DYNAMIC LOAD BALANCING IN COMPUTATIONAL MECHANICS*

BRUCE HENDRICKSON AND KAREN DEVINE†

Abstract.
In many important computational mechanics applications, the computation adapts dynamically during the simulation. Examples include adaptive mesh refinement, particle simulations and transient dynamics calculations. When running these kinds of simulations on a parallel computer, the work must be assigned to processors in a dynamic fashion to keep the computational load balanced. A number of approaches have been proposed for this dynamic load balancing problem. This paper reviews the major classes of algorithms, and discusses their relative merits on problems from computational mechanics. Shortcomings in the state-of-the-art are identified and suggestions are made for future research directions.

Key words. dynamic load balancing, parallel computer, adaptive mesh refinement

1. Introduction. The efficient use of a parallel computer requires two, often competing, objectives to be achieved. First, the processors must be kept busy doing useful work. And second, the amount of interprocessor communication must be kept small. For many problems in scientific computing, these objectives can be obtained by a single assignment of tasks to processors that doesn't change over the course of a simulation. Calculations amenable to such a *static* distribution include traditional finite element and finite difference methods, dense linear solvers and most iterative solvers.

However, a number of important applications have computational requirements that vary over time in an unpredictable way. For such applications, high performance can only be obtained if the work load is distributed among processors in a time-varying, *dynamic* fashion. These kinds of computations seem to be particularly prevalent in computational mechanics and include the following applications.

- **Adaptive mesh refinement (AMR)**. AMR is a rapidly maturing technology in which the computational mesh is locally refined to minimize the error in a calculation. Elements are added or removed (h-refinement) and/or the degree of the approximation on individual elements is varied (p-refinement) to obtain a desired accuracy. During the course of a simulation, the amount and location of refinement can vary, so work must be migrated between processors to maintain load balance.
- **Contact detection in transient dynamics**. A simulation of a car crash is a prototypical transient dynamics calculation. A critical step in such a calculation is detecting when the deforming mesh intersects itself, e.g., when the bumper has crumpled into the radiator. This step requires a geometric search over a steadily evolving geometry. As the mesh moves, the work needs to be divided among processors in a different way.
- **Adaptive physics models**. In many computations, the computational effort associated with a data point varies over time. For instance, a constitutive model for a high strain region may be more expensive than one for low strain.

*To appear in Comp. Meth. Applied Mechanics & Engineering.
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.

†Sandia National Labs; Albuquerque, NM 87185-1111, {bah,kddevin}@cs.sandia.gov.

As the simulation evolves, the computational work associated with a point can vary, so the points may need to be redistributed among processors to balance the work.

- **Particle simulations**. When simulating particles under the influence of short-range forces (e.g., smoothed particle hydrodynamics or molecular dynamics with van der Waals forces), the particles interact only with geometrically near neighbors. As these neighbors vary over time, redistribution of the particles among processors can keep the communication cost low.
- **Multiphysics simulations**. It is increasingly common to couple multiple physical phenomena (e.g., crash and burn) into a single simulation instead of isolating them. While the simulation of isolated phenomena may be possible with a single distribution of data, coupled physics can require multiple distributions.

Different applications impose different requirements upon dynamic load balancing algorithms and software. Some applications are fundamentally geometric in nature (e.g., contact detection and particle simulations), while others are better described in terms of mesh connectivity (e.g., AMR). There are also complicated tradeoffs between the cost of the load balancer, the quality of the partition it produces and the amount of data that needs to be redistributed. The optimal tradeoff between these metrics depends on the properties of the application as well as those of the parallel machine, as we discuss in detail in §2. For all these reasons, there is no single dynamic load balancing algorithm that is suitable in all settings. A number of different methods have been been devised for specific problems or problem classes. In §3 we review the major categories of approaches and critically evaluate their strengths and weaknesses for the various problems in computational mechanics, using the metrics discussed in §2.

In general, dynamic partitioners are significantly more complex to parallelize than static ones. The reasons for this are not fundamentally algorithmic, but instead have to do with software complexity. While static partitioning tools can be run as a sequential preprocessing step, dynamic partitioning must be performed on the parallel machine. The load balancer must be called as a subroutine from the simulation code, so the two must agree on some data structures. The time and memory consumed by the partitioning algorithm take resources away from the simulation. Determining when it is advantageous to redistribute the data and tasks requires a code to monitor and predict its performance. If data are moved between processors, the data structures must be rebuilt and the interprocessor communication patterns need to be updated. All of these issues add considerable complexity to the application code. We will discuss this software problem in more detail in §4. We then draw some general conclusions and suggest some future research directions in §5.

2. Load Balancing Issues and Objectives. The ultimate goal of any load balancing scheme, static or dynamic, is to improve the performance of a parallel application code. To achieve this objective, partitioners specify how the *objects* comprising a computation are divided among processors. Objects can be mesh points or elements, atoms, smoothed particles, matrix elements or any other entity that the application treats as indivisible. For most applications in mechanics, the natural objects are components of a computational mesh or some sort of particle. A load balancer will determine how to assign these objects to processors to enable efficient parallel computation. In general, a parallel code will perform well if this assignment of objects has the following two properties.

A. The computational work is well balanced across processors.
B. The time spent performing interprocessor communication is small.

To satisfy these properties, several questions must first be addressed.

- What work needs to be balanced? Most real applications have several time consuming sections of code (or phases) such as constructing a matrix and solving the resulting linear system, or computing forces and integrating forward in time, or for transient dynamics the force calculation, time integration and contact detection. Multiphysics simulations can have an even larger number of computational phases. Unfortunately, a good division of objects for one phase may be poor for another. This problem can be handled in one of three ways. First, the load balancer can focus on the most time consuming phase. Second, the load balancer can try to balance an aggregate model of work, allowing non-optimal performance in each phase. Or third, each phase can be independently load balanced. The right approach will depend upon the application.
- How is workload measured? A simple approach is to count the number of objects assigned to each processor. But if objects have varying computational costs, then more refined estimates are needed. Examples include the number of matrix elements associated with a mesh object, the number of interactions a particle participates in, or the number of degrees of freedom for a p-refined finite element. A static load balancer is forced to use some such model of workload, but a dynamic load balancer can use actual measurements of runtime.
- What determines the cost of interprocessor communication? On most parallel machines, the communication cost grows with message volume and with the number of messages. But is it more important to minimize the sum of these, or the maximum among all processors? How important is message congestion or competition for wires? Is it worthwhile to allow some load imbalance if it reduces the communication cost? The answers to these kinds of questions are highly specific to the parallel machine and the application, but load balancers must have some model of communication cost.

These issues are relevant to both static and dynamic load balancing. But the dynamic load balancing problem has some additional requirements. In the static case, the decomposition of the problem can be performed by a stand-alone code on a sequential machine. But a dynamic load balancer has to run on a parallel machine, and must be invoked by the application code. Any memory or time consumed by the load balancer is lost to the application. Thus, a dynamic load balancer must satisfy a larger list of requirements than a static one. In addition to properties (A) and (B) above, a dynamic load balancer should strive to achieve the following.

