PERGAMON Computers and Mathematics with Applications 46 (2003) 105–123

An International Journal
computers & mathematics
with applications

www.elsevier.com/locate/camwa

Data Structures and Load Balancing for Parallel Adaptive *hp* Finite-Element Methods

A. K. PATRA*, A. LASZLOFFY AND J. LONG
State University of New York, Buffalo, NY 14260, U.S.A.
<lend><j124><abani>@eng.buffalo.edu

Abstract—Adaptive *hp* finite-element methods (FEM), in which both grid size h and local polynomial order p are dynamically altered, are very effective discretization schemes for the numerical solution of a large class of partial differential equations. However, these schemes generate computations that require dynamic and irregular patterns of data storage, access, and computation, making their use on multiprocessor machines very difficult. We describe here the development of a suite of data structures and load balancing techniques that addresses these concerns. The central idea is the use of a spatially local ordering of all data and computation using a key generated from geometric data using a space filling curves-based ordering for data storage, distribution, and access. We also evaluate the suitability of tree and table type data structures for adaptive meshing. Example applications and performance data complete the presentation.

Keywords—Adaptive *hp* finite-element methods, Parallel computing, Data management, Load balancing.

1. INTRODUCTION

Adaptive *hp* finite-element methods can yield highly advantageous cost/accuracy ratios. Several researchers [1–5] have, in fact, shown that the reduction in discretization error with respect to the number of unknowns can be exponential for general classes of elliptic boundary-value problems when properly designed meshes are used ($O(\exp^{-\gamma N})$ where N is the number of unknowns and γ is a constant). The methods are intrinsically complex and this is reflected in the complexity of the techniques required to manage the data and computations associated with these schemes (see [4] for an early implementation). Parallel versions of these methods offer the potential for very accurate solution of large scale physically realistic models of important physical systems. However, parallel processing introduces the following additional difficulties:

- need to dynamically allocate and deallocate memory in a distributed memory environment as the computation proceeds;
- need to maintain refinement/enrichment constraints during the adaptive process;
- need to balance the computational load dynamically among the multiple processes;

The financial support of the National Science Foundation through Grant ASC9702947, ACI0102805 is acknowledged. Computer time was provided by the National Partnership for Advanced Computing Infrastructure, San Diego, and Center for Computational Research, University at Buffalo.
*Author to whom all correspondence should be addressed.

0898-1221/03/$ - see front matter Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
PII: S0898-1221(03)00110-X

- need to use linear equation solvers, designed to exploit the special structure of the irregular sparse systems generated from adaptive *hp* schemes.

These problems have attracted much attention in recent years and several approaches have been published, especially, for the related though simpler problems of adaptive mesh refinement with *h*-version finite elements and finite differences. The contributions of Berger and Saltzman [6] with the CHAOS++ system [7], Baden *et al.* [8] with the KeLP system, and Parashar and Browne [9] with the DAGH system have been particularly notable. The KeLP [8] programming system developed by Baden *et al.* used simple geometric abstractions which enable the programmer to conveniently express complicated communication patterns for dynamic applications. Parashar and Browne [9] developed the distributed adaptive grid hierarchy package (DAGH) for supporting parallel adaptive mesh refinement using hierarchical finite-difference grids. The load balancing and processor mapping strategy of DAGH that maps grids to processors through locality-preserving space-filling curves (SFC) is one that we adopt in the current work.

More recently, there have been a few efforts at supporting parallel *hp* adaptivity. The first one of this kind being the scalable distributed dynamic arrays (SDDA), developed by Edwards and Browne [10,11], SDDA supports fine-grain data parallel operations on arrays distributed across processors. It automatically manages data decomposition, interprocessor communication, and synchronization. SDDA has been applied to two-dimensional problems using adaptive *hp* finite-element methods. Another large family of integrated tools designed to support *hp* schemes has been developed by Flaherty, Devine, Shephard, Loy, Ozturan *et al.* [12–15] using octree-based structures for mesh generation and data decomposition. The finite-element data is stored in the parallel mesh database based on the octree decomposition. Dynamic load balancing, refinement, and derefinement algorithms are also provided by this library.

