A Hierarchical Partition Model for Adaptive Finite Element Computation

J. D. Teresco, M. W. Beall, J. E. Flaherty and M. S. Shephard

Scientific Computation Research Center (SCOREC)
and Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180

Software tools for the solution of partial differential equations using parallel adaptive finite element methods have been developed. We describe the design and implementation of the parallel mesh structures within an adaptive framework. The most fundamental concept is that of a hierarchical partition model used to distribute finite element meshes and associated data on a parallel computer. The hierarchical model represents heterogeneous processor and network speeds, and may be used to represent processes in any parallel computing environment, including an SMP, a distributed-memory computer, a network of workstations, or some combination of these. Using this model to segment the computation into chunks which can fit into cache memory provides a potential efficiency gain from an increased cache hit rate, even in a single processor environment. The information about different processor speeds, memory sizes, and the corresponding interconnection network can be useful in a dynamic load balancing algorithm which seeks to achieve a good balance with minimal interprocessor communication penalties when a slow interconnection network is involved.

Preprint submitted to Elsevier Preprint *29 May 1998*

1 Introduction

The finite element method (FEM) has become a standard analysis tool for solving partial differential equations (PDEs). Computationally demanding three-dimensional problems make adaptive methods and parallel computation essential. Adaptive FEMs provide reliability, robustness, and time and space efficiency. In such a method, the computational domain is discretized into a mesh. During the adaptive solution process, portions of the mesh may be refined or coarsened (h-refinement) or moved to follow evolving phenomena (r-refinement). The method order may also be varied (p-refinement). Each adaptive process concentrates the computational effort in areas where the solution resolution would otherwise be inadequate [7]. Conventional array-based data representations, which work well for fixed-mesh solutions, are not well-suited to solutions involving mesh adaptivity [1]. Traversal of the data must be efficient in all cases, but when adaptivity is introduced, modification of the mesh structures and corresponding solution data must also be efficient.

Parallel computation introduces complications such as the need to balance processor loading, coordinate interprocessor communication, and manage distributed data. The standard methodology for optimizing parallel FEM programs relies on a static partitioning of the mesh across the cooperating processors. This is insufficient in an adaptive computation, since load imbalance will be introduced by adaptive enrichment, necessitating dynamic partitioning and redistribution of data. Data structures must support this dynamic mesh migration.

Reusable software libraries are essential for the development of specific applications to solve diverse problems without concern for the details of the underlying mesh structures or parallelization. A variety of such tools have been developed to facilitate the design, implementation, and use of parallel adaptive finite element software. Static data structures cannot be used with adaptive computations, which makes automatic detection by a parallelizing compiler difficult. Parallelism is generally explicit, achieved through the use of a message passing library such as the Message Passing Interface (MPI) [18], and requires a partitioning algorithm to distribute data among processing nodes. Parallel adaptive FEM computations can be distributed in a natural way by a domain decomposition of the mesh underlying the computation, but being adaptive, these meshes will change and the system must account for this.

The approaches can be categorized by a low-level view of the data structures. Two basic paradigms have been used as the underlying structure for these computations. An array-based approach, such as the Scalable Distributed Dynamic Array [11], uses the distribution of arrays as the fundamental unit of parallelism. The mesh-based approach [1,4,14,21,24–27] uses mesh connectivity and distribution of mesh entities to achieve parallelism.

Our recent efforts have focused on implementing a framework for the reliable automated solution of problems in science and engineering over arbitrary domains using scalable parallel adaptive techniques [2]. This framework provides common interfaces to various pieces of software typically needed for scientific computation, and in particular when writing finite element analysis procedures. Central to this framework is a distributed mesh structure, described in Section 3.

While parallel adaptive methods have been used successfully in the solutions of many problems [3,5,8,9,11,14,20], it is difficult to take full advantage of a particular parallel computing environment. A partitioning and dynamic load balancing strategy which is efficient in a shared-memory environment may not be in a distributed-memory environment. This is especially the case with heterogeneous or hierarchical computers which combine multiprocessing shared memory nodes connected by networks of various speeds. A partitioner must have some knowledge of the parallel environment to achieve a good distribution. In Section 3, we describe a hierarchical partition model which makes such information available.

2 Target Parallel Environments

The large range of problem sizes to be addressed and user computing environments available requires software that runs efficiently on computers ranging from single workstations to the largest parallel supercomputers. In order for a parallel program to be efficient on a specific computer, it must obtain and use knowledge of the parallel computational environment on which it is running. Information that is useful in this respect includes available memory, available processing power, and availability and properties (bandwidth and latency) of communication resources. MPI [18] does not provide such information (nor does the MPI-2 specification [19]). Some of this information can be computed through environment queries, but other information will need to be provided manually.

Consider five specific parallel computing environments, represented in Figure 1. The first (Figure 1a) is an eight-node computer with all uniprocessor nodes connected by a fast, nonblocking switch. Next (Figure 1b), is an eight-processor symmetric multiprocessing (SMP) workstation. The third (Figure 1c), is a collection of eight workstations connected by Ethernet, in a typical network of workstations (NOW) configuration. Figure 1d shows a hybrid system, a six-node computer which has four single-processor nodes and two four-way SMP nodes, all connected by a high-speed switch. And finally, Figure 1e shows a more complex networking scheme, with two clusters of Asynchronous Transfer Mode-connected (ATM) four-way SMP workstations, where the two clusters are connected by Ethernet.