C. It should run fast in parallel.
D. Memory usage should be modest.

Another key difference is that in the dynamic situation the objects are already assigned to processors. When the load balancer suggests a new assignment, some objects must be moved to new processors. The time spent moving objects can be very significant, so it is important to have the new partition be an *incremental* modification to the existing one. Once the objects have been moved, data structures must be updated. The modified data structures include those describing the objects owned by a processor and also those depicting the communication patterns for the application. These considerations suggest two more desirable properties for a dynamic load balancer.

E. A small change in the problem should induce only a small change in the decomposition. Algorithms that have this property will be called *incremental*.

F. It should be easy to determine the new communication pattern. For example, simple geometric regions may be better than complex ones.

The need for incrementality makes most static load balancing algorithms inappropriate for the dynamic problem. Algorithms can be *explicitly* incremental by specifically striving to limit data movement. Or, they can be *implicitly* incremental by achieving this property automatically. Several of the geometrical algorithms discussed in §3.2 and §3.3 are implicitly incremental.

During a simulation, the set of objects can change in several ways. The first possibility is for new objects to be created and existing ones to disappear. If the objects are grid points in adaptive mesh refinement, this type of transition is observed. A second possibility is that the set of objects remains unchanged, but the work associated with each object varies over time. This kind of behavior occurs in AMR if the objects are elements weighted by their polynomial degree or number of child elements. A third possibility is that interactions between objects appear and disappear. Objects representing moving particles typically have this property. Different load balancing algorithms vary in their capacity for handling these different kinds of changes.

Since the dynamic load balancer is called as a subroutine, its interface should be carefully designed. This design includes the mundane question of the argument lists, but a deeper issue is how well the abstraction in the tool reflects the needs of the application. One aspect of this problem relates to data structures. If the tool requires a data structure that isn't native to the application, the structure must be created. This construction takes time, memory and additional software development. Although generally not considered as part of the load balancing problem, this overhead is a significant deterrent to using new tools. We feel that object-oriented software techniques can play a significant role in addressing some of these problems. We will discuss these issues further in §4, but for now we simply add one more property to our list.

G. The tool should be easy to use.

3. Algorithms for Dynamic Load Balancing. In this section, we describe the major classes of dynamic load balancing algorithms and critically evaluate them using the criteria discussed in §2. In §3.1 we introduce master/slave techniques, which are powerful when appropriate, but are only suitable to a small minority of scientific computing applications.

Most dynamic load balancing algorithms can placed in one of two general classes: *geometric* and *topological*. Geometric methods divide the computational domain by exploiting the location of the objects in the simulation. Not surprisingly, geometric methods are well suited to problems in which interactions are inherently geometric like particle simulations or contact detection. But geometric methods can also be applied to divide meshes, where nearness is an approximation to connectivity. We discuss the two major kinds of geometric partitioners in §3.2 and §3.3.

Topological methods work with the connectivity of interactions instead of with geometric coordinates. The connectivity is generally described as a graph. These methods are best suited to partitioning computational meshes, where the connectivity is implicit in the mesh, but they can also be applied to particle systems. Topological methods can lead to better partitions than geometric algorithms, but are usually more expensive. We describe two classes of topological approaches in §3.4 and §3.5. In §3.6

we present some hybrid algorithms that can combine features of both geometric and topological methods.

3.1. Master/Slave. The master/slave approach is a very simple parallel computing paradigm. At its most basic, one processor, the *master*, maintains a pool of tasks which it doles out to the remaining *slave* processors. Slaves perform a computation and then ask the master for more work. This is a very flexible model, and many variations are possible including multiple masters, a hierarchy of slaves and variation in the units of work [16, 44].

The master/slave approach has some attractive features that make it well suited for many computational problems. It is quite simple to implement. It transparently handles wide variation in the cost of individual tasks, and even works on heterogeneous networks of workstations. Unfortunately, it is not a good match for most problems in scientific computing. Master/slave approaches are only appropriate if the tasks can be performed independently and asynchronously by a single processor. Furthermore, it must be possible to describe a task with a small amount of information so that the messages between master and slave are small. These assumptions are not satisfied by most scientific computations. But when they are, master/slave approaches are highly effective. Examples include Monte Carlo calculations [1], ray tracing [21], visualization [51], and parameter studies in which a sequential code needs to be run repeatedly with different inputs [15].

3.2. Simple Geometric. Most mechanics calculations have an underlying geometry. And for many physical simulations, objects (e.g., mesh points, particles, etc.) interact only if they are geometrically near each other. These properties enable the use of geometric methods to divide the work among processors. Here we review some simple but effective geometric partitioners. A different class of methods that also relies on geometry is described in §3.3.

Consider the problem of dividing a mesh into two pieces. A simple way to do this is to slice the mesh with a line (in 2D) or a plane (in 3D). Let all the mesh points or elements on one side of the plane comprise the first partition and those on the other side comprise the second. Depending on where the cut is made, partitions of varying sizes can be created. When performing a computation, communication will be required for each mesh element which has a neighbor on the other side of the plane, so ideally, the number of such elements will be small. Intuitively, the use of a line or plane to cut the mesh should help keep the amount of communication small, at least for well shaped meshes. This intuition has been proved correct by Cao, Gilbert and Teng [7], as long as the ratio between the sizes of the largest and smallest mesh elements is bounded.

Several partitioning algorithms have been proposed that exploit this idea of recursively dividing the domain using lines or planes. The simplest such method is known as Recursive Coordinate Bisection (RCB), and was first proposed as a static load balancing algorithm by Berger and Bokhari [5]. The name of the algorithm is due to the use of cutting planes that are orthogonal to one of the coordinate axes. The algorithm takes the geometric locations of all the objects, determines in which coordinate direction the set of objects is most elongated, and then divides the objects by splitting based upon that long direction. The two halves are then further divided by applying the same algorithm recursively.

Although this algorithm was first devised to cut into a number of sets which is a power of two, the set sizes in a particular cut needn't be equal. By adjusting the partition sizes appropriately, any number of equally-sized sets can be created.

This freedom to adjust set sizes was exploited by Jones and Plassmann [29] to improve the basic algorithm in their work on adaptive mesh refinement. They call their approach Unbalanced Recursive Bisection, or URB. The basic RCB algorithm divides the set of objects into halves, without paying attention to the geometric properties of the resulting sub-regions. If there is a large variation in object density, one of the two sub-regions could have a large aspect ratio. Since the communication volume is related to the size of the region boundary, large aspect ratios are undesirable. URB avoids large aspect ratios by considering geometric shape as it chooses a cutting plane. Instead of dividing the objects in half, it tries to divide the geometry in half using the observation that, if the set of objects is to be divided among q processors, it is OK for this cut to produce subsets with relative sizes $1 : (q-1)$, or $2 : (q-2)$, or .... URB selects whichever ratio leads to the most nearly equal division of the geometry. Jones and Plassmann report that this leads to a modest reduction in communication cost.

