut Deceit

As mentioned above, graph partitioning is an important conceptual and practical tool for load balancing. As a prototypical example, consider a calculation performed on a mesh, where at every step each mesh point updates its value by some function of its neighbors' values. Each mesh point can be considered as a vertex in a graph, and two vertices are connected by an edge if they need each other's values. Graph partitioning can be used to divide the vertices into sets with nearly identical numbers of vertices such that few edges cross between sets. If each set is assigned to a processor, then the workload will be balanced while the communication requirements are kept modest. A useful generalization of this model allows for weights on vertices and edges to encode heterogeneous computation and communication requirements respectively. This correspondence between load balancing and graph partitioning has motivated much algorithmic and software development.

Unfortunately, as discussed in detail in [6], the correspondence is less close than it appears for two reasons that we elaborate upon below. First, although it is widely believed to be the case, the volume of communication required by an application is not proportional to the number of graph edges cut by the partition. Second, the volume of communication is at best a weak predictor of communication cost. Consider the mesh example depicted in Fig. 1. In this example, the curve represents the partition between two sets. Each mesh point with neighbors in the other set needs to send its value to the other processor. Thus, the communication volume from the left set to the right set is 8, while that from the right to the left is 7. This contrasts with the value of 10 edges cut.

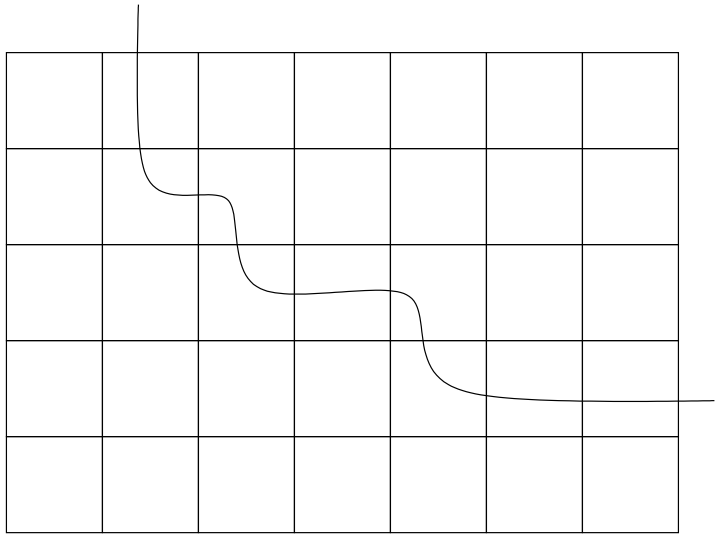

Fig. 1. Edge cuts versus communication volume.

The problem with the edge cut metric is that several edges can describe the same need for data transfer. Since a datum need only be communicated once, no matter how many vertices on the other side may need it, edge cuts tend to over-estimate the true volume of communication. A simple corollary is that reducing edge cuts might not reduce the true communication volume. This behavior was observed in [7]. But unfortunately, the graph partitioning software in wide use today fails to minimize a correct metric of communication volume.

Two alternative approaches are possible. One is to keep the same graph description, but to explicitly try to minimize the true communication volume. For an unweighted graph, this would merely be the number of vertices which have neighbors in another partition. This approach has been adopted in recent versions of METIS [8]. Unfortunately, this approach is not well suited to problems in which the amount of data associated with the different edges varies. In this case, a more accurate and elegant approach is the hypergraph model of Çatalyürek and Aykanat [9,10]. In this model, the set of vertices which generate and consume a particular value are all connected by a single *hyperedge*. If that hyperedge is split between two processors then a single communication cost is incurred. Each hyperedge can be assigned a different weight if the communication costs are non-uniform. In this way the total number (or weight) of cut hyperedges exactly equals the volume of communication.

Not only have the graph partitioning tools been minimizing an inaccurate measure of communication volume, the true cost of a communication operation is not well modeled by volume. For a distributed memory, message passing computer, the cost of a single message is usually well modeled by two terms: a startup or latency cost which is independent of message size and a bandwidth term which is proportional to the size of the message. Although graph partitioning models try to minimize the latter term, they do nothing for the former. If messages are of only modest size, the latency cost can dominate.

A second effect which is not well captured by standard graph partitioning approaches is the effect of communication congestion. Typically in scientific computations, many messages are simultaneously competing for network resources. Messages which use independent paths through the network can proceed at the same time, but those that compete for paths must take turns. Thus, the pattern of collective communication can have a significant impact on the communication cost. Current graph partitioners fail to accurately deal with this issue.