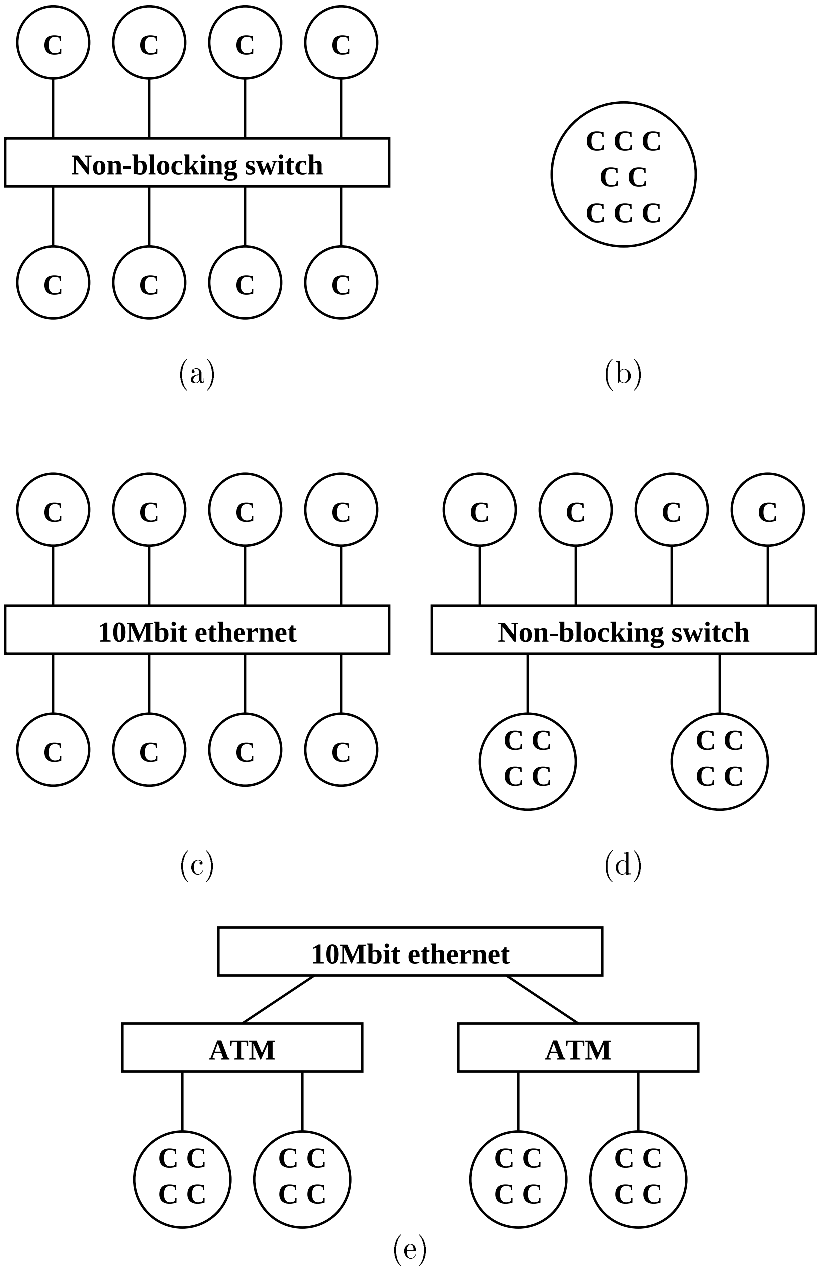


Fig. 1. Examples of parallel computing environments. A "C" represents a CPU. Multiple C's in a process node represent an SMP node. (a) An 8-node computer with all uniprocessor nodes, connected by a fast switch. (b) An 8-way SMP workstation. (c) Eight uniprocessor workstations connected by Ethernet. (d) An 8-node system with 6 uniprocessor nodes and 2 4-processor SMP nodes connected by a fast switch. (e) Two clusters of ATM connected 4-processor SMP nodes connected by Ethernet.

Consider a parallel program that has been developed and optimized for the environment of Figure 1a, which is characterized by a distributed-memory environment with communication provided by a high-speed, nonblocking switch. The computational domain is distributed among the processors by a partitioning algorithm such that the workload is approximately equidistributed and the interfaces between the subdomains are small. Perhaps it has been determined that a small load imbalance is tolerable if it will reduce the size of the interfaces; hence, the amount of interprocessor communication across the switch.

Suppose that the program is subsequently targeted for execution on an SMP workstation (Figure 1b), where the processes communicate via shared memory, which is fast relative to a nonblocking switch. The assumption allowing a small computational imbalance to reduce interprocessor communication may no longer be effective. Since communication is less expensive, the loss of CPU utilization due to the imbalance may significantly degrade performance. A network of eight uniprocessor workstations connected by Ethernet (Figure 1c) is at the opposite extreme. In this case, interprocessor communication is very expensive. A more significant computational imbalance may be acceptable if some communication can be avoided. Since the same program can use both environments, the partitioner needs run-time information to determine the existing situation and to decide the optimal relationship between a balanced computational load and minimal communication.

The situation becomes more complex with heterogeneous environments. In the case of ATM clusters (Figure 1e), there are three levels of interconnection network which may be used for one process to communicate with another. Processes on the same SMP node communicate via shared memory. Processes on different SMP nodes communicate via ATM if they are in the same ATM cluster and via Ethernet if they are in different clusters. Communication over the shared-memory interface is inexpensive, so there is no need for great expense in partitioning data stored on the same SMP node. Communication over the ATM interface is slower, but still fast enough to use an inexpensive partitioning and to maintain a close balance. When slower communication over Ethernet is involved, it is worthwhile to minimize communication over the slow network interface, including use of a more expensive partitioning method and tolerating a larger load imbalance. Heterogeneous processors are also important to consider. Processors within a computation may have different speeds and different amounts of memory available. In cases where computing resources are shared, the full processing power of a given computer may not be available, so a measure of available processing power, depending on CPU demands of other processes, must be maintained.

It is unreasonable to maintain distinct parallel and serial versions of all applications. However, it is also important to eliminate the overhead associated with parallel computation when the program is run on a single processor. Some parallel overhead may be removed at compile time while other overhead is trivial to remove at run time.

3 Parallel Mesh Infrastructure

We are developing the *Rensselaer Partition Model* (RPM), which allows a hierarchical partitioning of finite element mesh data and provides additional information about the parallel computational environment in which a program is executing. The model works with the mesh data structures described here; however, many of the ideas may be applied to other systems.

3.1 Basic Mesh Structures

The *SCOREC Mesh Database* (MDB) [1] provides an object-oriented, hierarchical representation of a finite element mesh. It includes a set of operators to query and update the mesh data structure. The basic mesh entity hierarchy consists of three-dimensional *regions*, and their bounding *faces*, *edges*, and *vertices*, with bidirectional links between mesh entities of consecutive order (Figure 2). In three-dimensional meshes, regions are used as finite elements while faces serve as elements in two dimensions, or as interface elements in three dimensions. Mesh entities have an explicit *geometric classification* relative to a geometric model of the problem domain to allow for the appropriate representation of the geometry as the mesh is enriched either through h- or p-refinement. Mesh classification against the geometric domain can be defined more precisely as the unique association of a topological mesh entity of dimension d_i, to a topological entity of the geometric domain of dimension d_j, where $d_i \leq d_j$. Multiple mesh entities can be classified on a geometric entity. A mesh region is classified on the domain region in which it lies. A mesh face is either classified on the domain region or face in which it lies. A mesh edge is classified on the domain region, face, or edge in which it lies. Finally, a mesh vertex is classified on the domain region, face, edge, or vertex in which it lies. Mesh entities are always classified with respect to the lowest order model entity possible. More formal definitions of classification and adjacency can be found in [1].

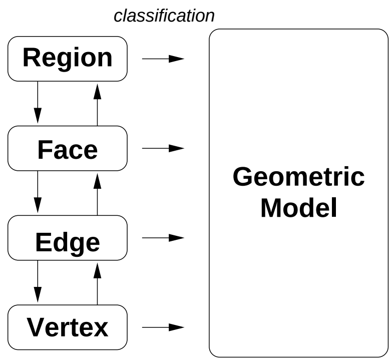


Fig. 2. MDB entity hierarchy with links to a geometric model.

The full entity hierarchy allows the efficient deletion and creation of mesh entities during h-refinement [24] and simplifies attachment of degrees of freedom to the mesh entities and determination of necessary geometric information during p-refinement [23]. The database allows for the fast retrieval of adjacency information. Examples of available data include the list of faces bounding an element and the edges sharing a common vertex. All entities can have attached attributes such as solution and error indicator data.

3.2 Mesh Data Organization

Mesh entities are listed with the geometric model entities on which they are classified (Figure 3). When mesh entities are added to or removed from a mesh, or when mesh entities are traversed, the appropriate lists of regions, faces, edges, and vertices stored on the geometric model entities are used. The appropriate geometric entity is readily available from the mesh entity, so no additional searching is needed. Traversal of the entire mesh requires some additional bookkeeping, since lists of geometric entities and the mesh entity lists within them must be traversed. This additional overhead is not significant for performance and the complications are hidden within the iterator implementation. The main advantage of this design is that the inverse classification information (which mesh entities are classified on a given model entity) is readily available. This is useful, for example, when applying a boundary condition on a model face. Rather than traversing all faces in the mesh and querying each to check if it is on the desired boundary, a list of the needed entities is readily available.

