# Parallel Dynamic Load Balancing Strategies for Adaptive Irregular Applications

Rupak Biswas

*MRJ Technology Solutions, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035. E-mail:* rbiswas@nas.nasa.gov

Sajal K. Das, Daniel J. Harvey

*Department of Computer Science & Engineering, The University of Texas at Arlington, Arlington, TX 76019. E-mail:* {das,harvey}@cs.unt.edu

Leonid Oliker

*NERSC, MS 50F, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail:* loliker@lbl.gov

**Abstract**

Dynamic unstructured mesh adaptation is a powerful technique for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult because of the load imbalance that mesh adaptation creates. To address this problem, we have developed two dynamic load balancing strategies for parallel adaptive irregular applications. The first, called PLUM, is an architecture-independent framework particularly geared toward adaptive numerical computations and requires that all data be globally redistributed after each adaptation to achieve load balance. The second is a more general-purpose topology-independent load balancer that utilizes symmetric broadcast networks (SBN) as the underlying communication pattern, with a goal to providing a global view of system loads across processors. Results indicate that both PLUM and the SBN approach are portable and remain viable on large numbers of processors, and that they achieve excellent load balance at the cost of minimal extra overhead.

*Key words:* Unstructured mesh adaptation, global dynamic load balancing, symmetric broadcast networks, parallel computing

## 1 Introduction

The ability to dynamically adapt an unstructured mesh is a powerful tool for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult because of the load imbalance that mesh adaptation creates. Dynamic load balancing aims to balance processor workloads at runtime while attempting to minimize the communication between processors. A problem is therefore load balanced when processors have nearly equal loads with reduced communication among themselves. The ultimate objective, of course, is to assign work to processors so that the total simulation runtime is minimized. With the proliferation of parallel computing, dynamic load balancing has also become extremely important in areas other than scientific computing like task scheduling, discrete event simulation, and web server applications.

Standard fixed-mesh numerical methods to solve a full-scale realistic problem can be made more cost-effective by locally refining and coarsening the computational mesh to capture physical phenomena of interest. However, when executed on a parallel machine, mesh adaptation could cause load imbalance among the processors. This, in turn, would lead to idle processors and adversely affect the total execution time. It is therefore imperative to have an efficient dynamic load balancing mechanism as part of the overall solution scheme. However, since the computational mesh will be frequently adapted for unsteady flows, the runtime load may also need to be frequently rebalanced (in the worst case, at each adaptation step). This implies that the dynamic load balancing procedure itself must not pose a major overhead. Although several dynamic load balancers have been proposed for multiprocessor platforms [1–6], most of them are inadequate for adaptive unstructured grid applications. This motivates our work.

Recently, we have developed a novel method, called PLUM [7], that dynamically balances processor workloads with a global view when performing adaptive numerical calculations in a parallel message-passing environment. The computational mesh is globally repartitioned from scratch after each adaptation, but a smart remapping technique is used to reassign the new partitions among the processors in a way that minimizes the data movement overhead. This redistribution cost can be estimated using a variety of metrics. The new partitioning is accepted if the migration cost is offset by the gain resulting from a better load balance. Results on a T3E, an SP2, and an Origin2000 show that PLUM is portable and remains viable on large numbers of processors.

We also describe another new dynamic load balancing strategy that is based on utilizing a robust communication pattern among processors, called symmetric broadcast networks (SBN), originally proposed in [8]. Our earlier experiments

with synthetic loads [2] have demonstrated that an SBN-based load balancer achieves superior performance when compared to other popular techniques [9–12]. The SBN-based algorithm is adaptive and decentralized in nature, and can be ported to any topological architecture through efficient embedding techniques. It has been implemented on an SP2 and an Origin2000, and its performance analyzed for an adaptive unsteady workload which is generated by propagating a simulated shock wave through a cylindrical volume. Results show that our approach reduces the redistribution cost at the expense of a minimal extra communication overhead. In many irregular applications in which the data redistribution cost dominates the processing and communication costs, this is an acceptable trade-off.

The work reported in this paper is also relevant in the context of the Information Power Grid (IPG) [13] which is intended to provide a convenient interface to the vast, heterogeneous, and geographically-distributed computing resources of NASA and its partners. The interface will hide details of machine particulars, such as location, size, connectivity, and name, thereby presenting a unified virtual machine. The Globus metacomputing toolkit [14], a joint effort by Argonne National Laboratory and the University of Southern California's Information Sciences Institute, provides a basic software infrastructure for computations that integrate distributed computational and information resources. It is, at present, the most realistic starting point for the implementation of the IPG. However, there are a number of software challenges that must be overcome before actual applications can be run efficiently in a parallel distributed fashion on the IPG. One of the most important issues will be to develop efficient architecture-independent solutions for real-life problems running on various platforms. Clearly, dynamic load balancing in such a heterogeneous environment will play a central role in the overall performance.

This paper is organized as follows. Section 2 deals with some basic concepts of dynamic load balancing for adaptive unstructured grids, the parallel architectures and programming models that were used, and a description of the workloads that were tested in the experiments reported in this paper. Section 3 describes the PLUM load balancer, and presents performance results on three parallel machines for an actual workload. Section 4 deals with the SBN-based load balancer based on a topology-independent communication pattern, and reports results for a simulated workload. Section 5 concludes the paper.

## 2 Preliminaries

This section addresses some of the basic issues involved in the dynamic load balancing of adaptive unstructured grids, surveys the parallel machine architectures that were considered for implementing our methods, reviews the

programming model that was used to facilitate portability of the developed software, and describes the actual workloads used in our experiments.

### *2.1 Dynamic Load Balancing*

Load balancing for unstructured grids essentially consists of two components: partitioning the computational mesh (to balance processor workloads and minimize runtime interprocessor communication), and mapping the partitions to processors (to minimize data movement). Diffusive partitioners modify existing partitions, while global partitioners generate new partitions from scratch that need to be remapped to processors.

#### *2.1.1 Partitioning*

Graph partitioning is NP-complete; thus, research has been focused on developing efficient heuristic algorithms. Several graph partitioning algorithms have been developed over the years, particularly for static grids. Significant progress has been made in improving the partitioning heuristics as well as in generating high-quality software.

The most general approach to graph partitioning is to use generic combinatorial optimization techniques based on cost functions. Simulated annealing [15] and genetic algorithms [16] are two techniques that yield good suboptimal solutions. Both methods require properly setting a large number of parameters that makes it difficult to obtain successful partitionings efficiently; however, they are usually very effective in fine tuning existing partitions.