Even if latency and congestion considerations are included, graph partitioning models invariably try to minimize some version of the total communication cost. But the determining factor in parallel performance is the slowest processor. Graph partitioning generally produces sets with widely varying communication loads. A better objective would be to minimize the maximum communication load, or perhaps to reduce the communication on the most computationally burdened processors. Current partitioning formulations and implementations are not well suited for these objectives.

In summary, current partitioning tools try to optimize an objective that is far removed from the true cost incurred by a parallel computation. An obvious question is why they have proved so successful at enabling large calculations. The reason is that the vast majority of the applications using graph partitioners come from differential equations and so the underlying graph is a mesh. This has several implications. First, the geometric properties of meshes ensure that good partitions exist. If a mesh in d dimensions has n vertices it will have separators of size $n^{(d-1)/d}$ [11]. This ensures that for sufficiently large problems, the ratio of communication to computation will be small, so any reasonable partition should lead to good parallel performance. Second, the vertices associated with computational meshes typically have a bounded number of neighbors. This limits the harm caused by the edge cut approximation. Finally, meshes are generally fairly homogeneous, so the various sets are similar. This limits the communication inhomogeneity possible in these applications.

Unfortunately, other applications are less forgiving. Matrices arising from interior point methods, the singular value decomposition for latent semantic indexing, and other applications are generally much less structured than mesh matrices. In particular, the variation in the number of edges associated with a vertex can be quite large. Çatalyürek and Aykanat find that for such matrices the hypergraph model can reduce communication by 30–40% [10]. The corresponding percentage improvement for meshes is only in the single digits [12]. But without the guarantee of good partitions, these highly unstructured matrices are liable to require more communication than meshes, and so the partition quality will more directly translate into increased runtime. So although our flawed models have been sufficient for parallelizing the solution of differential equations, future applications will require much more careful models and approaches.

3 Fiction II: Simple Graphs are Sufficient

As described above, the standard manner in which graph partitioning is applied to load balancing involves an undirected graph with weighted vertices and edges. In addition to the problems discussed in §2 with accurately representing communication cost in this model, these simple graphs also suffer from a lack of expressibility. Many important classes of computations cannot be accurately described by this model. Examples include the following.

- **Multi-Stage Calculations.** Many applications consist of a sequence of different, time-consuming calculations. One important example is multiphysics simulations which consist of interleaved computations of, for example, fluids and structures. More mundane examples include the application of a matrix and a preconditioner during an iterative solver, or the different grid calculations in a multi-grid scheme. Although each individual stage of such calculations might be describable by a simple graph, the union of stages often cannot be. This is particularly true if there is a barrier between the different stages, so that overall load balance is only achieved if each individual stage is balanced. When a single stage dominates the runtime, a standard graph may be sufficient, but the need to balance several stages is an objective which cannot be described this way.
- **Complex Constraints.** When calculations consist of multiple stages, there are often complicated couplings between them. For instance, the different grids comprising a multigrid solver are related. In some (but not all) cases, the coarse-grid vertices are a subset of the fine-grid vertices. Mechanisms to express these relationships to the partitioner are needed. Another important example arises in finite element calculations where some computations occur on the nodes and others on the elements. A partition which balances each of these computations would be desirable, and the intimate ties between

nodes and elements needs to be encoded. For mesh-based applications, the DRAMA interface provides a uniquely rich model for this problem [13]. Unfortunately, current graph partitioning tools are not yet up to this complexity.

- **Unsymmetric Dependencies.** In the standard graph model, if a vertex i needs data from a vertex j, then j also needs data from i. Not all dependencies exhibit this symmetry. For instance, matrix-vector multiplication leads to symmetric dependencies only if the matrix is structurally symmetric. Although many methodologies for solving differential equations lead to structurally symmetric matrices, other applications do not. Important examples include latent semantic indexing, least squares problems for data analysis and interior point methods for optimization. The standard graph model does not describe these kinds of problems well.
- **Differing Inputs and Outputs.** When the set of dependencies is symmetric, then the set of inputs to a calculation is equivalent to the set of outputs. But when the dependencies are nonsymmetric the inputs and outputs can differ. For instance, consider the problem of matrix-vector multiplication for non-square matrices. In this case, the set of inputs is of different size than the set of outputs. For such examples, a model is needed which allows the inputs and outputs to be handled separately. The standard graph model is not able to do this.