3.3 Partition Model

The mesh data in a parallel computing environment is distributed. Each entity in the finite element mesh is uniquely assigned to a *partition model entity* or, more simply, to a *partition*. The direct assignment of one partition to each process has been a limitation of prior systems [14]. While this is straightfor-

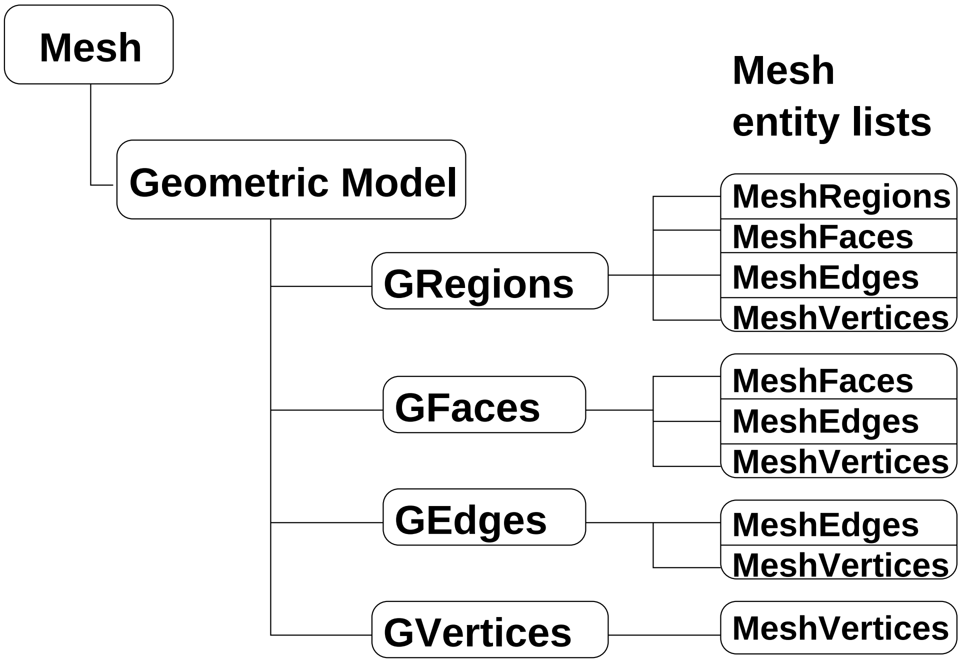


Fig. 3. Data organization with mesh data stored relative to a geometric model. `GRegions`, `GFaces`, `GEdges`, and `GVertices` refer to lists of geometric model regions, faces, edges, and vertices, respectively.

ward, there are advantages in allowing multiple partitions on a single process. Our approach to mesh migration (Section 3.9) requires this. Other motivations include the use of the partitions as a coarser structure for a load balancer, as a method to encourage locality of reference to enhance cache performance, and as a means of performing out-of-core computations by swapping partitions in and out of memory. Each of these will be discussed further in Section 6. Each partition is assigned to a specific *process*. Multiple partition model entities may be assigned to a single process. "Process" in this context refers to an address space. The model is hierarchical, with partition model entities assigned to a *process model* and process model entities assigned to a *machine model*.

The machine model represents the physical processing nodes involved in the computation, and information about the communication methods to other involved computers. The process model maps processes to the computer, and maps interprocess communication to inter-computer networks or, perhaps, to a shared-memory interface. The mesh is represented at the partition model level. Partitions must know the mesh entities that they contain, and a mesh entity must know its partition assignment, which we refer to as its *partition model classification.*

3.4 Augmented Model

The simplest implementation of the partition model introduces pointers on each mesh entity to store (*i*) partition classification and (*ii*) geometric classification. For space efficiency, these may be combined into a single topological unit called the *augmented model.* Additional topological entities are added to the model which correspond to partitions and partition boundaries, forming a toplogical structure which contains geometric and partition information. Consider the simple geometry and corresponding mesh of Figure 4. The geometric model contains 2 faces, 8 edges, and 7 vertices. Mesh entities are classified onto these geometric entities as shown on the right.

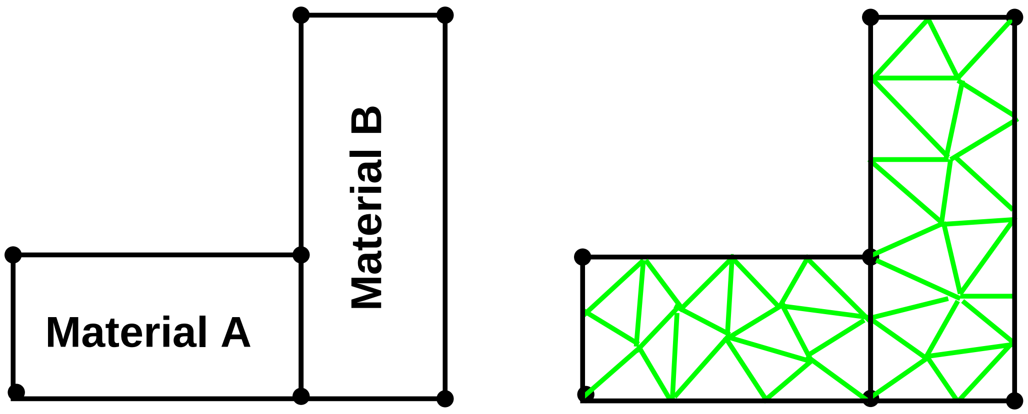


Fig. 4. Example domain geometry (left) and mesh classified on that domain (right). The distinct geometric faces could represent two different materials, for example.

In Figure 5, we divide the mesh into 5 partitions (shown as solid lines). The partitioning is done independently of the geometric model, but the partitioning induces a similar topological structure (Figure 5, right) When this information is combined with the geometry of Figure 4, left, the augmented model (Figure 6) is obtained. The augmented topology has 7 faces, 19 edges, and 15 vertices.

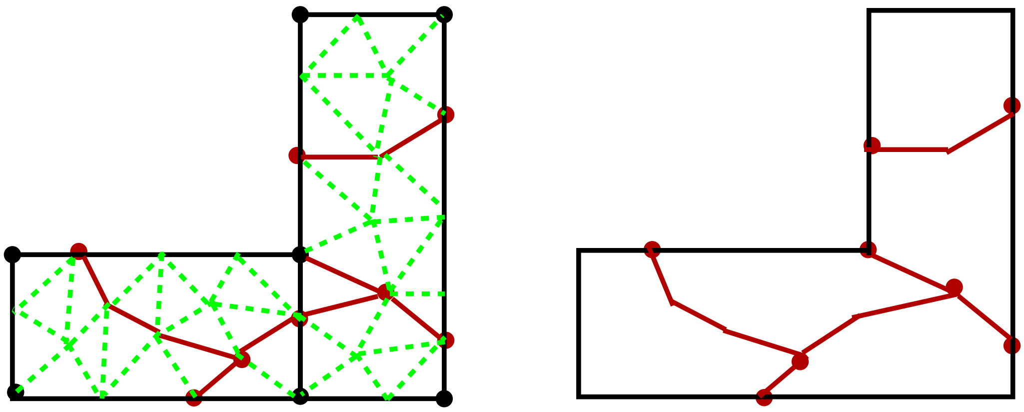


Fig. 5. Mesh partitioning into 5 parts (left) and the associated partition model (right).