In this paper, we describe a set of distributed dynamic data structures and load balancing schemes that address the needs listed above. The central thesis in our data management schemes is that the complexity of the task can be greatly alleviated by using a global index space based on a *spatially local ordering of the data*. The primary contributions of this work are

- support for adaptive *hp* finite-element schemes using constrained node-based approximations (Figure 1) necessary for using all-quadrilateral and all-hexahedral meshes,

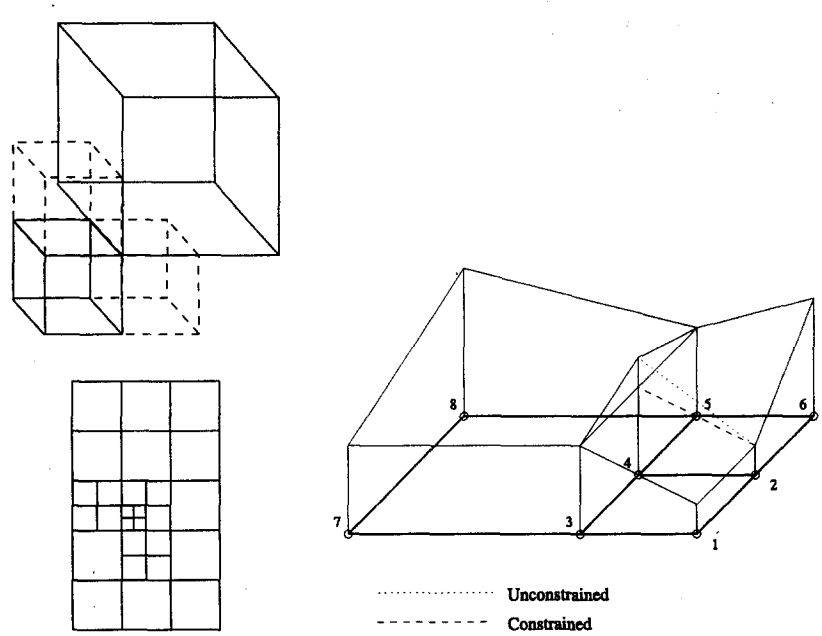


Figure 1. Constrained node approximations to maintain continuous approximations across interelement boundaries.

- a clean classification of finite-element data and a delineation of the requirements for data structures necessary to support parallel adaptivity,
- a clear evaluation of the suitability of tree and table type data structures for parallel adaptive meshing,
- the use of predictive strategies for dynamic load balancing that greatly reduce excessive data migration during rebalancing of load among processors, and finally,
- providing very close integration with equation solvers designed for such grids.

We begin by describing the data that has to be managed for an adaptive finite-element simulation and basic requirements for data structures suitable for this application. We then describe two suitable distributed dynamic data structures, two simple schemes for load balancing, and a brief review of solution algorithms. A presentation of extensive numerical tests completes the paper.

2. FINITE-ELEMENT DATA AND REQUIREMENTS FOR DATA STRUCTURES

2.1. *hp* Finite-Element Data

Adaptive finite-element simulations generate data that can be decomposed into persistent mesh/solution data and transient computational data. The mesh data is retained and modified through multiple solution cycles, and hence, is labeled as persistent, while during each solution cycle, large amounts of temporary computational data are generated. The transient computational data is fairly voluminous but has a simple matrix/vector structure. The persistent mesh data is complex and comprises geometric location, node connectivity, neighbor information, parent/child element information, local orders of approximation, material, and solution data. All of this data is also easily associated with nodes and/or elements (see Figure 2). Note the ownership relationship between the mesh and computed data. This relationship will be central to our data parallel computing strategy where computations will be distributed among processors using an "owner-computes" strategy. This complex data has to be stored and accessed efficiently in properly designed parallel data structures. The choice of the data structure is vital for this application, and a bad choice can largely destroy advantages obtained by adaptivity and parallel computing. We will now clearly delineate the requirements of a data structure to store this data efficiently.