To address these deficiencies, several alternative partitioning models have been proposed in recent years. One of these is the hypergraph model of Çatalyürek and Aykanat introduced in §2. In addition to correctly accounting for communication volume, this model can encode unsymmetric dependencies and, to a certain extent, differing inputs and outputs.

Closely related is the bipartite graph model of Hendrickson and Kolda [14,15]. In this model, inputs are designated as one set of vertices and outputs as a distinct set of vertices. An edge connects an input to an output that depends upon it. This model was devised specifically to handle the problems of unsymmetric dependencies and differing inputs and outputs. It can also encode certain limited classes of multi-stage calculations like two matrices acting on a vector. But it still minimizes the non-optimal metric of edge cuts.

A more general approach than the bipartite model is the multi-constraint, multi-objective partitioning approach of Karypis, Kumar and Schloegel [16,17]. This model was designed to partition multi-stage calculations, but its generality allows for other applications. It begins with a standard graph model, but augments it in two ways. First, instead of having a single weight for each vertex, vertices are given a vector of weights. A partition is only considered balanced if each of the components of the weight vector is balanced. For instance, value k in the weight vector could reflect the computational cost associated with stage k of the computation. The partitioner would then try to find a single decomposition that partitions each stage evenly. The second augmentation involves edge weights. As with vertices, each edge is allowed to have several weights, reflecting the communication cost associated with different stages.

The generality of this model is appealing, but it leads to very challenging partitioning problems. In many cases, a more focused model may be easier to work with. However, in many ways even the multi-objective, multi-constraint model is not general enough. It cannot, for instance, address the problem of minimizing both the volume of communication and the number of messages. And the model still suffers from all the shortcomings of the edge-cut metric.

Despite the recent activity in alternative partitioning models, much work remains to be done. New approaches which address some of the remaining deficiencies are urgently needed.

4 Fiction III: Partition Quality is Paramount

When advocating a new partitioning algorithm, researchers invariably argue their case by comparing their partition quality and (sometimes) runtime against existing tools. While these two criteria are important, they are far from comprehensive. As discussed in more detail in [18], desirable properties of a partitioner include the following.

- **Balance the Load.** This requirement may seem obvious, but it isn't always simple. For example, the challenge of balancing multi-stage calculations was discussed in §3. More generally, how should the work load be measured? How important is precise balance?
- **Minimize Communication Cost.** As argued in §2, communication cost is a somewhat elusive quantity. The most appropriate metric certainly depends upon the properties of the parallel computer, and probably on those of the application as well.
- **Run Fast in Parallel.** When static partitioning can be performed as a serial preprocessing step, efficiency isn't crucial. But when the partitioning is being performed on an expensive parallel machine instead of a cheap workstation, runtime can be critical. Time spent partitioning is time lost to the application, and can only be justified by a resulting improvement in application performance. This is particularly true for dynamic or adaptive calculations which need to be repartitioned repeatedly.
- **Be Incremental.** The issue of incrementality is critical for dynamic problems, but has no counterpart in static partitioning. When an adaptive or dynamic calculation becomes unbalanced and invokes a repartitioner, the data is already distributed across processors. If the partitioner dictates a new decomposition which is quite different from the current one, then sig-

nificant quantities of data will need to be transferred between processors. Thus, partitioners which adjust the decomposition in an incremental way are desirable. As an example of the importance of this issue, Touheed, et al. report results of several different partitioning algorithms for an adaptive flow calculation on 32 processors of an SGI Origin [19]. In their experiments, the time for data migration was significantly larger than the time spent determining the new partition. Obviously, the relative importance of partition quality, runtime and incrementality depends upon the application. But it may be worthwhile to sacrifice partition quality to improve incrementality.

- **Be Frugal With Memory.** As with runtime, the partitioner is competing with the application for finite space on the parallel machine. A partitioner which requires only modest memory will allow larger calculations to be performed. Again, the importance of this issue varies enormously between applications.
- **Support Determination of Communication Pattern.** Once a new decomposition has been determined and the data redistributed, the application still needs to figure out how to work with the new partition. Specifically, each processor needs to know what data to exchange with which other processors. In its most general form, this determination can be expensive. Some partitioners, generally those based upon geometric properties of the data, can greatly simplify this task.
- **Be Easy to Use.** Although difficult to quantify, ease of use is a critical aspect of a good tool. Many, if not most, application developers would gladly trade some performance for simplicity. For static partitioners with their file interfaces, this isn't a big issue. But for dynamic partitioners which are invoked via parallel subroutine calls, the design of the interface is very important.