The key computational step in these algorithms is the determination of the placement of the cutting plane. Say, for example, we have chosen to divide in the direction orthogonal to the x-axis. We now need to find a value x_0 so that a specified fraction of the objects' x-coordinates are larger than x_0. We also need to solve this *selection* problem in parallel. One obvious approach would be to sort all the x-coordinate values, but this is unnecessarily expensive. A better method is to guess a value for x_0, count the number of objects to the left and right, and use the result to make a better guess. The counting process is performed in parallel, and the results are summed via a communication operation. The approach runs quite quickly, and is fairly simple to implement.

An alternative approach which also uses cutting planes is the Recursive Inertial Bisection (RIB) algorithm described by Simon [48]. This algorithm doesn't confine its cutting planes to be orthogonal to an axis. Instead, it tries to find a naturally long direction in the object distribution automatically, using a mechanical principle. The objects are treated as point masses, and the direction of principle inertial is identified. The cutting plane is then chosen to be orthogonal to this direction. The inertial direction can be found by computing eigenvectors of a 3×3 matrix, but the matrix construction requires summation of contributions from all the objects. Thus, this algorithm is somewhat more expensive than those that use coordinate directions, but it generally produces slightly better partitions.

The quality of partitions generated by these cutting plane algorithms are generally much poorer than those produced by some of the graph partitioning algorithms discussed in §3.5. However, their simplicity and speed are appealing. Also, for meshless problems which lack a static connectivity, the graph to partition can be difficult or expensive to construct, so geometric methods are preferable. Although RCB and URB generally produce slightly worse partitions, they have two compelling advantages relative to RIB. First, with RCB and URB the geometric regions owned by a processor are simple rectangular parallelepipeds. This geometric simplicity can be very useful. For instance, in [42] it is used to speed up the determination of which processors' regions intersect an extended object.

Second, and more universally, RCB and URB partitions are incremental. If the objects move a small amount, then the cutting planes will only change slightly and the new partition will be very similar to the previous one. This is not true of RIB partitions since the direction of the inertial axes is more sensitive to perturbations.

This mixture of simplicity, speed and incrementality make RCB and URB very attractive algorithms for dynamic load balancing. In addition to the adaptive mesh refinement applications mentioned above, they have been applied with success to the contact detection problem [42] and particle simulations [50, 42]. The shortcomings of these algorithms are the mediocre partition quality and the limitation to applications which possess geometric locality.

A more sophisticated partitioning approach developed by Miller, Teng and Vavasis uses circles or spheres instead of planes to divide the geometry [35]. The interior of the sphere is one partition while the exterior is the other. This algorithm has very attractive theoretical properties – it comes within a constant factor of the best possible bound for all well-shaped meshes. Unlike the result for cutting with planes, this proof does not require any additional constraints on the sizes of mesh elements. Experiments by Gilbert, Miller and Teng [28] show that this approach can generate partitions of quality comparable to those produced by graph partitioning algorithms. However, this algorithm is considerably more complex and expensive than the simple approaches that use cutting planes. Although this algorithm has not yet been applied to the dynamic partitioning problem, we feel it deserves consideration. It would be easier to parallelize than the more sophisticated graph partitioning algorithms. And incrementality can be enforced by not allowing the centers of the cutting spheres to move (although this restriction might invalidate the theoretical analysis).

3.3. Octrees and Space Filling Curves. A very different type of geometric partitioning approach is based upon a fine-grained division of the geometry. The small pieces are then combined to form the region owned by a processor. The fine-grained spatial division is performed by simultaneously dividing each of the coordinate axes in half. This produces four subregions in 2D and eight subregions in 3D. Note that unlike the cutting plane algorithms discussed in §3.2, this division is entirely geometric and takes no account of the locations of objects. Each of these subregions is divided further if it contains multiple objects. A simple data structure known as an *octree* (or *quadtree* in two dimensions) keeps track of the relationships between these geometric regions. The root of the octree represents the entire geometry. When a geometric region is divided, each of the eight octants becomes a child of the vertex representing the region. This data structure is widely used for mesh generation and adaptive mesh refinement [2, 11, 36, 47].

Octrees can also be used for partitioning. A traversal of the tree defines a global ordering on the tree leaves, which correspond to individual objects. This ordered list can then be sliced to generate any number of partitions. This basic algorithm is called Octree Partitioning [10, 19], or Space-Filling Curve (SFC) Partitioning. This approach was first used by Warren and Salmon for gravitational simulations [57]. Patra and Oden were the first to apply it to adaptive mesh refinement [38, 40]. Pilkington and Baden used this approach for smoothed particle hydrodynamics and reported results similar to using Recursive Coordinate Bisection [41].

For applications that don't already have an octree, a binning algorithm based on coordinate information can be used to build an octree for the load balancer. Each processor stores a part of the global octree. Two passes through the local octrees are needed. The first pass sums into each node of the octrees the amount of work contained in the node and its subtrees. A parallel prefix operation is performed on the processor work loads to obtain a cost structure for the set of all processors. Thus, the total cost and optimal cost per partition are known to all processors. A partial depth-first search of the octrees then uses the cost information to determine which processor should own the subtrees in the new decomposition. An entire subtree is added to a partition if its costs do not exceed the optimal partition size; otherwise, the traversal recursively descends the tree and evaluates each child node.

The partitions produced by SFC are slightly worse than those produced by the simple geometric algorithms discussed in §3.2 [19]. This is because the geometric region assigned to a processor is not a simple box, but rather a union of boxes which increases the boundary size. But SFC has several nice properties. It is fast; Flaherty and Loy report it to be faster than Recursive Inertial Bisection [19], and Pilkington and Baden observe it to be comparable in runtime to Recursive Coordinate Bisection [41]. It is also incremental. In the comparisons of Flaherty and Loy, it requires less data migration than RIB (but recall that RIB is not incremental).

Although the partitions can be generated without explicitly sorting the vertices, the global ordering induced by a sort can be useful. The ordering exhibits geometric locality which can improve cache performance in the computation. A global numbering also simplifies tools which automate a translation from a global numbering scheme to a per-processor scheme. This can simplify application development [17, 39].

To summarize, the runtime and quality of Space Filling Curve Partitioning are roughly comparable to the simple geometric approaches described earlier. SFC is perhaps a bit faster, but a bit lower quality. It is more complex to understand and to implement than simple geometric algorithms, but simpler than the graph partitioning approaches discussed in §3.5. In our opinion, as a dynamic load balancing tool, SFC has little advantage over the simpler Recursive Coordinate Bisection. However, the global numbering it enables can simplify some other aspects of code development and performance tuning.

3.4. Local Improvement. Local load-balancing methods use measures of work within small sets, or neighborhoods, of closely-connected processors to improve load balance within each neighborhood. The neighborhoods are chosen to overlap, so that, over several iterations of the local load-balancing, work can be spread from neighborhood to neighborhood, and eventually across the global processor array. Topological connections of the data are used to select objects for migration in a way that attempts to minimize communication costs for the application.

Unlike global methods, each iteration of a local method is usually quite fast and inexpensive. Because all of the information and communication is performed within small sets of processors, the methods scale well with the number of processors. By design, local methods are incremental, as they move objects only within a small group of processors. Moreover, they can be invoked to only improve load balance, rather than requiring that a global balance be reached before termination. One iteration of a local method can reduce a single heavily loaded processor's work load significantly. Since the total computation time is determined by the time required by the most heavily loaded processor, a small number of iterations may be all that is needed to reduce imbalance to an acceptable level.