Clustering is another intuitive approach to graph partitioning. The greedy algorithm described in [17] grows the first partition from a given starting point. Each new partition is then begun from the boundary of the previous partition until the whole domain is decomposed. Bandwidth reduction algorithms like Reverse Cuthill-McKee (RCM) [18] also belong to this class of partitioners. The bad aspect ratio problem associated with RCM can be somewhat reduced if the scheme is used recursively, as in recursive graph bisection (RGB) [19].

Geometry-based algorithms partition the graph by exploiting its geometric properties. Recursive coordinate bisection (RCB) [19] uses spatial geometric coordinates, whereas recursive inertial bisection (RIB) [20] is based on inertial coordinates.

A completely different class of algorithms is based on spectral methods. Recursive spectral bisection (RSB) [19] is the most famous of such partitioners, and uses a strategy based on the Fiedler vector of the Laplacian of the graph.

Multidimensional spectral partitioning [21] improves RSB runtimes by performing a $k$-way partitioning at each recursive step. Many of these geometric and spectral methods are used with the Kernighan-Lin refinement strategy [22] to improve the partitioning quality.

However, the most successful state-of-the-art partitioners use multilevel algorithms [23–25] that reduce partitioning times considerably by contracting the graph, partitioning the coarsened graph, and then refining it to obtain a partition for the original graph. ParMETIS [24] and JOSTLE [25] are currently the fastest multilevel schemes, are available in various flavors, and can be run either as partitioners from scratch or as diffusive repartitioners.

### *2.1.2 Remapping*

The overall effectiveness of repartitioning algorithms is determined by how successful they are in load balancing the computations while minimizing the edge-cut, as well as reducing the cost associated with redistributing the load in order to realize the new partitioning. Data redistribution is generally considered the most expensive phase in dynamic load balancing. The migration of mesh objects incurs a number of costs such as the communication overhead of remote-memory latency, and the computational overhead of rebuilding internal and shared data structures. Since global partitioners do not consider the redistribution cost when generating subdomains, an intelligent algorithm is needed to map the new partitions onto the processors such that the data migration cost is minimized. Diffusive repartitioners explicitly attempt to minimize this data redistribution cost when generating new subdomains. These strategies are generally successful when there are gradual changes in the load distribution. However, experimental results [26,27] have indicated that diffusive schemes may not be well suited for problems which incur dramatic shifts in processor workloads between redistributions, such as unsteady adaptive applications.

Diffusion was first presented as a method for load balancing in [1]. The process can be mapped onto the diffusion equation, and much is known about its properties. In particular, it can be shown that this process will eventually converge. A nonlinear variant of this scheme which considers strip decompositions of the domain was presented in [5]. This algorithm shows better convergence characteristics than the standard diffusion method.

Tiling is another approach to dynamic load balancing [28]. Each processor is considered a neighborhood center, where a neighbor is defined as that processor and all others that share its subdomain boundaries. Processors within a neighborhood are balanced with respect to each other using local performance measurements. Task migration occurs from heavily-loaded to under-

loaded neighbors within each neighborhood. The iterative process continues until the load is globally balanced. Iterative tree balancing (ITB) [29] follows the basic tiling strategy; however, the algorithm views workload requests as forming a forest of trees rather than considering a neighborhood of processors. Each tree is then linearized, and a scan operation is used to determine the amount of data migration. ITB incorporates a more global information, and has an improved worst case load imbalance over the flat tiling model if enough iterations are permitted.

Local iterative techniques are sometimes unsuitable for dynamically balancing unsteady numerical calculations. Such applications are prone to shifting the load distribution from one adaptation phase to the next, causing small regions of the domain to suddenly incur high computational costs. Local diffusion techniques would require several iterations before global convergence, or accept an unbalanced load in exchange for faster performance. Also, by limiting load movement to nearest neighbors, several hops may be needed for a work unit to arrive at its final destination. Since the remapping must be frequently applied, its cost can become a significant part of the overall performance and must therefore be minimized. By moving large chunks of work units directly to their destinations, the high start-up cost of interprocessor communication can be amortized. This is the strategy that has been implemented within the PLUM and SBN frameworks.

### *2.2 Parallel Machines Considered*

The dynamic load balancing methods proposed in this paper are implemented on three state-of-the-art commercial parallel machines: an IBM SP2, an SGI Origin2000, and a Cray T3E. The SP2 and T3E belong to the class of what is commonly known as distributed-memory multicomputers, whereas the Origin2000 is a distributed shared-memory computer. The architectural features of these machines are briefly discussed below for the sake of completeness.

#### *2.2.1 Distributed-Memory Multicomputers*

In a distributed-memory or message-passing architecture, each processor has its own local memory which only it can access directly. Since there is no common bus, remote memory is not directly addressable, and data are shared by explicitly sending and receiving messages. In this message-passing model, data transfers usually require operations to be performed by both the sending and the receiving processors. Such architectures are inherently scalable because they do not require tight system coordination for maintaining address space or data coherence. A key performance advantage of these machines is the abil-

ity for the programmer to explicitly associate specific data with processes. This allows the system to effectively use the caches and memory hierarchy on each processing unit. Unfortunately, writing message-passing code is a complex and cumbersome task since the user must explicitly partition the data among the local processors. Once a data structure has been distributed, all sections of the code using that data must be implemented in parallel. Thus, it is very difficult to incrementally parallelize a serial code using this paradigm.

The IBM 9076 Scalable POWERParallel SP2 is an example of a distributed-memory parallel system. The nodes, each consisting of a six-instruction issue superscalar RISC6000 processor, are connected through a high-performance switch called the Vulcan chip. The topology of the switch is an any-to-any packet switched network similar to an Omega network. An advantage of this interconnection mechanism is that the available bandwidth between any pair of communicating nodes remains constant, regardless of where in the topology the two nodes lie. [1]

The Cray T3E is a scalable parallel system with a high-performance, globally addressable memory. The processors are cache-based RISC-type DEC chip 21164 (DEC Alpha EV5), have pipelined functional units, and issue multiple instructions per cycle. The network is a high-bandwidth, low-latency bidirectional 3-D torus which can accommodate up to 2,048 processors. The T3E uses adaptive routing to allow messages on the interconnection network to be rerouted around temporary hot spots.

### *2.2.2 Distributed Shared-Memory/CC-NUMA Machines*

A distributed shared-memory architecture combines physically distributed memory together with hardware which treats it as a unified, global address space. Thus, all of the physical memory is directly addressable from any processor. These systems have the advantage of efficiently supporting a shared-memory programming model, while preserving scalability. A drawback, however, is the overhead necessary for maintaining address and data coherence. To enable a global address space, the system must be tightly connected at either the software or the hardware level. This overhead can degrade performance with increasing number of processors.