The importance of these different issues varies between applications. But it is insufficient to focus our attention solely on only one or two criteria.

5 Fiction IV: Existing Tools Solve the Problem

A number of good partitioning and load balancing tools have been developed in recent years, and new packages continue to appear. Developers of such software (including the author) like to believe that their tools meet the needs of a wide range of parallel applications. Clearly, the usage of these tools supports this belief. But application developers have no choice but to use the available tools. In fact, the existing tools are far from ideal. Two shortcomings in particular are worth mentioning.

The first shortcoming is the inadequacy of the graph partitioning model which was discussed in §2. Although it has been a useful abstraction, load balancing is not identical to graph partitioning.

The second shortcoming is in the nature of the software engineering. Here, it is worth differentiating between static and dynamic tools. Most load balancing packages address the static partitioning problem, and run on a serial computer. They are designed to work as part of a preprocessing step in which a large computation is prepared for a parallel machine. The tools are generally invoked as stand-alone codes with file interfaces. This approach is fairly mature, and sufficient for many applications.

However, a growing number of applications are ill-suited to this approach and would benefit from more sophisticated software engineering. Parallel load balancing tools are clearly needed for adaptive calculations, but they are necessary in several other settings as well. For instance, if the computational problem is generated on a parallel machine (e.g. by a parallel mesh generator) then the load balancing needs to be done in parallel too. Also, if the target parallel computer is heterogeneous, then the partitioning can't easily be done in advance. Only at runtime will an application know what resources are available to it, and without this information it cannot properly balance the load.

Unfortunately, parallel or dynamic partitioning software is much less mature than its static counterpart. To a large degree this is merely a consequence of the added complexity associated with developing parallel algorithms and software. But several algorithmic and software issues make dynamic load balancing libraries inherently more difficult. First, as discussed in §4, the dynamic partitioning problem has to contend with the issue of incrementality. For the static problem, there are only two principle metrics: quality and runtime. And multilevel methods seem to cover this two-dimensional space quite well, providing high quality solutions in modest time. But the need for incrementality adds a third key metric to the dynamic problem. It seems unlikely that a single algorithm will be able to cover this full three-dimensional space adequately. So a good tool will need to provide a suite of partitioners, some with greater incrementality, some with higher quality, etc. It requires more work to build such a toolkit than to implement a single algorithm.

Another important difference between the static and the dynamic cases is that dynamic partitioners cannot be stand-alone codes with file interfaces. Instead, they will be invoked as subroutine libraries, which raises new challenges in software design. The input arguments to most existing parallel partitioners (e.g. ParMETIS [20], PJostle [3], DRAMA [13]) include a graph in some format. The burden of constructing the graph in the specified format is placed upon the application developer. The exception to this rule is Zoltan [21], in which functions are passed across the interface instead of data. In our view, this object-oriented design has several advantages. It reduces memory requirements, allows for the partitioner to only extract the information it needs, and the tool can be modified without changing the interface. But must importantly, we believe it is easier to use.

In summary, the load balancing community has yet to embrace the software engineering techniques which have simplified the inter-operability of commercial software. We believe that new tools developed with this mindset will significantly improve the utility of load balancing software. This is particularly true for dynamic load balancing tools.

6 Fiction V: Load Balancing Means Finding the Right Partition

A good distribution of work among processors is the key to obtaining high performance on a parallel computer. Each processor should have useful work to do for the duration of the computation, and the overhead of interprocessor communication must be minimized. A natural, but flawed, conclusion is that the best way to parallelize an application is to find a good partitioning.

This conclusion is flawed for two reasons. First, as discussed in §4, dynamic and adaptive calculations require the partition to be adjusted on the fly. No single partition is adequate for the duration of the computation. The second, and more interesting, reason arises in the context of multi-stage calculations which were introduced in §3. The different stages in these calculations can have conflicting partitioning objectives. An optimal partition for one stage may be far from optimal for another.