When global balance is desired, however, many iterations of a local method may be required to spread work from a few heavily loaded processors to the other less-loaded processors. The convergence rate to global balance is determined by the particular local algorithm used to compute the amount of work to migrate. A number of methods will be discussed below. In addition, since global information is not used by local methods, maintaining high partition quality is more difficult with local methods. Local methods typically use simple heuristics to determine which objects should be migrated. These heuristics attempt to minimize parameters such as cut edges or domain interface size. A number of selection strategies will be discussed below.

In general, local methods are extremely effective at dealing with small changes in load balance, such as those caused by local refinement in adaptive finite element methods. However, large changes in the processor work loads may more appropriately be handled by a global method that would produce higher quality partitions.

Global load-balancing strategies are necessarily synchronous. All processors perform some computation, synchronize, and then enter the load-balancing phase. Thus, lightly loaded processors must wait idly for more heavily loaded processors to complete their computation before performing the load balancing. Local methods can also be executed synchronously, following the same model as global methods (e.g., [10, 11, 55, 61]). Some local methods, however, can be performed asynchronously. Processors can initiate load balancing when they become idle, requesting work as they need it. Single neighborhoods may perform load balancing while other processors are computing.

The synchronous model is straightforward to implement, portable, and does not require operating system or application support for handling interrupts to initiate balancing. However, computing time is lost when lightly loaded processors wait to synchronize with heavily loaded processors. The asynchronous model allows processors to acquire work as soon as they become idle. However, it is significantly more difficult to program. Logic must be included to handle interrupts or check for load-balancing messages during the computation. Some implementations use threads to implement asynchronous load-balancing [8, 59, 63]; these implementations may require operating system support and be less portable than synchronous implementations. Other implementations include message checking within the application, complicating the logic of the application [18, 62]. Since many parallel scientific applications have natural synchronization points, synchronous algorithms can often be used successfully, without the added complication of asynchronous algorithms.

Many local methods have two separate steps. In the first step, the algorithm decides how much work should be moved from a processor to each of its neighbors in order to balance the load. In the second step, the algorithm selects the objects (e.g., elements, nodes, particles, surfaces) that will be migrated to satisfy the work transfers determined in the first step. The quality of the partition depends upon this selection process. For both steps of the algorithm, many methods have been suggested.

3.4.1. Determining Work Flow. To determine the flow of work between processors, a *diffusive* algorithm is often used. These algorithms were first proposed by Cybenko [9]. In their simplest form, they model the processor work loads by the heat equation

$$\partial u / \partial t = \alpha \nabla^2 u \tag{3.1}$$

where u is the work load and α is a diffusion coefficient. Using the processors' hardware connections or the application's communication patterns to describe the computational mesh, Eq. (3.1) is solved using a first-order finite difference scheme. Since the stencil of the difference scheme is compact (using information from only neighboring processors), the method is local. The resulting method takes the form

$$u_i^{t+1} = u_i^t + \sum_j \alpha_{ij} (u_j^t - u_i^t) \tag{3.2}$$

where u_i^t is processor i's work load after iteration t, and the sum is taken over all processors j. The weights $\alpha_{ij} \geq 0$ are zero if processors i and j are not connected in the processor graph, and $1 - \sum_j \alpha_{ij} \geq 0$ for every i. The choice of α_{ij} affects the convergence rate of the method. It depends on the processor connectivity due to the architecture or application communication pattern; see [9, 14] for details.

Several methods have been proposed to accelerate the convergence of diffusion methods. For example, diffusion methods have been used with parallel multilevel methods [26, 46, 54, 56], as described in §3.5. Watts, et al. [58, 59] propose using a second-order implicit finite discretization of Eq. (3.1) to compute work transfers. This scheme converges to global balance in fewer iterations, but requires a bit more work and communication per iteration.

The method of Hu and Blake [27] is used in several parallel decomposition packages [46, 55]. They compute a diffusion solution while minimizing the flow of work over the edges of the processor graph, enforcing incrementality. To compute work transfers, they use a more global view of the processors' load distributions. They solve a linear system $Lx = b$, where x is the diffusion solution and b contains the difference between the processors' work loads and the average load. L is a Laplacian matrix. Each diagonal entry l_{ii} is the degree of processor i in the processor graph; non-diagonal entries l_{ij} are -1 if processors i and j are connected in the processor graph and zero otherwise. The system is solved by a conjugate gradient method. The method determines the diffusion coefficients iteratively rather than keeping them fixed as in the diffusion methods above. This method may increase the amount of global communication to obtain global balance, but the additional cost can be worthwhile since the amount of load movement is minimized.

A variation of the diffusion model is a *demand-driven* model, where under-loaded processors request work or overloaded processors export work when they detect that their neighbors have become idle. The result is similar to diffusion algorithms, except that work is transferred to only a subset of neighbors rather than distributed to all neighbors as in the diffusion model. It can increase the size of the maximum work transfer, but can reduce the total number of work transfers per iteration.

There are several implementations of the demand-driven model [18, 34, 60, 61, 62]. For example, Leiss and Reddy [34] use a demand-driven model in neighborhoods that follow the hardware connectivity of the parallel machine. Wheat, et al. [61] extend their definition of a neighborhood to include all processors within the communication connectivity of the application. Each processor requests work from the most heavily loaded processor in its neighborhood. Processors then satisfy requests based on request size, satisfying the largest requests first until all work available for export is exhausted. Processors may not export so much work that their own work loads fall below the neighborhood average; this restriction prevents oscillations of work between processors in subsequent balancing steps. To prevent convergence to an imbalanced state (which can happen in the diffusion algorithms in [9, 26]), every processor that receives a request for work must export at least one object.

In [62], a sender-initiated model is compared with a receiver-initiated model. Both models are asynchronous. In the sender-initiated model, an overloaded processor determines which of its neighbors are less loaded and sends a portion of its excess work to those neighbors. In the receiver-initiated model, an under-loaded processor determines which of its neighbors are more heavily loaded and requests work from those neighbors. The neighbors can satisfy work requests with up to half of their work loads. The receiver-initiated model holds several advantages over the sender-initiated model [62]. First, the majority of the load-balancing overhead is assumed by the lightly loaded receivers. Second, since the models are asynchronous, work load information may be out-of-date by the time it is used. Thus, in the sender-initiated model, a processor may transfer excessive or insufficient amounts of data to under-loaded processors. In the receiver-initiated model, a processor may request excess or insufficient work, but its neighbor will transfer no more than half its own work, thus using up-to-date information in the transfer and reducing the effects of aging.

Another diffusion-like algorithm is *dimensional exchange*, introduced in [9] and analyzed further in [13, 62, 63]. A hypercube architecture is assumed to describe the algorithm. In a loop over hypercube dimensions i, a processor performs load balancing with its neighbor in that dimension, i.e., with the processor whose processor number matches that of the given processor except in bit i. The two processors divide the sum of their loads equally among themselves. After iterating over all dimensions of the hypercube, the system is completely balanced.