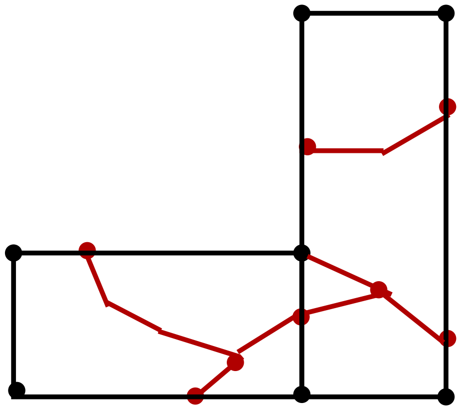


Fig. 6. Topological structure resulting from augmented model combining the geometric and partition model.

The size of the augmented structure will grow with the number of partitions. Such a structure would need to be distributed. The implementation of the augmented model becomes complex, because the CAD packages underlying the geometric model do not support distribution. However, the abstraction is still useful, and the same functionality is achieved by the data organization described in the next section. This can be done because the topological model represents an independent abstraction of the domain, in which the definition of the basic entities can be based on any desired criteria as long as the rules on entity adjacencies are maintained.

Partitions are represented by assigning separate "mesh" pointers to each, as shown in Figure 7. This is equivalent to the augmented model in functionality, but does not require the overhead of the more complex distributed augmented model structure. In this case, the partition model is only a topological construct. It contains no geometric information and is distinct from the geometric model. The mesh entity lists within each geometric model entity are extended to contain one list for each partition within the local process, but avoid the second per-mesh entity pointer for partition classification. There is a small bit field on each entity which is used to retrieve the mesh pointer to which a mesh entity belongs, and this is used to determine partition classification, since each partition is stored as a separate mesh.

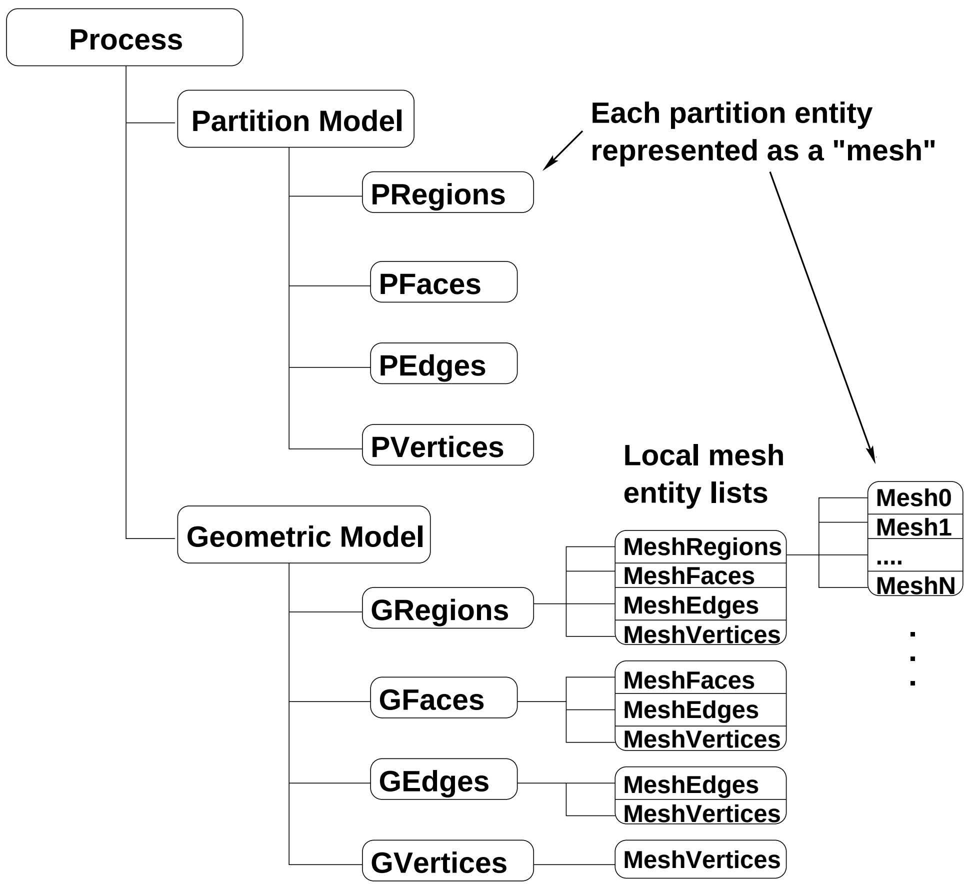


Fig. 7. Data organization with both geometric and partition models

3.6 Process and Machine Models

Partition model entities are assigned to a process model (Figure 8). Each process, with a unique address space in a parallel computation, is assigned a process model entity. The process model represents the "live" state of the job and is concerned with its computational and communication performance. Each process tracks the work assigned to it and, at any given time, the amount of this work that has been completed. The actual communication performance, between this process and other processes to which it communicates, is tracked within the *Process Interconnect* structures associated with the active connection between process objects.

In contrast, the machine model represents the theoretical performance that the hardware environment is capable of delivering while the process model represents the actual performance that is being achieved. Process model entities are assigned to a machine model (Figure 9).

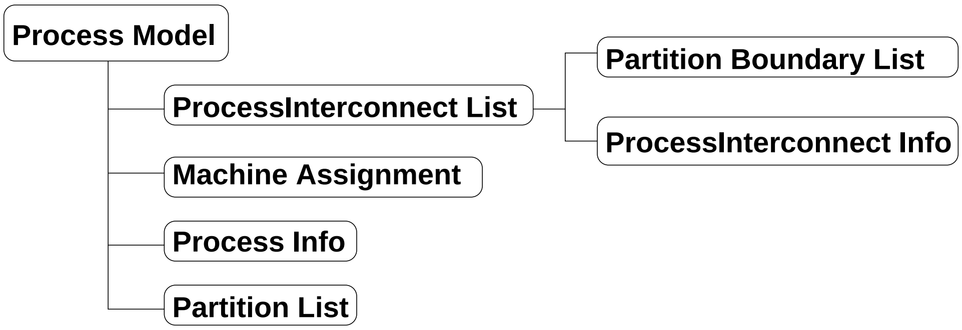


Fig. 8. Process model data organization.

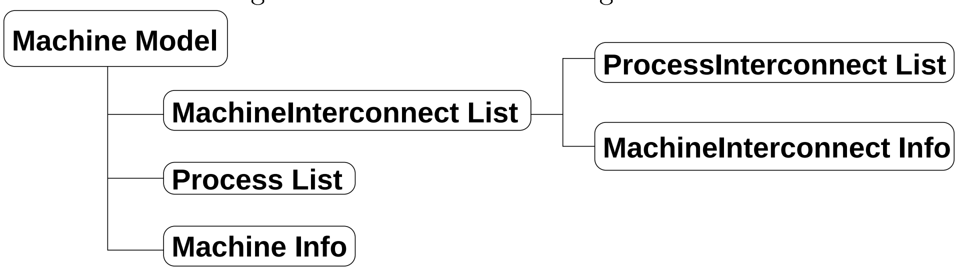


Fig. 9. Machine model data organization.