As discussed in §3, several alternative partitioning models are able to combine two or more stages into a single partitioning problem. However, this may not be the best answer. Perhaps no single decomposition exists which enables good performance from all the stages. An alternative is to support multiple decompositions so that each stage can be performed efficiently, independent of the partitioning demands of the other stages. The disadvantage of this approach is that data must be moved between the decompositions, incurring a cost in both time and memory. But in some applications, this approach enables complex calculations to run efficiently on many more processors than a single decomposition would allow. One such application is the crash simulation work of Plimpton, et al. [22].

Low speed impacts (e.g. car crashes) are usually simulated with Lagrangian techniques. A mesh is constructed of the car in its native geometry. As the simulation proceeds and the car hits, for example, a telephone pole, the mesh distorts to follow the deformation of the bumper. As the bumper deforms, it eventually strikes the radiator. In the simulation, this is revealed when the mesh of the bumper contacts the mesh of the radiator. At this point, new forces need to be included in the simulation.

There are two dominant stages in these calculations. The first is a finite element analysis of the impacting bodies to reveal how they deform under stress. The second is the geometric search for contacts in the mesh. Unfortunately, these two stages have quite different decomposition needs. The finite element analysis is a prototypical example for which graph partitioning works well. The pattern of data dependencies is static and determined by the topological structure of the mesh. The contact detection stage changes every timestep, and data dependencies reflect geometric proximity. In the work by Plimpton, et al., graph partitioning is used for the finite element stage and a geometric partitioner is used for contact detection. The resulting crash code was the first to scale well to hundreds and even thousands of processors and has enabled simulations of unprecedented scale and fidelity [23].

Several aspects of crash simulations make them good candidates for the multiple-decomposition approach. These applications are generally not memory intensive, so the duplication of data associated with the multiple decompositions is not a problem. Also, each stage of the computation is expensive, so the cost of the data transfer can be tolerated. And finally, the partitioning needs of the two stages are quite different, so single-decomposition approaches have exhibited only limited scalability. For multi-stage applications with these features, multiple decompositions should be considered.

7 Fiction VI: All the Problems are Solved

Within the load balancing community, there is a sense of satisfaction with the status quo. The complacency that this engenders makes this the biggest and most damaging fiction of all. As discussed above, there is a compelling need for more accurate and expressive models, new partitioning algorithms to address these models and better designed software tools and interfaces. In addition to these general needs, a number of interesting and important problems have not yet been adequately addressed. Among them are the following.

- **Partitioning for sparse solve on each subdomain** Some preconditioners based on domain decomposition involve a sparse, direct solver on each subdomain. An important example is the FETI class of preconditioners [24]. The work and memory required by each subdomain for such a calculation is quite difficult to predict in advance. It requires a much more detailed analysis than can be provided by merely summing vertex weights. Current graph partitioning models are unable to balance the work or memory required on each subdomain. New models and partitioning ideas would be very helpful here, and would also be applicable to the parallelization of sparse direct methods.

- **Partitioning for good aspect ratios** Another property of FETI preconditioners is that they work best when the individual subdomains have small aspect ratios (i.e. not be long and skinny) [25]. Recent work by Diekmann, et al. [26,27] has shown an elegant method for encoding this objective into the standard graph model. Further work in this area would be welcome.
- **Partitioning for heterogeneous parallel architectures** Most new and emerging parallel architectures have heterogeneous networks. For instance, many machines are clusters of symmetric multi-processors. The communication properties within an SMP are different than those for communication between SMPs. The growth of commodity cluster computing is also leading to parallel machines with heterogeneity among the processors. Existing partitioning and load balancing tools fail to incorporate this information. Several important questions arise including how to model the architecture and how to adapt partitioning algorithms for them. Some interesting early work on these questions has been pursued by Teresco, et al. [28], but much remains to be done.
- **Partitioning to minimize congestion** As discussed in §2, the true cost of a collective communication operation can be difficult to predict. In the not–too–distant past, vendors provided specialized high performance networks that efficiently routed complex sets of messages. But for the large number of current machines with commodity networking, network performance is less robust. In particular, if many messages are all competing to use a single wire, then performance will suffer. Methods for generating partitions in which this contention is avoided are needed. Some ideas along this line can be found in the work of Pellegrini [29] or Hendrickson, Leland and Van Driessche [30], but new insights are needed.

Despite the general feeling that load balancing is a mature area, there is a great need for new ideas and insights. The wide range of open or only imperfectly solved problems provides many opportunities for new research.