An example of a distributed shared-memory architecture is the SGI Origin2000, the first commercially available 64-bit cache-coherent nonuniform memory access (CC-NUMA) system. A small high-performance switch connects two CPUs (which are superscalar RISC processors), memory, and I/O.

[1] The same does not hold true for direct networks such as rings, meshes, or multidimensional toroids which require an additional performance optimization to consider the number of hops between two communicating nodes.

This module, called a node, is then connected to other nodes in a hypercube fashion. An advantage of this interconnection network is that additional nodes and switches can be added to create larger systems that scale with the number of processors. The memory access time is no longer uniform as in a true shared-memory system, and varies depending on its location. Each processor in the Origin2000 has a private cache; specialized hardware maintains the image of a single shared memory as well as cache coherence.

### *2.3 Message-Passing Programming Model*

One of the fundamental issues in parallel computing is the choice of programming paradigms. Several paradigms have evolved over the last two decades, which include shared-memory programming, message-passing protocols, active messages, data parallel programming, multithreading, and so on. Some of these paradigms have turned out to be architecture-independent in the sense that parallel programs, software, and tools developed with their help can be easily and efficiently ported onto various machine platforms. In this paper, we use the message-passing programming model.

Message passing is a popular paradigm for a wide range of parallel systems since it offers programmers direct access to communication between processes on different machines. With the explicit message-passing model, processes in a parallel application have their own private address spaces and share data via explicit messages. Several vendors have demonstrated that a message-passing library can be efficiently implemented and made portable. However, because of the need for explicit communication, message passing presents a complex programming interface and is difficult to debug. Furthermore, the performance of an application depends heavily on the expertise of the programmer. Programs expressed in this manner may run on distributed-memory multiprocessors, distributed shared-memory machines, or a heterogeneous network of workstations. Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) are the two most popular standards for writing message-passing codes, and have been successfully ported onto a large number of systems including distributed-memory multicomputers and distributed shared-memory/CC-NUMA machines. Although we have conducted extensive experiments with both PVM and MPI, we prefer MPI since it has shown superior results on a variety of high-performance architectures.

#### *2.3.1 Message Passing Interface (MPI)*

MPI is currently the most widely-used standard for writing message-passing programs, with its MPI-2 specifications released in 1997. The main goals for

designing this interface are portability, efficiency, and programmability. MPI provides the syntactic and semantic core of library routines for a distributed-memory communication environment. Providing this high-level standardization allows an efficient implementation of lower-level message-passing functions, and the possibility of hardware support by various vendors. MPI is suitable for use by general MIMD programs, as well as those written in the more restricted SPMD style. It is a flexible communication library which can be used in a heterogeneous environment and contains bindings to both C and Fortran77. MPI-2 extends the language bindings to C++ and Fortran90.

MPI provides an interface that allows processes in a parallel program to communicate with one another. In MPI-1, processes cannot be added to or deleted from an application after it has been started. The MPI-2 process model, however, allows the dynamic creation and termination of processes after an application has started (similar to that allowed by PVM). Various communication primitives are supported within MPI. Point-to-point communications allow simple blocking and nonblocking send/receive primitives. Collective communications involve groups of processes, and allow operations such as barrier synchronizations, gather/scatters, and global reductions. MPI-2 also provides one-sided communications that allow a process to specify all communication parameters for both the sending and the receiving sides. This is a powerful construct which enables programmers to take advantage of fast communication mechanisms provided by various platforms, such as coherent or non-coherent shared memory, DMA engines, hardware-supported put/get operations, and communication coprocessors.

### *2.4 Real Workloads*

Two sets of experiments were performed to study the effectiveness of our load balancers on realistic-sized meshes. We first examined the performance of PLUM on a steady-state, helicopter rotor test case. Next, we evaluated the efficacy of SBN in an unsteady environment where a simulated shock wave moved across a uniform domain over nine levels of mesh adaptation. Results demonstrate that these two dramatically different approaches to dynamic load balancing can be successfully used for adaptive unstructured problems, in an architecture-independent fashion.

#### *2.4.1 Workload-1 (Helicopter Rotor Test Case)*

The computational mesh used to evaluate PLUM is the one used to simulate an acoustics wind-tunnel experiment of a UH-1H helicopter rotor blade [30]. A 1/7th-scale model was tested over a range of rotational speeds. A cut-out

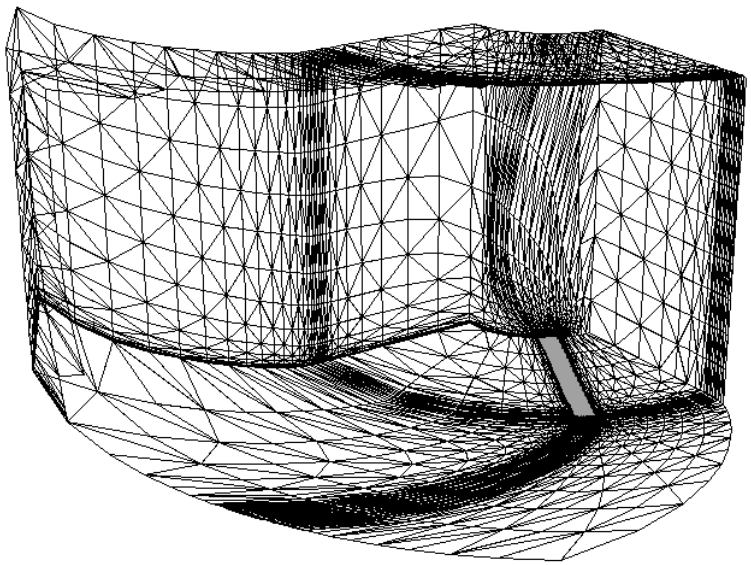

Fig. 1. Cut-out view of the initial tetrahedral mesh.

view of the initial tetrahedral mesh, consisting of 60,968 elements, is shown in Fig.1. The domain was refined based on an error indicator calculated directly from the flow solution, that increased the number of elements to 321,841.