The machine model describes the computational environment in which the process is running. This information includes available memory, available processing power, available communication resources and the properties of these resources. In addition, the machine model provides capabilities to manage and modify the resources used by the computation. The model is a graph that describes the various hierarchies in the computational environment. A machine model entity represents a processing node, which may have multiple processing units. Multiple processes can be assigned to a single computer when either a computer includes SMP nodes or multiple processes are run on a single CPU. The latter is often desirable when debugging, such as when debugging a test case with a large number of processes on a smaller number of nodes. Multiple processes on a computer are not specifically managed by RPM. They are under the control of the operating system scheduler. *MachineInterconnect* entities can be used to model different network speeds.

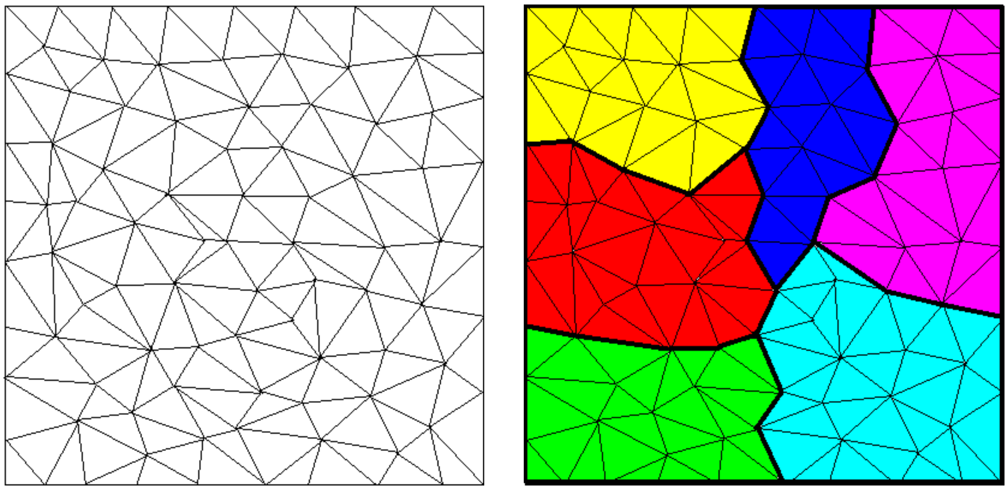


Fig. 10. A sample two-dimensional mesh (left) and its partitioning (right)

In Figure 10, we see a sample two-dimensional mesh (left) and its classification onto the partition model (right). Mesh entities are classified on a partition model entity, which, in this case, is a partition face or boundary. Partition model entities are assigned to process model entities (Figure 11, left), allowing multiple partitions to be assigned to a single process. And the process model entities are associated with a machine model (Figure 11, right), allowing multiple processes to be assigned to a single machine "entity."

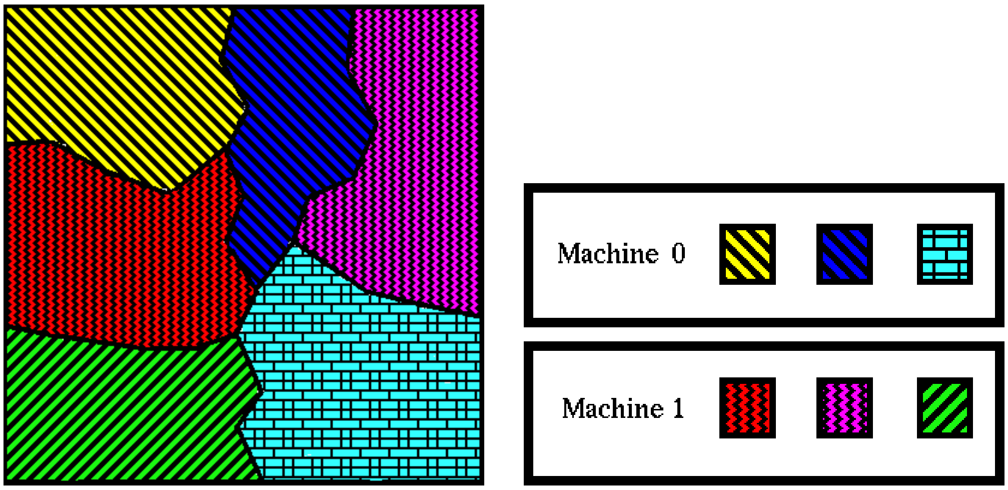


Fig. 11. Assignment of partitions to a process model (left), and partition and process assignments on a machine model (right). Patterns indicate assignments on the process model.

3.7 Boundary Structures

Mesh entities need only be replicated when on a partition boundary that is also a process boundary. Figure 12 shows two-dimensional examples of mesh faces which share a common edge across a partition boundary. The shared mesh edge is classified on the partition edge in each case. On the left, the partition edge is local to the process, so the mesh entity need not be replicated and is stored only on the "mesh" associated with the partition edge. On the right, the partition edge is also on the process boundary. In this case, the partition boundary is replicated in each process. The mesh edge classified on the partition boundary must also be replicated. It is still classified on the partition edge.

Consider a mesh adjacency operator that retrieves the list of faces which bound a given region. One or more of the faces returned could be on a partition boundary and may be replicated. The operator must return a pointer to a local entity. There is a unique instance of the duplicated face which is the actual entity (the one located on the *owner* of the entity). The copies returned

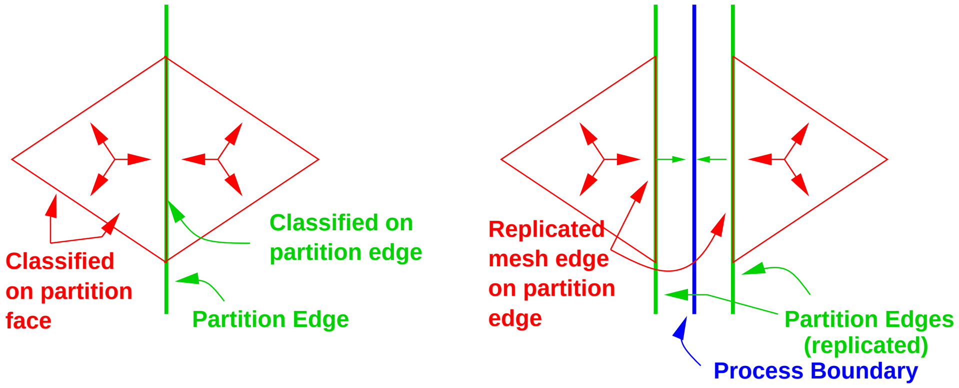


Fig. 12. Partition classification of mesh entities for the case of same-process partition (left) and at the partition boundary which is also a process boundary (right)

by non-owners contain appropriate information for the process to obtain the “real” mesh entity, which in this case would be located within the address space of another process.

3.8 Mesh Entity Counts and Traversals

A parallel adaptive FEM code must query and traverse a mesh in a number of different ways. A solution procedure will need to traverse all entities where solution data can be stored. This may be restricted to entities associated with a particular portion of the geometry. A linear algebra solver will need to traverse the interiors of partitions to solve local problems and partition boundaries to solve interface problems. A partitioning algorithm will need to query the number of mesh entities in each partition and on partition boundaries.

Given these possibilities, the question “how many mesh faces” can have different meanings. Within RPM, the answer that is returned depends on “who” you ask. The global mesh returns the global count. A partition model entity (a partition or a partition boundary) provides the number classified on that partition model entity. A geometric entity returns the number associated with that geometry. Traversal functions hide the partitioning when possible, and, like entity-count operators, separate operators are provided for each type of traversal.

An important feature is the ability to traverse mesh entities by geometric entity or by partition. This is much more efficient than traversing all entities and checking the classification (geometric or partition) of each to determine whether it is one we wish to use.