Acknowledgements

The ideas in this paper have been influenced by my collaborations with Rob Leland, Tammy Kolda and Karen Devine. I have also benefited from conversations with Alex Pothen, Steve Plimpton and George Karypis. In addition, I am indebted to Guy Lonsdale and the organizers of the Third DRAMA Workshop for an invitation to speak, which motivated the organization of this paper.

References

[1] Bruce Hendrickson and Robert Leland. The Chaco user's guide, version 2.0. Technical Report SAND95-2344, Sandia National Labs, Albuquerque, NM, 1995.

[2] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, Dept. Computer Science, Univ. Minnesota, Minneapolis, MN, 1995.

[3] 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: Systems and Tools*. Saxe-Coburg Publications, 1998.

[4] R. Preis and R. Diekmann. The PARTY partitioning-library, user guide – version 1.1. Technical Report tr-rsfb-96-024, Dept. of Computer Science, University of Paderborn, Paderborn, Germany, 1996.

[5] F. Pellegrini. SCOTCH 3.1 user's guide. Technical Report 1137-96, LaBRI, Université Bordeaux I, France, August 1996.

[6] Bruce 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, pages 218–225. Springer-Verlag, 1998.

[7] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 2000. To appear.

[8] George Karypis. Personal communication, 1998.

[9] Ümit V. Çatalyürek and Cevdet 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, pages 75–86. Springer-Verlag, 1996.

[10] Ü. Çatalyürek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, 1999.

[11] S.-H. Teng. *Points, Spheres and Separators: A Unified Geometric Approach to Graph Partitioning*. PhD thesis, Dept. Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1991.

[12] Cevdet Aykanat. Personal communication, 1998.

[13] http://www.ccrl-nece.de/DRAMA.

[14] Tamara 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, pages 68–79. Springer-Verlag, 1998.

[15] B. Hendrickson and T. Kolda. Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel processing. *SIAM J. Sci. Comput.*, 2000. To appear.

[16] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, Dept. Computer Science, Univ. Minnesota, Minneapolis, MN, 1998.

[17] K. Schloegel, G. Karypis, and V. Kumar. A new algorithm for multi-objective graph partitioning. In *Proc. EuroPar'99,*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[18] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Comp. Meth. Appl. Mech. Eng.*, 184(2–4):485–500, 2000. Invited paper.

[19] N. Touheed, P. Selwood, P. K. Jimack, and M. Berzins. A comparison of some dynamic load-balancing algorithms for a parallel adaptive flow solver application. *Parallel Comput.*, 2000. To appear.

[20] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. Technical Report 95-036, Dept. Computer Science, Univ. Minnesota, Minneapolis, MN, 1995.

[21] E. Boman, K. Devine, B. Hendrickson, W. Mitchell, M. St. John, and C. Vaughan. http://www.cs.sandia.gov/Zoltan.

[22] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner. Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics. *J. Parallel Distrib. Comput.*, 50:104–122, 1998. Previous version appeared in Proc. Supercomputing'96.

[23] K. Brown, S. Attaway, S. Plimpton, and B. Hendrickson. Parallel strategies for crash and impact simulations. *Comp. Meth. Appl. Mech. Eng.*, 184(2–4):375–390, 2000. Invited paper.

[24] C. Farhat and F. X. Roux. An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems. *SIAM J. Sci. Stat. Comp.*, 13:379–396, 1992.

[25] C. Farhat, N. Maman, and G. Brown. Mesh partitioning for implicit computation via domain decomposition: Impact and optimization of the subdomain aspect ratio. *Internat. J. Numer. Methods Engrg.*, 38:989–1000, 1995.

[26] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Aspect-ratio for mesh partitioning. In D. Pritchard and J. Reeve, editors, *Proc. Euro-Par'98*, pages 347–351. LNCS 1470, Springer-Verlag, 1998.

[27] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for adaptive fem. *Parallel Comput.*, 2000. To appear.

[28] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard. A hierarchical partition model for adaptive finite element computation. Technical report, Rensselaer Polytechnic Institute, Troy, NY, 1998. Submitted to Comp. Meths. Appl. Mech. Engrg.

[29] F. Pellegrini. Static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. Scalable High Performance Comput. Conf.*, pages 486–493. IEEE, 1994.

[30] Bruce Hendrickson, Robert Leland, and Rafael Van Driessche. Skewed graph partitioning. In *Proc. 8th SIAM Conf. Parallel Processing for Scientific Computing*. SIAM, 1997.