### *2.4.2 Workload-2 (Unsteady Simulation Test Case)*

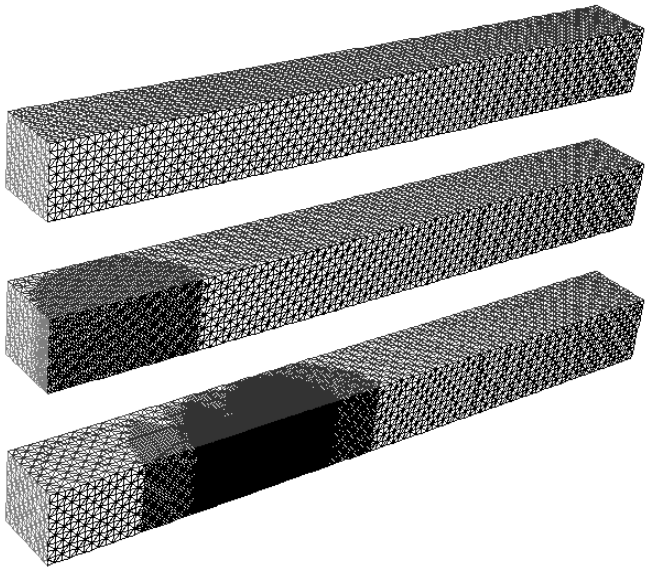

Fig. 2. Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

To evaluate the SBN framework, a computational grid is used to simulate an unsteady environment where the adapted region is strongly time-dependent. This experiment is performed by propagating a simulated shock wave through the initial grid shown at the top of Fig. 2. The test case is generated by

refining all elements within a cylindrical volume moving left to right across the domain with constant velocity, while coarsening previously-refined elements in its wake. Performance is measured at nine successive adaptation levels. The size of the computational mesh increased from 50,000 elements to 1,833,730 over the nine levels of adaptation.

## 3 Architecture-Independent PLUM Load Balancer

PLUM is an automatic and portable load balancing environment, specifically created to handle adaptive unstructured grid applications. It differs from most other similar load balancers in that it dynamically balances processor workloads with a global view. Prior work [7,26,31] has successfully demonstrated the viability and verified the effectiveness of PLUM for various test cases involving adaptive unstructured grids. In this paper, we examine its architecture-independent feature by comparing results on three parallel machines.

Figure 3 provides an overview of PLUM. After an initial partitioning and mapping of the unstructured grid, a solver executes several iterations of the application. A mesh adaptation procedure is invoked when the computational mesh needs to be refined or coarsened. PLUM then gains control to determine if the workload among the processors has become unbalanced due to the mesh adaptation, and to take appropriate action if necessary. If load balancing is required, the adapted mesh is repartitioned and reassigned among the processors so that the cost of data movement is minimized. If the expected gain resulting from a better load balance exceeds the estimated remapping cost, the grid is remapped among the processors before resuming the computation. A brief description of the salient features of PLUM is given in the following subsections.

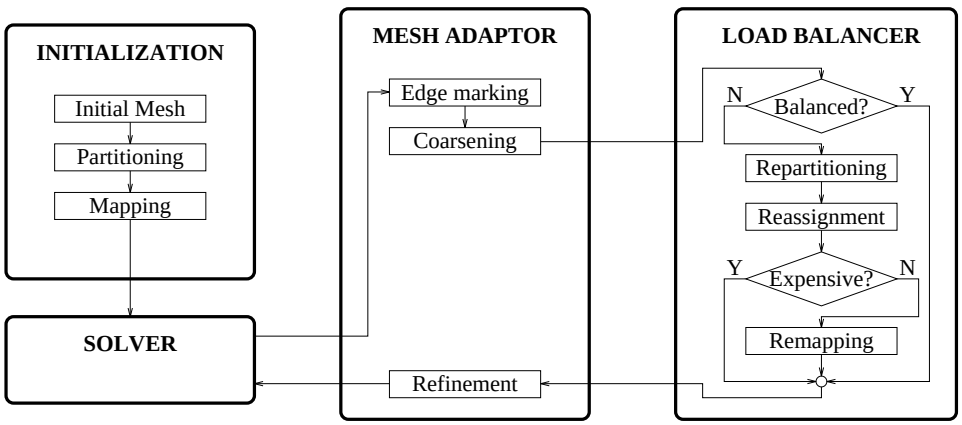


Fig. 3. Overview of PLUM load balancer.

### *3.1 Reusing the Initial Dual Graph*

PLUM always uses the dual of the initial mesh for load balancing, thus keeping the complexity of the partitioning and reassignment phases constant during the course of an adaptive computation. New computational grids obtained by adaptation are translated by changing the weights of the vertices and edges of the dual graph. The weight, $W_v$, of a dual graph vertex $v$ models the computational workload, and is set to the number of leaf elements in the corresponding refinement tree. This is because only those elements with no children participate in the actual computation. The weight, $R_v$, of $v$ models the redistribution cost, and is the total number of elements in the refinement tree. This is because all descendents of the root element must be moved from one partition to another when the load is to be rebalanced. Lastly, the weight, $C_e$, of a dual graph edge $e$ models the communication cost, and is set to the number of corresponding faces in the computational mesh. These three weights are used to balance the workload among processors, minimize the runtime communication, and optimize the data movement cost after a repartitioning.

### *3.2 Parallel Repartitioning*

After a mesh adaptation, PLUM usually needs to rebalance the processor workloads and minimize the interprocessor communication. PLUM can use any general-purpose partitioner; however, for it to be viable, the repartitioning must be performed rapidly. As discussed in Section 2.1.1, several excellent parallel partitioners are now available. For the results presented in this paper, the ParMETIS parallel multilevel partitioner was used [24]. It reduces the size of the graph by collapsing vertices and edges, partitions the smaller problem, and uncoarsens the graph back to the original size. ParMETIS uses a greedy graph bisection algorithm for partitioning the coarsest graph, and uncoarsens it by using a combination of boundary greedy and Kernighan-Lin [22] refinement. A detailed performance comparison of ParMETIS with other partitioners is given in [26].

### *3.3 Processor Remapping*

The goal of processor reassignment is to find a mapping between partitions and processors that minimize the data redistribution cost. In theory, the number of new subdomains can be an integer multiple $F$ of the number of processors. Each processor is then assigned $F$ unique subdomains. The rationale for allowing multiple subdomains per processor is that remapping at a finer granularity reduces the volume of data movement at the cost of a slightly larger

partitioning time. We first compute a similarity measure that indicates how the new subdomains are distributed over the $P$ processors. The measure is represented as a matrix $S$, where entry $S_{ij}$ is the sum of the $R_v$ values of all the dual graph vertices in new partition $j$ that already reside on processor $i$. Figure 4 shows an example of a similarity matrix for four processors where two partitions are assigned to each processor. Only the non-zero entries are shown for clarity.

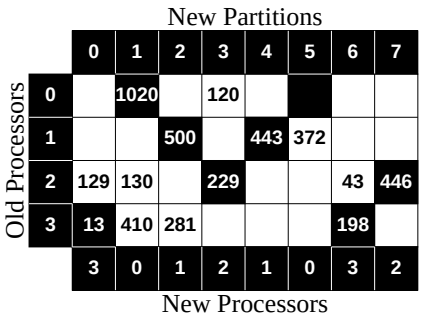


Fig. 4. A similarity matrix after processor reassignment.