3.9 Inter-Process Migration

The approach to mesh migration is similar to the one described by Flaherty *et al.* [14], but is implemented at a more abstract level. To migrate a subset of the mesh, the subset is first reclassified on a new partition model entity. This reclassification is very inexpensive: it involves two pointer updates and an entity count adjustment. Most of the partition reclassifications will not involve a change to the topology of the partition model. More extensive operations are only necessary in those cases when the reclassification produces new partition adjacencies. Following reclassification of mesh entities on the new partition, the partition is migrated to the destination process (abstractly, a "reclassification" of the partition on the process model) and migration of the associated mesh follows. Some entities on interpartition boundaries become the interprocessor boundary. Since the new partition boundary structures already exist, the interprocess boundary update is straightforward. This abstraction of migration allows appropriate packing to avoid the inefficiency of using frequent, small messages. This strategy requires the ability to assign multiple partition model entities to a process and dynamic creation and deletion of partition model entities.

4 Network Performance Comparisons

Specific operations were run in a variety of parallel environments to appraise differences in communication devices at both the hardware and software levels. All cases used MPI for communication. IBM's implementation of MPI was used on their SP2, while others used the MPICH package [17]. MPICH is freely available and runs on architectures ranging from networks of workstations, where it uses p4 as an underlying communication library, to SMPs where it takes advantage of the availability of shared memory as a communication device, and on high-end parallel machines, where it uses the vendor's optimized parallel communication libraries.

The test program simulates work done at the mesh level during a parallel adaptive computation. The specific test sequence was (*i*) load the mesh onto one processor, (*ii*) partition the mesh onto n processors, (*iii*) refine a portion of the mesh, and (*iv*) repartition the mesh to rebalance. Computers involved in the test configurations (Table 1) were a 36-node IBM SP2, with communications over the high-speed switch and over an Ethernet network that connects all nodes; an eight-processor SGI Onyx2 with communication via shared memory; and three dual-processor Sun Ultra 2/2200 workstations, using shared memory and both slow (10Mb/s) and fast (100Mb/s) Ethernet. The intial meshes contained 44,177 elements (coarse case), 79,944 elements (medium case), and 169,733 elements (fine case). In each case, local refinement was applied which increased mesh sizes to 87,170, 158,046, and 453,900 for the coarse, medium, and fine meshes. Tables 2–7 show wall-clock execution times for portions of the code. The "Load" time is the time for the initial mesh to be loaded on a single processor. This is a totally serial process and is used as a consistency check and a gauge of the relative performance of the different processors. "Partition Total" time is the total time to partition the initial mesh. "Partition Migration" time is the portion of the "Partition Total" time that was spent doing message passing for mesh migration during partitioning. “Refine” time is the time spent refining the partitioned mesh. The amount of communication in this step depends on the number of processors that are affected by mesh refinement. “Rebalance Total” time is the total time to rebalance the mesh after refinement, and “Rebalance Migration” is the subset of “Rebalance Total” time spent in message passing for migration during the rebalancing.

Architecture	**IPC vehicle**	**Procs.**	**Computers**	**Key**
IBM SP2	switch	2–32	2–32	SP2s
IBM SP2	Ethernet	2–32	2–32	SP2e
Sun Ultra 2/2200	shared memory	1–2	1	SUNsh
Sun Ultra 2/2200	local p4	1–2	1	SUNlp4
Sun Ultra 2/2200	Ethernet p4	1–6	1–3	SUNp4(10)
Sun Ultra 2/2200	fast Ethernet p4	1–6	1–3	SUNp4(100)
SGI Onyx II	shared memory	1–8	1	SGIsh
SGI Onyx II	local p4	1–8	1	SGIlp4

Table 1
Parallel environments involved in network performance comparisons. Variations in architecture, interprocess communication vehicles, numbers of processes and computers are examined.

These network comparison tests provide insight into some important issues. Given current MPI implementations, the same communication device must be used for all communications, regardless of other available devices. The shared-memory device should be the fastest within a given computer, and it is interesting to know its performance relative to using or not using it. The results indicate that the shared-memory device is not always fastest and the difference between the two communication devices is not as large as expected. This could be due to a number of factors including system overhead, overhead introduced by MPICH, and the quality of the implementation of the shared-memory device. Further investigation is planned. On the Sun and SGI platforms (Tables 3, 4, 6, 7), direct comparisons between the shared-memory and p4 network devices were run. The results were unexpected. The parts of

System	Load	Partition		Refine	Rebalance	
		Total	Migration		Total	Migration
SP2s-2	35.65	14.27	8.37	32.87	15.78	8.79
SP2e-2	36.07	22.83	13.97	33.00	25.44	14.40
SP2s-4	35.69	19.56	12.27	20.60	53.37	4.81
SP2e-4	35.89	44.92	27.79	21.59	72.38	13.87
SP2s-8	36.02	20.99	14.34	11.54	23.01	4.54
SP2e-8	38.00	61.29	33.34	16.32	62.55	20.23
SP2s-16	36.10	21.50	15.39	11.39	12.48	4.58
SP2e-16	36.47	90.42	37.99	59.27	87.09	36.39
SP2s-32	37.21	25.12	16.79	8.99	8.32	3.59
SP2e-32	48.08	131.18	39.22	93.62	269.55	181.43

Table 2
Wall-clock times for network tests using coarse meshes on the IBM SP2.

System	Load	Partition		Refine	Rebalance	
		Total	Migration		Total	Migration
SUNsh-2	9.27	7.13	4.32	15.79	8.04	4.25
SUNlp4-2	7.48	7.03	4.04	15.81	8.14	4.41
SUNp4(10)-2	7.64	15.62	9.59	20.16	18.81	10.81
SUNp4(10)-4	7.66	19.60	12.34	11.64	21.45	8.46
SUNp4(10)-6	7.77	25.04	15.04	7.54	25.94	10.99
SUNp4(100)-2	8.14	8.38	4.84	16.10	9.58	5.18
SUNp4(100)-4	7.82	10.55	6.76	10.19	11.30	3.23
SUNp4(100)-6	7.67	11.21	7.54	5.98	7.91	2.71

Table 3
Wall-clock times for network tests using coarse meshes on Sun Ultra 2/2200 workstations.

the code which require communication typically ran at the same or slower speed relative to the shared-memory device. The serial portion of the code (loading the mesh) was significantly slower when using the shared memory device, especially on the Sun computer. One speculation for the reason for this is that the shared-memory device uses threads, forcing the use of thread-

System	Load	Partition		Refine	Rebalance	
		Total	Migration		Total	Migration
SGIsh-2	6.59	7.94	5.01	17.65	9.18	5.37
SGIlp4-2	6.59	8.44	5.31	18.02	9.77	5.62
SGIsh-4	6.69	10.63	7.65	11.35	8.21	3.35
SGIlp4-4	6.64	11.28	7.85	11.59	8.76	3.47
SGIsh-6	6.62	10.80	8.18	6.21	5.44	2.09
SGIlp4-6	6.65	12.12	8.62	6.63	6.78	2.66
SGIsh-8	6.71	11.46	8.77	6.17	5.54	2.84
SGIlp4-8	6.59	13.24	9.05	6.99	7.04	3.42

Table 4
Wall-clock times for network tests using coarse meshes on the SGI Onyx.