Although the dimensional exchange algorithm is described in terms of a hypercube, it can be applied to other architectures such as meshes. However, communication for non-hypercube architectures is non-local, as logical neighbors will not necessarily be physical neighbors. The generalized dimension exchange method [63] suggests using an edge-coloring to maintain nearest-neighbor exchanges in non-hypercube architectures; however, it requires more iterations to reach convergence. More importantly, dimensional exchange may migrate work to distant processors and processors with non-contiguous data regions, increasing communication costs for the application code. This disadvantage outweighs the improved convergence of the method over other diffusion methods.

3.4.2. Selecting Objects to Migrate. The second step of a local method is deciding which objects to migrate to satisfy the work transfers computed in the first step. A number of heuristics have been used to determine which objects should be transferred. Typical goals include minimizing communication costs (through minimizing the number of edge cuts in the application's communication graph), minimizing the amount of data migrated, minimizing the number of neighboring processors, optimizing the shapes of the subdomains, or combinations of these goals.

Many load-balancing algorithms use a version of the gain criteria from the algorithm by Kernighan and Lin (KL) [33] to select objects to transfer (e.g., [11, 24, 55, 61, 63]). For each of a processor's objects, the *gain* of transferring the objects to another processor is computed. For example, to minimize edge-cuts, the gain can be taken as the net reduction of cut edges if the object is transferred to the new target processor. The set of objects with the highest total gain is selected for migration. Objects are selected until the sum of their work loads is approximately equal to the work transfer computed by the first load-balancing phase. In some variants of the KL algorithm only objects on subdomain boundaries are examined for transfer.

The element selection priority scheme of Wheat [60] is an example of a KL-like algorithm with uniform object and edge weights. Gain is measured by edge-cuts in the graph. All transfers of objects are one-directional, so collisions (the simultaneous swapping of adjacent objects between two processors which counteracts their individual gains) do not arise. In [11], Wheat's work is extended by weighting the edges by frequency of communication and the objects by their computational load. To reduce migration costs, high-gain objects with the largest computational loads are selected for migration.

Walshaw, et al. [55] modify the definition of gain to describe a relative gain. The relative gain of an object with respect to a neighboring processor is the object's gain minus the average gain of its neighboring objects in the neighboring processor. This relative gain measures the likelihood of neighboring objects migrating to the object's processor, thus reducing the number of collisions. However, thrashing may occur over several iterations; additional stopping criteria are needed to end the iterations when the cost of the partition has not decreased.

Hammond [22] performs pairwise exchanges of objects between pairs of processors to improve an existing decomposition. The processor graph is edge-colored to allow parallel computation between pairs of adjacent processors throughout the graph. For each pair of processors, the best object to be moved in each direction is selected. If the total gain of moving both objects is positive, the objects are exchanged between the processors. This process is repeated for each color in the processor graph.

Still other variations on KL-based selection strategies are possible. In Schloegel, et al. [46], objects that decrease the edge cut while maintaining graph balance, maintain the edge cut while improving graph balance, or maintain both edge cut and balance while moving the object to its initial processor are chosen for migration. This third condition lowers migration costs by assigning objects to their originating processor whenever possible. Walshaw, et al. [55] migrate objects that improve the edge-cut while maintaining the balance or improve the balance while maintaining the edge cut. Unlike Schloegel, et al., they also migrate any object that improves the balance, even if it results in a higher edge cut.

Most selection methods optimize some measure of the subdomain interface size in an attempt to minimize the application's communication costs. In some applications, however, criteria other than subdomain interface size become important. For domain decomposition linear solvers, for example, the aspect ratio of the subdomains affects the convergence of the solvers. In [12, 53], the cost function to be minimized is a weighted combination of the load imbalance and the subdomain aspect ratio. Thus, objects whose coordinates are farthest from the average coordinates of all the processor's objects are selected for migration. While this criterion is specific to a particular application, it demonstrates that, just as no one load balancing method is suitable for all applications, no one object selection criterion is optimal for all applications.

3.5. Graph partitioners. A number of algorithms and software tools have been developed for the problem of statically partitioning a computational mesh. The most powerful of these algorithms use a graph model of the computation, and apply graph partitioning techniques to divide it among processors. In principle, a graph could be constructed for any computation, but this model is most commonly used for mesh-based applications where the graph is closely related to the mesh. Unlike the approaches described in §3.4, the static partitioning algorithms are global – they examine all of the data at once and try to find the best possible partition.

These static partitioners generally run on sequential machines as a preprocessing step. As a consequence, most of the static algorithms are inappropriate for the dynamic load balancing problem. They are either too expensive or too difficult to parallelize, and they have no notion of incrementality. Despite these problems, there have been several attempts to apply graph partitioning ideas to the dynamic load balancing problem. Although these algorithms tend to be expensive, they can generate high quality partitions. In situations where they are applied infrequently, they can be very useful partitioners.

One of the more popular static partitioning algorithms is known as Recursive Spectral Bisection (RSB) [43, 48]. This approach uses an eigenvector of a matrix associated with the graph to partition the vertices. Although it usually produces high quality partitions, the eigenvector calculation is very expensive. Barnard tried to circumvent this problem via a parallel implementation [3]. The result is primarily useful for static partitioning, but it can also be used in a dynamic setting. However, the eigenvector calculation is very expensive relative to the geometric methods and the local improvement schemes discussed above. Also, the basic RSB algorithm has no mechanism for encouraging a new partition to be an incremental modification of the current one.

In [52], Van Driessche and Roose show how RSB can be modified to include incrementality. In [25], this insight is generalized to apply to a class of graph partitioning algorithms, including the multilevel methods described below.

The most popular static partitioning algorithms are multilevel techniques [6, 24, 30]. These methods construct a sequence of smaller and smaller approximations to the graph. The smallest graph in this sequence is partitioned. Then this partition is propagated back through the intermediate graphs, periodically being refined. Although good sequential implementations of this algorithm have been developed [23, 30], parallel implementations have proved to be quite difficult. Both the construction of the smaller approximations and the refinement operation are challenging to parallelize. These challenges have been addressed in two recent efforts: ParMETIS [31, 32, 46] and JOSTLE [54, 56]. These tools essentially perform a local improvement like those described in §3.4, but they use a multilevel approach to select which objects to move. This makes them more powerful than other local improvement algorithms, but they are also more expensive in both runtime and memory.

In summary, graph partitioning algorithms give the highest quality partitions of any class of dynamic load balancing algorithms, albeit at a high computational cost. In practice, they are restricted to mesh-based calculations. The standard formulations are generally not incremental, but they can often be modified to become so. The interface to a graph partitioning routine is usually significantly more complex than the interface to a geometric partitioner. The construction of the graph takes time and space. We will discuss this issue further in §4. It is our opinion that these limitations, particularly the runtime, will limit the use of graph partitioning algorithms to applications in which they are invoked only infrequently. One important setting for such usage is hybrid methods as discussed in §3.6.

3.6. Hybrid Methods. Several dynamic load balancing algorithms don't conveniently fall into any of the previous sections. One such method is the dynamic spectral algorithm of Simon, Sohn and Biswas [49]. This approach is a hybrid of spectral methods and simple geometric techniques. As a preprocessing step, a few eigenvectors of a matrix associated with the graph are computed. These k eigenvectors provide a mapping from each vertex in the graph to a point in a k dimensional Euclidean space. During the parallel simulation, this Euclidean representation is used to partition the graph using the Recursive Inertial Bisection algorithm described in §3.2. The partition will change as weights associated with the vertices evolve.