Various cost functions are usually needed to solve the processor reassignment problem using $S$ for different machine architectures. We have developed three general metrics: `TotalV`, which minimizes the total volume of data moved among all the processors; `MaxV`, which minimizes the maximum flow of data to *or* from any single processor; and `MaxSR`, which minimizes the sum of the maximum flow of data to *and* from any processor. Experimental results [7,31] have indicated the usefulness of these metrics in predicting the actual remapping cost.

The `TotalV` metric assumes that remapping time will be reduced by decreasing network contention and the total number of elements moved. Finding the optimal mapping for `TotalV` can be reduced to solving the *maximally weighted bipartite graph* (MWBG) problem. The optimal algorithm has been implemented with a runtime of $O(|V|^3)$ [31], where $V$ represents the number of partitions. The metric `MaxV`, unlike `TotalV`, considers data redistribution in terms of solving a load imbalance problem, where it is more important to minimize the workload of the most heavily-weighted processor than to minimize the sum of all the loads. We can solve for the `MaxV` metric optimally by considering the problem of finding a maximum-cardinality matching whose maximum edge cost is minimum. We refer to this as the *bottleneck maximum cardinality matching* (BMCM) problem. The optimal BMCM algorithm has been implemented with a runtime of $O(|V|^{1/2}|E|\log|V|)$ [31], where $|E|$ is the number of entries in the similarity matrix ($|E| \approx |V|^2$). Our third metric, `MaxSR`, is similar to `MaxV` in the sense that the overhead of the bottleneck processor is minimized during the remapping phase. `MaxSR` differs, however, in that it minimizes the sum of the heaviest data flow *from* any processor and

*to* any processor. This is referred to as the *double* bottleneck maximum cardinality matching (DBMCM) problem. We have developed an algorithm for computing the minimum DBMCM with a runtime of $O(|V|^{1/2}|E|^2 \log |V|)$ [31].

A greedy heuristic algorithm for solving the reassignment problem in $O(|E|)$ steps has also been developed, and shown to generate a solution that can never result in a data movement cost that is more than twice that of the optimal `TotalV` assignment [7]. A detailed comparison of reassignment algorithms using different metrics is given in [31].

### *3.4 Remapping Cost Model*

Our redistribution algorithm consists of three major steps: first, the data objects moving out of a partition are stripped out and placed in a buffer; next, a collective communication appropriately distributes the data to its destination; and finally, the received data is integrated into each partition and the boundary information is consistently updated. Performing the remapping in this bulk fashion, as opposed to sending small individual messages, has several advantages including the amortization of message start-up costs and good cache performance. However, a model is needed to quickly predict the expected redistribution cost for a given architecture.

The expected time for the redistribution procedure can be expressed as $\gamma \times \texttt{MaxSR} + O$, where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, $\gamma$ represents the total computation and communication cost to process each redistributed element, and $O$ is the predicted sum of all constant overheads [7]. This model demonstrates the need to use the `MaxSR` metric for processor reassignment because the computational overhead of our remapping algorithm can be reduced by minimizing `MaxSR`. Since the computational workload is architecture independent, we are effectively solving two load balancing problems separated by a collective communication. Moreover, by reducing `MaxSR`, we can achieve a savings in the communication overhead on many bandwidth-rich systems. The values of $\gamma$ and $O$ can be obtained for each architecture by simple experiments.

### *3.5 Experimental Results*

Table 1 shows how the execution time is spent on an Origin2000, an SP2, and a T3E during the refinement and subsequent load balancing phases of PLUM when processing Workload-1 (described in Section 2.4.1). The processor reassignment times are not presented since they are comparatively negligible. These timings are for the case when data is remapped before the actual mesh refinement. The advantages of redistributing data in this predictive fashion

Table 1
Execution times (in secs) on the Origin2000, SP2, and T3E using PLUM with Workload-1

| | Refinement Time | | | Remapping Time | | | Partitioning Time | | |
|---|---|---|---|---|---|---|---|---|---|
| P | O2000 | SP2 | T3E | O2000 | SP2 | T3E | O2000 | SP2 | T3E |
| 2 | 9.422 | 21.59 | 6.304 | 3.675 | 3.679 | 1.678 | 0.628 | 0.797 | 0.685 |
| 4 | 4.736 | 10.98 | 3.175 | 2.786 | 3.000 | 1.266 | 0.584 | 0.554 | 0.475 |
| 8 | 2.492 | 5.682 | 1.674 | 2.606 | 3.351 | 1.303 | 0.522 | 0.469 | 0.346 |
| 16 | 1.296 | 2.903 | 0.868 | 2.353 | 2.049 | 0.844 | 0.396 | 0.384 | 0.297 |
| 32 | 0.651 | 1.490 | 0.463 | 0.435 | 1.260 | 0.595 | 0.389 | 0.452 | 0.303 |
| 64 | | 0.794 | 0.254 | | 1.031 | 0.978 | | 0.605 | 0.429 |
| 128 | | | 0.170 | | | 2.565 | | | 0.433 |

is documented in [26]. The refinement speedups are almost identical on all three machines, and show an efficiency of almost 90% on 32 processors. However, note that the T3E refinement times are significantly faster (as are its remapping and partitioning times).

The remapping times shown in Table 1 closely follow the cost model given in Section 3.4 for all three machines, except in a couple of cases. First, there is a dramatic reduction in remapping time when using all 32 processors on the Origin2000. Second, for $P \geq 64$ on the T3E, the remapping time increases with increasing number of processors. Further investigations are required to explain and/or resolve these anomalies.

Another interesting observation is that when using up to 16 processors, the remapping times on the SP2 and the Origin2000 are comparable. Since remapping within PLUM involves both communication and computation, and the results indicate that computation is faster on the Origin2000, it is reasonable to infer that bulk communication is faster on the SP2.

Table 1 also presents the ParMETIS partitioning times on all three systems. Note that there is some dependence on the number of processors used. When there are too few processors, repartitioning takes more time because each processor has a bigger share of the total work. When there are too many processors, an increase in the communication cost slows down the repartitioner.

## 4 Topology-Independent SBN-Based Load Balancer

Our second load balancer is based on a symmetric broadcast network (SBN), which takes into account the global view of system loads among the processors. The SBN is a robust, topology-independent communication pattern (logical or physical) among the processors of a multicomputer system [8]. Before utilizing it to load balance adaptive grids, let us give a brief overview of SBN.