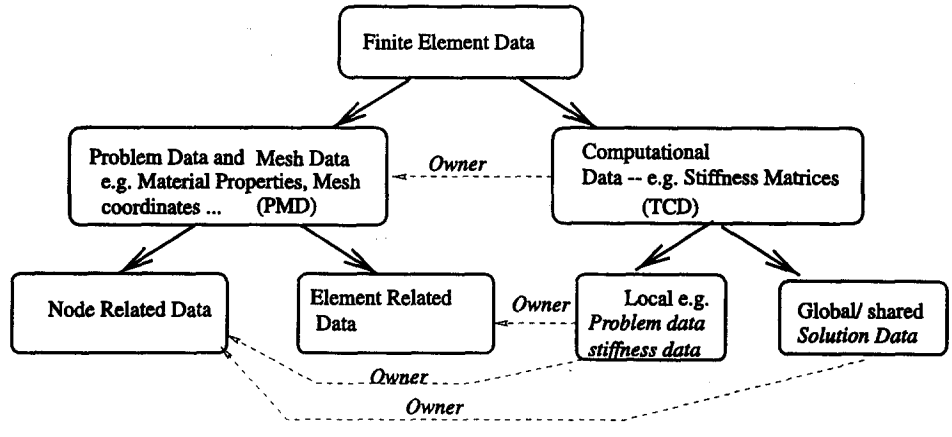


Figure 2. Organization of finite-element data into persistent mesh data and transient computational data. Note the ownership relations among the data sets.

2.2. Data Structure Requirements

The most important requirements of a data structure storing mesh data in supporting parallel *hp* adaptive FEM are as follows.

- *Dynamic data storage.* During adaptive mesh refinement/derefinement and mesh redistribution, objects are created/transferred and have to be inserted; and old objects are removed and have to be deleted from the data structure. In other words, the storage scheme must be able to grow/shrink at runtime.
- *Fast storage and retrieval.* Quite often, one particular object has to be recovered from the database and modified, e.g., neighbor of a refined element for information update, etc.
- *Fast data set traversal.* Some of the procedures in the solution process (e.g., stiffness matrix construction) involve operations which must be performed on all, or a list of the entities. In this case, a sweep through these objects is necessary. In other words, the whole or parts of the data structure have to be traversed in sequence.
- *Uniqueness of the identifier.* Each object in the data set should possess an identifier that is unique and remains so as the mesh evolves and new objects are introduced and old ones removed. Renumbering (reidentifying) schemes are not trivial to implement and expensive, especially in the case of parallel processing.
- *Memory and cache efficiency.* Data has to be stored with the most efficient memory usage possible. The aim is to use the local memory associated with the processor and avoid reading from/writing to remote memory as much as possible. It is also preferable to maximize cache hits when processing the data. An effort has to be made to store data in memory in the order in which it is accessed to maximize cache reuse. On superscalar processors, such cache reuse can greatly increase efficiency.

Unfortunately, a scheme efficient in one criteria might not be efficient in another, and usually an acceptable balance should be found. For example, linked lists [16] are very suitable for full set traversal and memory efficiency requirements but have poor data retrieval properties. Tables are fast access but can be difficult to grow and shrink. Trees can be easy to grow and shrink but make access complex and slow. In this work, the local data structure has been implemented using two different schemes, namely, B-trees and hash tables. Both are dynamic data structures with different characteristics. The unique identifier is obtained from a key given by a space-filling curve (SFC) discussed in the next section.

3. SFC-BASED INDEXING

The starting point of the data structure design is the construction and assignment of an indexing using identifier (a key) assigned to each piece of persistent mesh data. Since all of the mesh data may be related to either a node or an element object, we begin by assigning keys or identifiers to each node/element object based on their geometric location. This key has to be associated with the object at the time of creation and ideally should be unique throughout the computation, or until the entity vanishes. Following Edwards [11], we will use here the location along a space-filling curve passing through the element centroid for element keys and one passing through nodes for node keys.