System	Load	Partition		Refine	Rebalance	
		Total	Migration		Total	Migration
SP2s-2	64.83	26.07	16.02	58.44	35.41	16.76
SP2e-2	64.83	42.78	26.75	59.11	51.39	26.54
SP2s-4	64.64	41.35	22.58	32.34	162.70	10.02
SP2e-4	65.14	68.11	39.85	32.96	203.01	28.33
SP2s-8	65.03	38.61	25.95	26.66	67.44	11.58
SP2e-8	65.73	110.40	55.05	38.82	179.34	43.08
SP2s-16	65.19	38.56	27.70	24.60	35.86	12.04
SP2e-16	68.26	129.74	63.39	52.15	177.55	102.16
SP2s-32	65.97	41.84	29.82	14.83	16.82	7.20
SP2e-32	77.11	152.45	64.20	99.56	442.96	325.18

Table 5
Wall-clock times for network tests using medium meshes on the IBM SP2.

safe libraries which are typically slower due to mutual-exclusion locks [16]. We would like to make use of multithreading in the future, so this somewhat disturbing result will be investigated further. On the SGI platform, no significant difference was observed between the two communication devices.

We are also interested in comparing parallel performance of various inter-

System	Load	Partition		Refine	Rebalance	
		Total	Migration		Total	Migration
SUNsh-2	40.56	26.63	15.87	99.33	55.81	30.14
SUNlp4-2	32.91	26.60	15.81	97.11	54.71	29.58
SUNp4(10)-2	33.11	59.28	36.52	121.75	129.95	78.35
SUNp4(10)-4	33.78	102.13	46.55	51.26	208.63	48.03
SUNp4(10)-6	33.45	91.21	58.32	37.29	206.42	60.08
SUNp4(100)-2	33.15	29.64	17.93	95.64	60.79	33.52
SUNp4(100)-4	34.39	70.20	25.91	50.43	156.44	20.65
SUNp4(100)-6	33.27	45.80	30.05	34.44	114.81	16.39

Table 6
Wall-clock timings for network tests using fine meshes on Sun Ultra 2/2200 workstations.

System	Load	Partition		Refine	Rebalance	
		Total	Migration		Total	Migration
SGIsh-2	28.04	33.64	20.38	131.38	74.37	38.94
SGIlp4-2	28.93	34.25	21.05	131.61	73.60	40.54
SGIsh-4	29.57	50.63	31.28	66.88	97.59	25.50
SGIlp4-4	28.41	52.39	31.33	61.82	83.65	24.89
SGIsh-6	27.85	45.32	33.21	43.47	55.54	15.68
SGIlp4-6	28.06	47.49	34.76	42.24	59.43	18.40
SGIsh-8	29.34	51.52	36.37	33.88	73.18	25.51
SGIlp4-8	27.54	51.47	36.12	32.28	76.58	28.25

Table 7
Wall-clock times for network tests using fine meshes on the SGI Onyx.

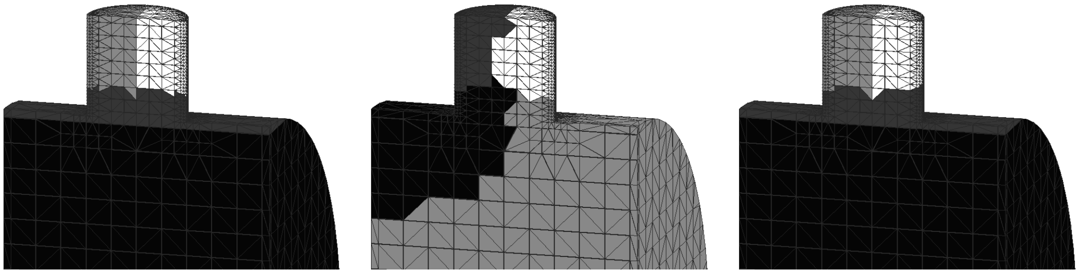
connection networks. Scalability for clusters of workstations would be to be limited to systems having approximately 10 computers due to bandwidth and latency issues. The procedures tested performed about as well as expected and in some cases a bit better. Comparisons of the runs on the IBM SP2 using the high-speed switch and Ethernet (Tables 2, 5) show that the Ethernet performance was fairly good up to about 8 processors. Times on Ethernet were higher as expected, but as the number of processors increased, total execution time, especially for refinement, decreased. Above 8 processors, not unexpectedly, the performance of the Ethernet falls rapidly. Similar, but less dramatic effects can be seen in the tests run between the Sun computers on 10 Mbs Ethernet and 100 Mbs fast Ethernet (Tables 3, 6).

In summary, these tests indicate that the mesh refinement and repartitioning codes scale well on reasonably large workstation clusters connected with, *e.g.*, fast Ethernet. The differences in the communication networks on this test case underscores the need for knowledge of these differences when trying to achieve a good partition for an adaptive computation.

5 Load Balancing Within RPM

Communication costs associated with interprocess boundary entities are useful in a partitioner. The availability of the information in the process and machine models allows a partitioner to avoid large communication volume on interprocess boundaries which communicate across a slow network. Many partitioners attempt to minimize the total size of the interprocessor boundaries, but that is not always equivalent to minimizing the actual communication cost. Like the computational costs of elements, the communication costs associated with boundary entities are not necessarily uniform.

Figure 13 shows three 4-way partitionings of the initial mesh used in a simulation of a perforated shock tube [13]. The mesh contains 44,177 regions. We consider only two factors in partition quality for this example, although other factors [6] must be considered for a more thorough analysis. We consider computational balance, and two *surface index* measures. The *global surface index* (GSI) measures the overall percentage of mesh faces which are on interpartition boundaries, and the *maximum local surface index* (MLSI) measures the maximum percentage of interpartition boundary faces on any one partition.



(a) (b) (c)

Fig. 13. Examples of partitioning using additional knowledge of parallel environment. Partitioning using (a) PSIRB, (b) ParMetis, (c) multilevel combination of PSIRB and ParMetis. Shading indicates processor assignments.

Partitioning using parallel sort inertial recursive bisection (PSIRB) [24] (Figure 13a) achieves an excellent computational balance, with three processors assigned 11,044 regions, and the fourth 11,045 regions, and reasonable surface indices with MLSI=4.67 and GSI=1.53. A partition such as this would be useful when the primary concern is a good computational balance, such as in a shared-memory environment. The partition using ParMetis [22] (Figure 13b) achieves excellent surface index values, MLSI=2.72 and GSI=1.00, but at the expense of a larger computational imbalance. Here, the partitions contain 10,731, 11,369, 10,732, and 11,345 regions. For a computation running on a NOW, it may be worth accepting the load imbalance to achieve the smaller communication volume.

When executing on a pair of two-way SMP nodes connected by a network, a partition like the one shown in Figure 13c may be desirable. Here, a combination of ParMetis and PSIRB is used. ParMetis is used to divide the computation among the two SMP nodes, resulting in partitions of 21,414 and 22,763 regions, with MLSI=0.99 and GSI=0.48. Within each SMP, the mesh is partitioned with PSIRB, resulting in the partitioning of Figure 13c, where the partitions contain 10,707, 10,707, 11,381, and 11,382 regions, and the overall MLSI=5.18 and GSI=1.69. However, since the extra effort was made to minimize communication across the slow network, this will be a very effective partitioning for this environment.