### *4.1 Symmetric Broadcast Networks*

An SBN of dimension $d \geq 0$, denoted as SBN($d$), is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage. It is constructed recursively as follows:

- A single processor forms the basis network SBN(0) consisting of a single stage, denoted as stage 0.
- For $d > 0$, an SBN($d$) is obtained from a pair of SBN($d-1$)s by adding a communication stage in the front with the following interprocessor connections: (i) Processor $i$ in stage 0 is connected to processor $j = (i + P/2) \bmod P$ of stage 1, and (ii) Processor $j$ in stage 1 is connected to the processor in stage 2 which was the stage 0 successor of processor $i$ in SBN($d-1$).

Figure 5 depicts an SBN(2), recursively constructed from two SBN(1)s. Note that SBN($d$) defines unique communication patterns (or broadcast trees) among the processors in the network. Precisely, for any source processor $p$ at stage 0, where $0 \leq p < P$, there exists a unique broadcast tree $T_p$ of height $d = \log P$ such that each of the $2^d$ processors appears exactly once.

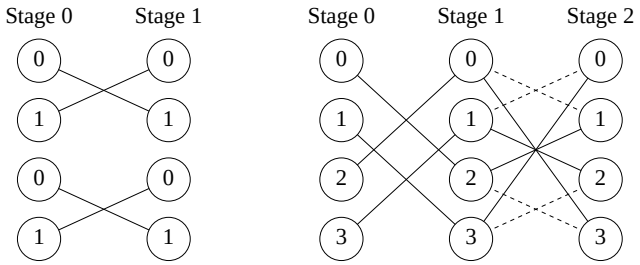


Fig. 5. Construction of SBN(2) from a pair of SBN(1)s. The new connections are shown by solid lines and the original connections by dashed lines.

Furthermore, the SBN communication pattern for any source processor $p$ can be derived from the template tree with processor 0 as the source. For example, let $N_p^s$ be a processor at stage $s$ in the broadcast tree $T_p$. Then $N_p^s = N_0^s \oplus p$, where $\oplus$ is the exclusive-OR operator, thus leading to $T_p = T_0 \oplus p$. The predecessor and successors of each processor are also uniquely defined by specifying

the source and the communication stage. As shown in [32], the versatility of the SBN lies in its efficient embeddings into other topologies such as hypercubes and meshes.

### *4.2 Load Balancing*

The proposed SBN-based load balancer can be classified as adaptive, decentralized, and global, making it effective for adaptive grid applications. The SBN load balancer processes two types of messages: load balance messages and work distribution messages.

A load balance message is broadcast when a processor $p$ determines that its weighted queue length $\texttt{QWgt}(p)$ is less than a minimum threshold, `MinTh`. A load balance message is also broadcast if $\texttt{QWgt}(p)$ is greater than a maximum threshold, `MaxTh`, or if distributing the excess workload will result in other processors exceeding `MaxTh`. As the load balance message is propagated through the SBN, the global weighted queue length (`GWLen`) and the weighted system load level (`WSysLL`) are computed.

Work distribution messages are used to distribute work when $\texttt{QWgt}(p)$ is greater than `MaxTh`. After `WSysLL` is calculated, a distribution message is broadcast through the SBN in order to route work to the lightly-loaded processors and to update the threshold values (`MinTh` and `MaxTh`). As a result, all the processor workloads are balanced.

Let us now discuss the various parameters and other details involved in the SBN-based load balancer implementation, a preliminary version of which appeared in [33].

- Weighted Queue Length:
  The queue length (computation time) of a processor $p$ is not an accurate estimate of the total time required to complete its work, particularly in applications where the grid is adapted. We therefore define a metric called weighted queue length, $\texttt{QWgt}(p)$, that also considers the communication and redistribution costs. If $W_v$ is the computational cost to process a dual graph vertex $v$, $C_v^p$ is the communication cost to interact with the vertices adjacent to $v$ but whose data sets are not local to $p$, and $R_v^p$ is the redistribution cost to copy the data set for $v$ to $p$ from another processor (see Section 3.1), then

$$\texttt{QWgt}(p) = \sum_{v \text{ assigned to } p} (W_v + C_v^p + R_v^p).$$

  Note that if the data set for $v$ is already assigned to $p$, no redistribution cost

is incurred ($R_v^p = 0$). Similarly, if the data sets of all the vertices adjacent to $v$ are also already assigned to $p$, the communication cost, $C_v^p = 0$.

- Weighted System Load:
  The weighted system load is defined as:

$$\texttt{WSysLL} = \left\lceil \frac{1}{P} \sum_{p=1}^{P} \texttt{QWgt}(p) \right\rceil ,$$

  where $P$ is the total number of processors used. Assuming that the workload is perfectly balanced among the processors, `WSysLL` estimates the time required to process the grid and reflects the computation, communication, and redistribution costs in the current grid-to-processor assignment. Hence, a global view of the system is captured.
- Prioritized Vertex Selection:
  When selecting dual graph vertices to process, the SBN load balancer utilizes the underlying grid structure to defer execution of boundary vertices as long as possible because they may be migrated for more efficient execution. Thus, the vertex to be processed next is selected such that it minimizes the overall edge cut of the adapted grid. To enable this, a priority min-queue is maintained, where the priority of vertex $v$ in processor $p$ is given by $(C_v^p + R_v^p)/W_v$. Therefore, vertices with no communication and redistribution costs are executed first, while those with high communication or redistribution overhead relative to their computational weight are processed last. Basically, internal vertices are processed before those on partition boundaries.
- Differential Edge Cut:
  To balance processor workloads, an optimal policy for vertex migration is required. Assume that processor $p$ needs to reassign some of its vertices to another processor $q$. The SBN load balancer running on $p$ calculates the differential edge cut [2], $\Delta\texttt{Cut}(v)$, for each vertex $v$ that is queued locally as follows:

$$\Delta\texttt{Cut}(v) = R_v^q - R_v^p + C_v^q - C_v^p.$$

  A negative value indicates a reduction in communication and redistribution costs if $v$ is migrated from $p$ to $q$, hence favoring the migration of vertices with the largest absolute reduction in these costs. The vertex $w$ with the smallest value of $\Delta\texttt{Cut}(w)$ is chosen for migration. The vertices adjacent to $w$ that are queued for processing on $p$ are also targeted. The entire process is repeated if more work needs to be transferred from $p$ to $q$. This migration policy therefore strives to maintain or improve the cut size during the execution of the load balancing algorithm.

[2] Here we are deviating from the usual definition of edge cut to account for the dynamic nature of the SBN-based load balancer.

- Data Redistribution Policy:
  Data is redistributed in a lazy manner, i.e. the data set for a vertex $v$ in processor $p$ is not moved to $q$ until $q$ is about to execute $v$. Moreover, the data sets of all vertices adjacent to $v$ that are assigned to $q$ are transferred as well. This policy significantly reduces the redistribution and communication costs by avoiding multiple migration of data sets and having resident on $q$ all adjacent vertices of $v$ while $q$ processes $v$. In other words, the communication overhead is reduced by considering the underlying grid structure.