This approach leads to partition quality that is nearly as good as Recursive Spectral Bisection, but at a much lower cost. However, it has three shortcomings. One obvious disadvantage is the expense of the up-front calculation of eigenvectors. A second limitation is the restriction of the method to problems in which vertex weights change, but vertex connectivity does not. A third, more minor, problem is that the inertial partitioning technique is not incremental. This could be alleviated by using a different geometric partitioner like Recursive Coordinate Bisection.

A broader class of hybrid methods can be constructed by combining global methods with local ones. Expensive, high-quality algorithms (like graph partitioning) can be used infrequently, while cheaper methods (like local improvement) can be applied more often. Combining the methods gives some of the advantages of both. The high-quality methods will ensure that good partitions are maintained. But their cost will be controlled by using cheaper methods most of the time.

This flexible approach allows a user to trade off quality for partition runtime to optimize the overall performance of an application. Unfortunately, it requires an easy-to-use library with a range of algorithms, all accessible via a simple interface. Designing and implementing such a library is difficult, and none has yet been developed. We will discuss these challenges further in §4.

3.7. Summary. In Table 3.1, we summarize our assessment of the relative merits of the principle algorithms we have described. The criteria we use in the evaluation are those (A-F) which were discussed in §2. We don't include a column for ease-of-use since this is so heavily dependent upon the implementation. But in general, the geometric algorithms are easier to interface with than the graph methods. The algorithms are given from 1 to 3 stars on most criteria, with more stars being better. In the column on incrementality, the incremental algorithms are designated as either (I)mplicit or (E)xplicit.

We have not included a row for hybrid methods in which different algorithms are combined. Although we feel this is a very promising direction, numerous combinations with different characteristics are possible.

Needless to say, this concise representation hides more information than it reveals. The assessments are unavoidably subjective and other researchers may have different opinions.

Method	Balance	Quality	Speed	Memory	Incremental	New Comm.
Master/Slave[1]	***	***	Not Scalable	***	N/A	N/A
Geometric Methods						
RCB/URB	***	*	***	***	I	***
RIB	***	*	**	***	No	**
Octree/SFC	***	*	***	**	I	**
Local Methods						
Diffusion[2]	*[4]	**	***	***	E	*
Demand-Driven[2]	*[4]	**	***	***	E	*
Dim. Exchange[3]	*[4]	**	***	***	E	*
Graph Partitioners						
RSB	***	***	*	*	No	*
Multilevel	***	***	**	*	E	*
Hybrid Methods						
Dynamic Spectral[5]	***	**	**	**	No	*

TABLE 3.1
Summary of dynamic load balancing methods and their characteristics using the criteria in §2.

[1] Limited applicability.
[2] Can be used to only improve balance; can be asynchronous.
[3] Hypercube architectures are best for this algorithm.
[4] Many iterations may be required to obtain balance.
[5] Does not handle changing mesh topology.

4. Software Challenges. There are several widely used software tools for static partitioning, e.g., Chaco [23] and Metis [30]. These tools typically read a graph or geometry from a file, and write the partition to another file. Although there have been a number of research efforts in dynamic load balancing, most of these projects have produced tools that are useful only in a single application or a closely related set of codes [4, 20, 37, 45]. No tool for dynamic load balancing has gained as wide a user base as the static partitioning tools. The principle reason for this is that using a dynamic load balancing tool is much more complex than using a static one. There are a variety of factors that combine to create this complexity, but the primary reason is that dynamic load balancers are invoked as subroutines. The static tools mentioned above use file interfaces, and so avoid the problem of linking with an application.

A second reason why software for the static problem is more mature is that dynamic balancers must be parallel, while static ones can be (and usually are) sequential. Yet a third reason is that there are more objectives to trade off in the dynamic setting. Specifically, while static partitioners need only worry about runtime and quality, dynamic algorithms must also consider the cost of moving the objects. With more objectives to trade off, any particular algorithm is likely to be optimal on only a fraction of the design space. So an ideal tool will have a multiplicity of algorithms, which adds complexity to the development and to the interface.

The key issue in interfacing with a dynamic load balancing tool is the type of data structures required to describe the problem. The time spent forming these data structures and the memory they consume can significantly impact application performance. And the code required to build complex entities like graphs raises a significant barrier to the adoption of a new tool. It is for these reasons that most dynamic load-balancing tools are specific to one application or a set of related codes. These domain-specific tools can work with the native data structures, simplifying the interfaces and improving performance. But these advantages come at a high cost. Each research group has to spend considerable time implementing its own tool. As a consequence, new algorithmic insights are slow to propagate into applications, and few careful comparisons of algorithms can be performed.

One way to avoid some of these problems is to use object-oriented software design. For instance, instead of requiring a full graph description, the load balancing tool could be passed a function that can generate the list of neighbors for any mesh element. If the load balancing algorithm can be constructed out of a few simple functions like this, a new data structure need never be formed explicitly. These functions will refer back to the native data structure instead. Each application code will need to provide these functions, but if they are sufficiently simple the functions should be easier to write than code to create a complex data structure. And duplication of memory is no longer an issue.

An object-oriented approach requires careful design to ensure that the selected functions are simple enough for the application, yet powerful enough for the load balancing algorithms. Also, the cost of many function invocations will slow down the load balancer. But we believe that the ease of migration between applications and the memory savings can more than compensate for the performance degradation.

Object orientation won't solve all of the software challenges associated with dynamic load balancing. There are a number of difficult questions that remain. One important question is the appropriate level of abstraction for a load balancing tool. Throughout this paper we have implicitly assumed that tools should be built around geometric coordinates and/or a graph with objects as vertices. The generality of these models is appealing, but they are also limiting. For instance, when partitioning a mesh, the elements, faces, edges and vertices all get divided (or in some cases shared) among processors. A simple graph is not rich enough to encode all of these relationships. So a load balancing tool focused on mesh applications could choose to use a richer abstraction. Of course, this complicates the interface issue.

A second example comes from work using geometric methods to dynamically load balance the contact detection problem [42]. In this problem, some finite element surfaces need to be shared among several processors. Figuring out which surfaces need to go where involves nontrivial interactions between the application data structures and the load balancer. Compared to a generic load balancer, a tool focusing on this application can provide a higher level of information to the application.

Another important question is what auxiliary functionality a load balancer should provide. For example, an important aspect of any adaptive calculation is a mechanism for deciding when it is worthwhile to rebalance the computation. This decision requires a tradeoff between load imbalance and the cost of rebalancing. A library that helps an application make this decision will be more valuable than one that requires each application to develop this functionality anew.

Another example of possible auxiliary functionality is assistance with the movement of objects. This can be one of the most daunting aspects of developing a parallel adaptive code, and one of the easiest to implement inefficiently. Although some of this functionality will be application specific, a general purpose tool could provide some assistance. A closely related problem is that of determining the new communication pattern after a rebalancing step. Again, a mechanism to assist with this process would add value to a load balancing tool.