A common operation when optimizing partitions is an interprocess boundary smoothing [10,12,14,21] which exchanges elements across interprocess boundaries to decrease communication volume. This operation can provide significant improvement in surface index values, especially when applied to partitions using spatial cuts to introduce a partition boundary (PSIRB or an Octree-based partitioner [15,20]). However, this improvement is achieved at the cost of the smoothing operation itself and the small load imbalances introduced. Depending on the parallel environment, this may or may not be worthwhile and represents an additional way of taking advantage of information provided by RPM.

The measurement of load imbalance in our existing algorithms is based on a weighted number of elements. Some sources of heterogeneity, such as the extra work on higher-order elements in adaptive p-refinement or the extra time steps taken on smaller elements in a local refinement method [15], can be accounted for using the weighted element scheme. Others, like the run-time imbalance caused by other processes in a nondedicated parallel environment, are not known *a priori* with respect to a given step and do not fit well into this model. Even when it is possible to estimate appropriate weights, there may be considerable effort involved. It will be possible to measure load imbalance dynamically by instrumenting the communication libraries to measure synchronization delays.

6 Further Enhancements

The idea of a "partition" may be extended to add functionality for overlapping mesh migration with computation (called a `PipePartition`), and for computations on meshes too large to fit in memory (the `FilePartition`). A

`StandardPartition` implements the behavior of a partition described previously. For a `StandardPartition` the mesh resides in memory within a single process. The `PipePartition` allows migration to be done incrementally as mesh entities are moved into the partition, while the `FilePartition` allows entities to be migrated to disk or other off-line storage. Each is derived from a generic `Partition`.

When a `PipePartition` is created it is given a destination partition within another process. The entities to be migrated are moved from their existing partition to this `PipePartition`. However, in the background, the entities are actually moved to the destination partition (in appropriately sized chunks) to maximize communication efficiency while the rest of the entities are being moved into the partition. In the final step, the partition is "sent" to the other process; however, at this point most of the data should already be sent. This step just packs and sends the remaining mesh entities. The most effective way to do this would be to have the `PipePartition` spawn a separate thread to transfer the entities. This would require the transfer of entities to the partition to be thread safe.

The `FilePartition` is a similar concept, but its destination is a file rather than another partition. The primary use of this would be to act as a coarse level paging scheme so that a portion of the mesh of a large problem could be loaded into memory and processed while the remainder is kept on disk. This may also be used as the means of storing the entire partitioned mesh to disk.

It remains to show how well the algorithms used will scale to computers with tens of thousands of processors. Algorithms which frequently compute global reductions and synchronize globally will break down in such an environment. There is very likely to be some hierarchy with groups of processors connected by fast and slow network interfaces, making the hierarchical partition model a valuable tool. Additionally, the ability to store multiple partitions per process could allow partitioning to be inherently multilevel, allowing more global algorithms to work with the coarse structure of the partitions, while balance among partitions could be done via local neighborhood optimizations.

Acknowledgement

Some authors were supported by the National Science Foundation through grants ASC-9318184 and CCR-9527151, by AFOSR grant F49620-95-1-0407, by ARO grant DAAG55-98-1-0200, and by Simmetrix, Inc. Computer systems used include the IBM SP systems at the Maui High Performance Computing Center, and the IBM SP2 at Rensselaer.

References

[1] M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. *Int. J. Numer. Meth. Engng.*, 40(9):1573–1596, 1997.

[2] M. W. Beall and M. S. Shephard. A geometry-based analysis framework. In *Advances in Computational Engineering Science*, pages 557–562, Forsyth, GA, 1997. Tech. Science Press.

[3] R. Biswas, K. D. Devine, and J. E. Flaherty. Parallel, adaptive finite element methods for conservation laws. *Appl. Numer. Math.*, 14:255–283, 1994.

[4] R. Biswas, L. Oliker, and A. Sohn. Global load balancing with parallel mesh adaption on distributed-memory systems. In *Proc. Supercomputing '96*, Pittsburgh, 1996.

[5] C. L. Bottasso, H. L. de Cougny, M. Dindar, J. E. Flaherty, C. Özturan, Z. Rusak, and M. S. Shephard. Compressible aerodynamics using a parallel adaptive time-discontinuous Galerkin least-squares finite element method. In *Proc. 12th AIAA Appl. Aero. Conf.*, number 94-1888, Colorado Springs, 1994.

[6] C. L. Bottasso, J. E. Flaherty, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. The quality of partitions produced by an iterative load balancer. In B. K. Szymanski and B. Sinharoy, editors, *Proc. Third Workshop on Languages, Compilers, and Runtime Systems*, pages 265–277, Troy, 1996.

[7] K. Clark, J. E. Flaherty, and M. S. Shephard. *Appl. Numer. Math., special ed. on Adaptive Methods for Partial Differential Equations*, 14, 1994.

[8] K. D. Devine and J. E. Flaherty. Parallel adaptive *hp*-refinement techniques for conservation laws. *Appl. Numer. Math.*, 20:367–386, 1996.

[9] M. Dindar, A. Lemnios, and M. Shephard. Adaptive solution procedures for rotorcraft aerodynamics. Submitted for publication, 1997.

[10] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In D. Bailey, editor, *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 615–620. SIAM, February 1995.

[11] 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.

[12] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int. J. Numer. Meth. Engng.*, 36:745–764, 1993.

[13] J. E. Flaherty, M. Dindar, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. An adaptive and parallel framework for partial differential equations. In D. F. Griffiths, D. J. Higham, and G. A. Watson, editors, *Numerical Analysis 1997 (Proc. 17th Dundee Biennial Conf.)*, number 380 in Pitman Research Notes in Mathematics Series, pages 74–90. Addison Wesley Longman, 1998.

[14] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for

adaptive finite element computation. *Appl. Numer. Math.*, 26:241–263, 1998.

[15] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. IMA Preprint Series 1483, Institute for Mathematics and its Applications, University of Minnesota, 1997. To appear, *J. Parallel and Dist. Comput.*

[16] W. Gropp. *Personal communication.*

[17] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.*, 22, 1996.

[18] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI: A Message Passing Interface Standard*, first edition, 1994.

[19] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI-2: Extensions to the Message-Passing Interface*, first edition, 1997.

[20] T. Minyard and Y. Kallinderis. Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations. *Int. J. Numer. Meth. Fluids*, 26:57–78, 1998.

[21] T. Minyard, Y. Kallinderis, and K. Schulz. Parallel load balancing for dynamic execution environments. In *Proc. 34th Aerospace Sciences Meeting and Exhibit*, number 96-0295, Reno, 1996.

[22] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Tech. Report 97-014, University of Minnesota, Department of Computer Science and Army HPC Center, Minneapolis, MN, 1997.

[23] M. S. Shephard, S. Dey, and J. E. Flaherty. A straight forward structure to construct shape functions for variable p-order meshes. *Comp. Meth. in Appl. Mech. and Engng.*, 147:209–233, 1997.

[24] M. S. Shephard, J. E. Flaherty, C. L. Bottasso, H. L. de Cougny, C. Özturan, and M. L. Simone. Parallel automated adaptive analysis. *Parallel Comput.*, 23:1327–1347, 1997.

[25] V. Vidwans, Y. Kallinderis, and V. Venkatakrishnan. Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids. *AIAA J.*, 32(3):497–505, 1994.

[26] C. H. Walshaw and M. Berzins. Dynamic load balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995.

[27] C. H. Walshaw, M. Cross, and M. Everett. Mesh partitioning and load-balancing for distributed memory parallel systems. In *Proc. Par. Dist. Comput. for Comput. Mech.*, Lochinver, Scotland, 1997.