### *4.3 Differences with PLUM*

The SBN-based load balancer differs from PLUM in several respects that make it difficult to directly compare the two strategies. These differences are itemized below:

- Load balancing under PLUM requires the temporary suspension of useful processing in order to generate new partitions and redistribute the data. Instead, the SBN-based approach allows processing to continue while the load is dynamically balanced. This feature makes it possible to use latency-tolerant techniques to hide the communication and redistribution costs during processing.
- Under PLUM, the suspension of processing and subsequent repartitioning does not guarantee an improvement in load balance quality. This is because the adapted grid is not redistributed if the remapping cost exceeds the benefits of load balancing. In contrast, the SBN approach will always improve the load balance among the processors.
- PLUM redistributes all necessary data to the appropriate processors before the computations are resumed. SBN, however, reduces redistribution and communication costs by migrating remote data to a processor only when it is ready to process the data.

### *4.4 Performance Metrics*

We chose the following metrics to evaluate the effectiveness of the SBN-based load balancer when processing Workload-2, described in Section 2.4.2. Recall that $v$ denotes a vertex to be processed and $P$ is the total number of processors.

- Load Imbalance Factor:
  This metric is formulated as:

$$\texttt{LoadImb} = \max_{p \in P} \texttt{QWgt}(p) \;/\; \texttt{WSysLL},$$

and should be as close to unity as possible.

- Cut Percentage:
  The runtime interaction between adjacent vertices residing on different processors is represented by this metric as:

$$\texttt{Cut\%} = 100 \times \sum_{p \in P} \sum_{v \text{ assigned to } p} C_v^p \Big/ \sum_{e \text{ in mesh}} C_e \ ,$$

  where $C_e$ is the weight of edge $e$ in the adaptive grid. The value of `Cut%` should be as small as possible.
- Maximum Redistribution Cost:
  Since a processor can either be sending or receiving data, the overhead for the two phases is modeled as a sum of two costs in this metric:

$$\texttt{MaxSR} = \max_{p \in P} \Big\{ \sum_{v \text{ sent from } p} R_v^p \Big\} + \max_{p \in P} \Big\{ \sum_{v \text{ recv by } p} R_v^p \Big\}.$$

  Recall from Section 3.1 that minimizing `MaxSR` guarantees a reduction in the total data redistribution overhead.

### *4.5 Experimental Results*

The SBN-based load balancing algorithm was implemented using MPI on an SP2 and an Origin2000, and tested with Workload-2. Performance results, averaged over nine levels of adaptation, are presented in Table 2.

In addition to achieving excellent load balance (`LoadImb`), the redistribution cost (`MaxSR`) is significantly reduced from that obtained with PLUM [33]. The `MaxSR` values are larger on the Origin2000 compared to the SP2 when using less than 16 processors because of a minor change that was made to the SBN load balancer. Basically, the algorithm for distributing jobs was modified to

Table 2
Performance results on the SP2 and Origin2000 using the SBN-based load balancer with Workload-2

| | `LoadImb` | | `Cut%` | | `MaxSR` | |
|---|---|---|---|---|---|---|
| P | SP2 | O2000 | SP2 | O2000 | SP2 | O2000 |
| 2 | 1.00 | 1.01 | 5.51 | 7.65 | 80,037 | 158,085 |
| 4 | 1.00 | 1.00 | 10.76 | 12.81 | 76,665 | 114,652 |
| 8 | 1.00 | 1.01 | 16.35 | 18.33 | 53,745 | 65,824 |
| 16 | 1.01 | 1.01 | 23.87 | 29.25 | 46,825 | 35,389 |
| 32 | 1.02 | 1.02 | 30.41 | 36.47 | 28,031 | 19,446 |

guarantee that the processor initiating load balancing was always supplied a sufficient workload. This strategy reduces the number of load balance messages but increases the number of vertices migrated, especially for small numbers of processors.

Unfortunately, the edge cut percentages (`Cut%`) are greater than those for PLUM [33], indicating that the SBN strategy reduces the redistribution cost at the expense of a higher communication cost. This is due to the lazy approach of data migration used by the SBN-based load balancer, but is an acceptable trade-off for applications where the data migration cost is dominant.

Table 3 shows the communication overhead of the SBN-based approach in terms of the number of bytes that was transferred among processors during the load balancing and work distribution phases. Note that the number of bytes exchanged in order to balance the system load is comparable on the two machines, but that the number of bytes required for workload distribution is significantly less on the Origin2000. This is due to the bulk migration strategy that was implemented in the SBN load balancer for the Origin2000 experiments. In any case, these results demonstrate that the cost of balancing the system load is negligible compared to the cost of workload migration.

Table 3
Communication overhead (in Mbytes) of the SBN-based load balancer

| | Load Balancing Phase | | Work Distribution Phase | |
|---|---|---|---|---|
| P | SP2 | O2000 | SP2 | O2000 |
| 2 | 0.342 | 2.518 | 3.919 | 0.403 |
| 4 | 0.150 | 0.674 | 7.939 | 0.576 |
| 8 | 0.463 | 0.586 | 25.397 | 0.770 |
| 16 | 0.581 | 0.793 | 30.454 | 0.865 |
| 32 | 1.550 | 1.197 | 38.244 | 1.071 |

Table 4 shows the fraction of time spent in the SBN load balancer compared to the total time required to process the mesh adaptation application. Each of the three columns corresponds to a certain type of load balancing activity: (i) the time needed to handle balance-related messages, (ii) the time needed to migrate messages from one processor to another, and (iii) the time needed to select the next dual graph vertex to be processed. Results show that the overhead related to vertex selection is the most expensive phase of the SBN load balancer. However, the total load balancing time on both machines is still relatively small compared to the time spent adapting the mesh.

In conclusion, these experimental results demonstrate that the proposed SBN-based dynamic load balancer is effective in processing adaptive grid applica-

Table 4
Percentage overhead of the SBN-based load balancer

| | Balancing Activity | | Message Migration | | Vertex Selection | |
|---|---|---|---|---|---|---|
| P | SP2 | O2000 | SP2 | O2000 | SP2 | O2000 |
| 2 | 0.0053 | 0.0575 | 0.0014 | 0.0326 | 0.4530 | 0.6908 |
| 4 | 0.0087 | 0.0025 | 0.0069 | 0.0428 | 2.0381 | 0.4839 |
| 8 | 0.1745 | 0.0077 | 0.0569 | 0.0420 | 2.8386 | 0.4713 |
| 16 | 0.2669 | 0.0134 | 0.0629 | 0.0988 | 0.8845 | 0.2790 |
| 32 | 0.1154 | 0.0162 | 0.0774 | 0.0531 | 2.1043 | 0.1054 |

tions. In fact, the SBN balancer would be preferred in situations where the data redistribution overhead dominates the runtime interprocessor communication cost.