5. Conclusions and Future Research Directions. For several reasons, dynamic load balancing is fundamentally more complex than static load balancing. Unlike the static case, the software must be integrated into an application code and it must run in parallel. Runtime is more critical since the cost of load balancing should not outweigh the cost of running the application in an imbalanced state. And the goal of moving only a small amount of data adds an additional dimension to the solution space. Despite these challenges, a number of viable algorithms have been proposed. In our opinion, the two classes of algorithms that are most attractive are incremental local methods and simple geometric approaches like RCB, but the graph partition methods are worthy of consideration too. However, different applications have different properties, so there is a need for a range of algorithms that provide differing tradeoffs between runtime, partition quality and amount of data movement. No single approach will always be best.

The plethora of proposed algorithms is contrasted by a shortage of widely used software. The principle reason for this shortage has to do with the complexity of interfacing an application code with a load balancer. A number of interrelated issues must be addressed. What is the abstraction the balancer works with – a graph, a geometry or something else? Can the balancer work with the application data structures, or does it construct a new data structure? What are the load balancer's costs in time and space? How complicated is the interface to the tool? How much assistance should the load balancer provide for adjusting the application's data structures during and after migration?

Overcoming these problems will require careful design in a dynamic load balancing tool. In our opinion, the following features are important for such a tool.

1. It should contain a variety of algorithms since none will be best in all situa-

tions. Specifically, it should contain both geometrically-based algorithms and topologically-based ones.

2. The interfaces must be kept clean. An appealing way to achieve this objective is to use an object-oriented approach. This can significantly simplify the migration to new applications. But it comes at a price in performance, so very careful design is essential.
3. Whenever possible, it should provide additional functionality such as determination of when to load balance, establishment of new communication patterns, and assistance with data migration.

With the increasing interest in parallel adaptive calculations, the need for a good solution to the dynamic load-balancing problem is becoming more acute. Our current work focuses on the software engineering issues involved in the design of a good, general purpose tool. This tool will be tested in a number of applications, including adaptive finite element methods, particle methods and contact detection. In addition, by implementing several algorithms in the tool, objective comparison of various algorithms can be made in a way that, to date, has been unavailable on a large scale.

Most of the research described in this paper was performed on MIMD computers. Much parallel computing, however, is moving toward clusters of distributed shared memory computers and heterogeneous computing systems. Load balancing on these systems is becoming an active area of research.

A naive approach to load balancing on distributed shared memory systems is to simply change the assignment of work to processors. The global address space of the machines can then locate data when needed by the application, thus skipping complicated data migration. However, since off-processor memory references by the application are expensive, it is advantageous to actually move the assigned data to the processor's own memory. For performance reasons, then, the dynamic load-balancing problem on distributed shared memory systems looks almost identical to that on MIMD computers.

Heterogeneous computer systems, on the other hand, raise interesting issues for dynamic load balancing. The processors in these systems can have different amounts of computing power and memory. This problem can, perhaps, be handled by asking the load balancer for partitions of different sizes, based on the speed and memory capabilities of the target processors. In selecting objects for migration, performance-based load measures, such as execution time per object, can be weighted based on the relative performance of the potential exporting and importing processors. Another complication is the possibility that network connections with different speeds are used in heterogeneous systems. For example, high-speed connections may be used within a cluster, while clusters are connected by low-speed links. This issue raises the complexity of forming subdomains that minimize the application's communication costs. Most scientific applications use communication to periodically update values along the subdomain boundaries. Is it sufficient, then, to make the amount of data to be shared across low-speed links "small" so that the majority of communication is performed over fast connections? If so, how small is sufficiently "small"? We expect these, and related, questions to be active areas of study in the coming years.

Acknowledgments. We are indebted to Steve Plimpton, Rob Leland and Carter Edwards for a number of stimulating conversations on the topics of this paper.

REFERENCES

[1] P. Altevogt and A. Linke, *An algorithm for dynamic load balancing of synchronous monte carlo simulations on multiprocessor systems*, Comp. Phys. Comm., 79 (1994).

[2] P. Baehmann, S. Wittchen, M. Shephard, K. Grice, and M. Yerry., *Robust geometrically based automatic two-dimensional mesh generation*, Intl. J. Numer. Meths. Engrg., 24 (1987), pp. 1043–1078.

[3] S. T. Barnard, *PMRSB: Parallel multilevel recursive spectral bisection*, in Proc. Supercomputing '95, ACM, 1995.

[4] M. Beall and M. Shephard, *Parallel mesh database and parallel load balancing.* `http://www.scorec.rpi.edu/software/Software.html#s2`.

[5] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Computers, C-36 (1987), pp. 570–580.

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

[7] F. Cao, J. R. Gilbert, and S.-H. Teng, *Partitioning meshes with lines and planes*, Tech. Rep. CSL–96–01, Xerox PARC, 1996. Submitted for publication.

[8] N. Chrisochoides, *Multithreaded model for dynamic load balancing parallel adaptive PDE computations*, ICASE Report 95-83, ICASE, NASA Langley Research Center, Hampton, VA 23681-0001, Dec. 1995.

[9] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[10] H. deCougny, K. Devine, J. Flaherty, R. Loy, C. Ozturan, and M. Shephard, *Load balancing for the parallel adaptive solution of partial differential equations*, Appl. Numer. Math., 16 (1994), pp. 157–182.

[11] K. Devine and J. Flaherty, *Parallel adaptive* hp-refinement techniques for conservation laws, Appl. Numer. Math., 20 (1996), pp. 367–386.

[12] R. Diekmann, D. Meyer, and B. Monien, *Parallel decomposition of unstructured fem-meshes*, in Proc. Parallel Algorithms for Irregularly Structured Problems, Springer LNCS 980, 1995, pp. 199–216.

[13] R. Diekmann, B. Monien, and R. Preis, *Load balancing strategies for distributed memory machines*, Tech. Rep. tr-rsfb-97-050, Dept. of Computer Science, University of Paderborn, Paderborn, Germany, Sept. 1997.

[14] R. Diekmann, S. Muthukrishnan, and M. Nayakkankuppam, *Engineering diffusive load balancing algorithms using experiments*, in Proc. IRREGULAR '97, Springer LNCS 1253, 1997, pp. 111–122.

[15] J. Eckstein, *Parallel branch-and-bound methods for mixed-integer programming on the cm-5*, SIAM J. Optimization, 4 (1994), pp. 794–814.

[16] ———, *Distributed versus centralized storage and control for parallel branch and bound: Mixed integer programming on the cm-5*, Computational Optimization and Applications, 7 (1997), pp. 199–220.

[17] H. C. Edwards, *A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its Application to Least Squares C^∞ Collocation*, PhD thesis, The University of Texas at Austin, May 1997.

[18] R. Enbody, R. Purdy, and C. Severance, *Dynamic load balancing*, in Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing, SIAM, February 1995, pp. 645–646.

[19] J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco, and L. Ziantz, *Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws*, J. Parallel Distrib. Comput., 47 (1998), pp. 139–152.

[20] L. Freitag, C. Ollivier-Gooch, M. Jones, and P. Plassmann, *Sumaa3d.* `http://www.mcs.anl.gov/Projects/mesh/index.html`.

[21] J. Gustafson, R. Benner, M. Sears, and T. Sullivan, *A radar simulation program for a 1024-processor hypercube*, in Proc. Supercomputing '89, ACM Press, 1989, pp. 96–105.

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

[23] B. Hendrickson and R. Leland, *The Chaco user's guide, version 2.0*, Tech. Rep. SAND94–2692, Sandia National Laboratories, Albuquerque, NM, October 1994.

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

[25] B. Hendrickson, R. Leland, and R. Van Driessche, *Skewed graph partitioning*, in Proc. Eighth SIAM Conf. Parallel Processing for Scientific Computation, March 1997.

[26] G. Horton, *A multi-level diffusion method for dynamic load balancing*, Parallel Computing, 19 (1993), pp. 209–218.

[27] Y. Hu and R. Blake, *An optimal dynamic load balancing algorithm*, Tech. Report DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK, Dec. 1995.

[28] G. L. M. J. R. Gilbert and S. Teng, *Geometric mesh partitioning: Implementation and experiments*, in Proc. 9th Intl. Parallel Processing Symp., IEEE Computer Society Press, 1995, pp. 418–427.

[29] M. T. Jones and P. E. Plassmann, *Computational results for parallel unstructured mesh computations*, Computing Systems in Engineering, 5 (1994), pp. 297–309.

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

[31] ———, *A coarse-grain parallel multilevel k-way partitioning algorithm*, in Proceedings of the eighth SIAM conference on Parallel Processing for Scientific Computing, 1997.

[32] G. Karypis and V. Kumar, *Parmetis: Parallel graph partitioning and sparse matrix ordering library*, Tech. Rep. 97-060, Department of Computer Science, University of Minnesota, 1997. Available on the WWW at URL `http://www.cs.umn.edu/~metis`.

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

[34] E. Leiss and H. Reddy, *Distributed load balancing: design and performance analysis*, W.M. Keck Research Computation Laboratory, 5 (1989), pp. 205–270.

[35] G. L. Miller, S.-H. Teng, and S. A. Vavasis, *A unified geometric approach to graph separators*, in Proc. 32nd Symp. Foundations of Computer Science, IEEE, October 1991, pp. 538–547.

[36] S. A. Mitchell and S. A. Vavasis, *Quality mesh generation in three dimensions*, in Proc. 8th ACM Symposium on Computational Geometry, ACM, 1992, pp. 212–221.

[37] S. Mitra, M. Panishar, and J. Browne, *Dagh: User's guide*. `http://www.cs.utexas.edu/users/dagh/`.

[38] J. T. Oden, A. Patra, and Y. Feng, *Domain decomposition for adaptive* hp *finite element methods*, in Proc. Seventh Intl. Conf. Domain Decomposition Methods, State College, Pennsylvania, October 1993.

[39] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski, *A common data management infrastructure for adaptive algorithms for PDE solutions*, in Proc. SC97, San Jose, CA, 1997.

[40] A. Patra and J. T. Oden, *Problem decomposition for adaptive* hp *finite element methods*, J. Computing Systems in Engg., 6 (1995).

[41] J. R. Pilkington and S. B. Baden, *Partitioning with spacefilling curves*, CSE Technical Report CS94–349, Dept. Computer Science and Engineering, University of California, San Diego, CA, 1994.

[42] 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., (1998). To appear.

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

[44] E. Pramono, H. Simon, and A. Sohn, *Dynamic load balancing for finite element calculations on parallel computers*, in Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computation, SIAM, 1995, pp. 599–604.

[45] D. Quinlan, *Overture*. `http://www.c3.lanl.gov/~dlb/Web_papers/Overture/OverturePP/sld025.htm`.

[46] K. Schloegel, G. Karypis, and V. Kumar, *Multilevel diffusion algorithms for repartitioning of adaptive meshes*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 109–124.

[47] M. Shephard and M. Georges, *Automatic three-dimensional mesh generation by the finite octree technique*, Intl. J. Numer. Meths. Engrg., 32 (1991), pp. 709–749.

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

[49] H. D. Simon, A. Sohn, and R. Biswas, *HARP: A fast spectral partitioner*, in Proc. Ninth ACM Symp. Parallel Algorithms & Architectures, ACM, 1997.

[50] S. G. Srinivasan, I. Ashok, H. Jonsson, G. Kalonji, and J. Zahorjan, *Dynamic-domain-decomposition parallel molecular-dynamics*, Computer Physics Comm., 103 (1997), pp. 44–58.

[51] S. Ueng and K. Sikorski, *Parallel visualization of 3d finite element analysis data*, in Proc. Seventh SIAM Conf. Parallel Processing for Scientific Computation, SIAM, 1995, pp. 808–813.

[52] R. Van Driessche and D. Roose, *Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem*, in High-Performance Computing and Networking, no. 919 in Lecture Notes in Computer Science, Springer, 1995, pp. 392–397. Proc. International Conference and Exhibition, Milan, Italy, May 1995.

[53] D. Vanderstraeten, C. Farhat, P. Chen, R. Keunings, and O. Ozone, *A retrofit based methodology for the fast generation and optimization of large-scale mesh partitions: beyond the minimum interface size criterion*, Comput. Methods Appl. Mech. Engrg., 133 (1996), pp. 25–45.

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

[55] C. Walshaw, M. Cross, and M. Everett, *Parallel dynamic graph-partitioning for unstructured meshes*, Mathematics Research Report 97/IM/20, Centre for Numerical Modeling and Process Analysis, University of Greenwich, London, SE18 6PF, UK, March 1997.

[56] C. Walshaw, M. Cross, and M. Everett, *Parallel dynamic graph-partitioning for unstructured meshes*, J. Par. Dist. Comput., (1998). To appear.

[57] M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, in Proc. Supercomputing '93, Portland, OR, Nov. 1993.

[58] J. Watts, *A practical approach to dynamic load balancing.* Master's Thesis, October 1995.

[59] J. Watts, M. Rieffel, and S. Taylor, *A load balancing technique for multiphase computations*, in Proc. High Performance Computing '97, Society for Computer Simulation, 1997, pp. 15–20.

[60] S. Wheat, *A fine-grained data migration approach to application load balancing on MP MIMD machines*, PhD thesis, The University of New Mexico, Dept. of Computer Science, Albuquerque, NM, 1992.

[61] S. Wheat, K. Devine, and A. Maccabe, *Experience with automatic, dynamic load balancing and adaptive finite element computation*, in Proc. 27th Hawaii International Conf. on System Sciences, IEEE, January 1994, pp. 463–472.

[62] M. Willebeek-LeMair and A. Reeves, *Strategies for dynamic load balancing on highly parallel computers*, IEEE Parallel and Distrib. Sys., 4 (1993), pp. 979–993.

[63] C. Xu, F. Lau, and R. Diekmann, *Decentralized remapping of data parallel applications in distributed memory multiprocessors*, Tech. Rep. tr-rsfb-96-021, Dept. of Computer Science, University of Paderborn, Paderborn, Germany, Sept. 1996.