## 5 Conclusions

We have demonstrated the applicability of our novel approaches (PLUM and SBN-based) to solving the load balancing problem for adaptive unstructured grids on three state-of-the-art commercial machines (SP2, T3E, and Origin2000). The experiments were conducted using actual workloads of steady and unsteady grids obtained from real-life applications. Results indicated that both PLUM and the SBN approach achieve excellent load balance at the cost of minimal extra overhead.

We are currently examining the portability of our software on other platforms, such as shared-memory environments and workstation clusters. We also believe that the performance of our load balancers can be improved by exploiting one-sided communication. This will be the subject of future research.

## Acknowledgements

The work of the first author was supported by NASA under contract number NAS 2-14303 with MRJ Technology Solutions. The work of the second and third authors was supported by Texas Advanced Research Program grant number TARP-97-003594-013. The work of the fourth author was supported by NASA under contract number NAS 2-96027 with Universities Space Research Association while he was with the Research Institute for Advanced Computer Science (RIACS).

## References


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

[2] Das, S.K., Harvey, D.J., Biswas, R., Adaptive load balancing algorithms using symmetric broadcast networks: Performance study on an IBM SP2. Proc. 26th Intl. Conf. on Parallel Processing, Bloomingdale, IL (1997) 360–367.

[3] Ghosh, B., Muthukrishnan, S., Dynamic load balancing in parallel and distributed networks by random matchings. Proc. 6th ACM Symp. on Parallel Algorithms and Architectures, Cape May, NJ (1994) 226–235.

[4] Horton, G., A multi-level diffusion method for dynamic load balancing. Parallel Comput. 19 (1993) 209–229.

[5] Kohring, G.A., Dynamic load balancing for parallelized particle simulations on MIMD computers. Parallel Comput. 21(1995) 683–693.

[6] Mahapatra, N., Dutt, S., Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing. Proc. 10th Intl. Parallel Processing Symp., Honolulu, HI (1996) 881–885.

[7] Oliker, L., Biswas, R., PLUM: Parallel load balancing for adaptive unstructured meshes. J. Parallel Distrib. Comput. 52 (1998) 150–177.

[8] Das, S.K., Prasad, S.K., Implementing task ready queues in a multiprocessing environment. Proc. Intl. Conf. on Parallel Computing, Pune, India (1990) 132–140.

[9] Eager, D.L., Lazowska, E.D., Zahorjan, J., A comparison of receiver-initiated and sender-initiated adaptive load sharing. Performance Evaluation 6 (1986) 53–68.

[10] Eager, D.L., Lazowska, E.D., Zahorjan, J., Adaptive load sharing in homogeneous distributed systems. IEEE Trans. on Soft. Engrg. 12 (1986) 662–675.

[11] Lin, F.C.H., Keller, R.M., The gradient model load balancing method. IEEE Trans. on Soft. Engrg. 13 (1987) 32–38.

[12] Shu, W., Wu, M.-Y., Runtime incremental parallel scheduling (RIPS) on distributed memory computers. IEEE Trans. on Parallel Distrib. Sys. 7 (1996) 637–649.

[13] http://www.nas.nasa.gov/IPG/

[14] http://www.globus.org/

[15] Kirkpatrick, S., Gelatt Jr., C.D., Vechhi, M.P., Optimization by simulated annealing. Science 220 (1983) 671–680.

[16] Khuri, S., Baterekh, A., Genetic algorithms and discrete optimization. Methods of Operations Research 64 (1991) 133–142.

[17] Farhat, C., A simple and efficient automatic FEM domain decomposer. Computers and Structures 28 (1988) 579–602.

[18] Chan, W., George, A., A linear time implementation of the reverse Cuthill-McKee algorithm. BIT 20 (1980) 8–14.

[19] Simon, H.D., Partitioning of unstructured problems for parallel processing. Comput. Sys. in Engrg. 2 (1991) 135–148.

[20] Nour-Omid, B., Raefsky, A., Lyzenga, G.A., Solving finite element equations on concurrent processors. Proc. Symp. on Parallel Computations and their Impact on Mechanics, Boston, MA (1987) 209–218.

[21] Hendrickson, B., Leland, R., Multidimensional spectral load balancing. Technical Report SAND93-0074, Sandia Natl. Labs., Albuquerque, NM (1993).

[22] Kernighan, B.W., Lin, S., An efficient heuristic procedure for partitioning graphs. Bell Systems Technical Journal 49 (1970) 291–307.

[23] Hendrickson, B., Leland, R., A multilevel algorithm for partitioning graphs. Proc. Supercomputing '95, San Diego, CA (1995).

[24] Karypis, G., Kumar, V., Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report 96-036, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, MN (1996).

[25] Walshaw, C., Cross, M., Everett, M.G., Parallel dynamic graph partitioning for adaptive unstructured meshes. J. Parallel Distrib. Comput. 47 (1997) 102–108.

[26] Biswas, R., Oliker, L., Experiments with repartitioning and load balancing adaptive meshes. Grid Generation and Adaptive Algorithms, IMA Volumes in Mathematics and its Applications 113 (1999) 89–111.

[27] Schloegel, K., Karypis, G., Kumar, V., Biswas, R., Oliker, L., A performance study of diffusive vs. remapped load-balancing schemes. Proc. 11th Intl. Conf. on Parallel and Distributed Computing Systems, Chicago, IL (1998) 59–66.

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

[29] Flaherty, J.E., Loy, R.M., Ozturan, C., Shephard, M.S., Szymanski, B.K., Teresco, J.D., Ziantz, L.H., Parallel structures and dynamic load balancing for adaptive finite element computation. Appl. Numer. Math. 26 (1998) 241–263.

[30] Strawn, R.C., Biswas, R., Garceau, M., Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise. J. Aircraft 32 (1995) 754–760.

[31] Oliker, L., Biswas, R., Gabow, H.N., Parallel tetrahedral mesh adaptation with dynamic load balancing. Parallel Comput. (2000) to appear.

[32] Das, S.K., Harvey, D.J., Performance analysis of an adaptive symmetric broadcast load balancing algorithm on the hypercube. Technical Report CRPDC-95-1, Dept. of Computer Science, Univ. of North Texas, Denton, TX (1995).

[33] Das, S.K., Harvey, D.J., Biswas, R., Parallel processing of adaptive meshes with load balancing. Proc. 27th Intl. Conf. on Parallel Processing, Minneapolis, MN (1998) 502–509.