The SFC [17] maps the n-dimensional data to a one-dimensional sequence and in the limit will fill the multidimensional space. Sagan [17] lists a series of such curves and algorithms for obtaining them. The standard procedure for obtaining an SFC is to use a mapping $h_n : [0,1]^n \rightarrow [0,1]$. Thus, to obtain keys for elements,

(1) we scale the coordinates $\mathbf{x}_i = (x_i, y_i, z_i)$ with the maximum of x_i, y_i, z_i over all element centroids,
(2) compute $k_r = h_n(x_i, y_i, z_i)$, and

(3) convert k_r into an integer using $k = \text{int}\,(k_r * B)$, where B is a large integer, e.g., 10^8.

Note that schemes for constructing h_n were published in [18].

A sorted list of the keys $C = \{k_i\}$ defines a space-filling curve passing through the points $\mathbf{x}_i$. Because the mapping is purely based on coordinates, the key remains unique over the entire computation in the case of stationary meshes. It should be noted that the use of this type of key effectively generates a global address space. This space can be easily partitioned to obtain a decomposition of the problem.

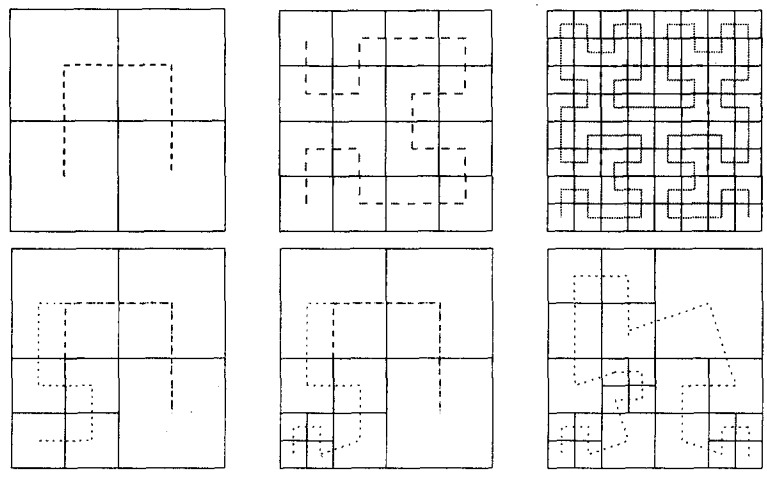


Figure 3. Space filling curve orderings of element centroids. Levels 1, 2, and 3 curves shown for uniform and adaptive meshes.

Two important characteristics of the SFC ordering pertinent for the case of adaptive meshes are clearly revealed in Figure 3. These are the following.

(1) *Subcube property.* If the entire domain is visualized as a hypercube and then split up into subcubes in a recursive fashion, the curve passes through all the points in each subcube at a particular level before going through points in a neighboring subcube.

(2) *Self-similarity.* The curve can be generated from a basic stencil at each level of subcubes.

The primary consequence of the subcube property is *geometric locality*. Thus, points close to each other in the n-dimensional space are also close to each other in the one-dimensional (key) space in the mean sense. Since the FEM process usually requires processing of elements that are geometrically adjacent in sequence, *geometric locality* can be exploited for the efficient storage and retrieval of the mesh data. Further, a geometrically local ordering will lead to the generation of sparse linear equation systems that require less storage and less work to factorize.

The locality properties of SFCs have been systematically explored by Gotsman and Lindenbaum [19] and Perez *et al.* [20] among others. While pointwise locality cannot be guaranteed for all sets of points in multidimensional space, one can show good behavior in a sum or mean sense. To make concrete our observations above, let us now denote the set $[N] = \{1 \ldots N\}$ and define an m-dimensional SFC of length N^m as a bijective mapping $C : [N^m] \rightarrow [N]^m$ such that $d(\mathbf{x_i}, \mathbf{x_{i+1}}) = \mathbf{1}$ for all $i \in [N^m - 1]$ where $d(\cdot, \cdot)$ is the Euclidean metric. A basic topological theorem [21] proves that it is impossible to show that any two points $\mathbf{x_i}, \mathbf{x_j} \in \mathbf{R^n}$, $n = 2, 3$ that are close to each other in a Euclidean distance metric $d(x_i, x_j)$ are also close in the mapped space. However, Gotsman and Lindenbaum [19] were able to prove the converse—"If two points are close in the mapped space then the points will also be close in the above-defined Euclidean metric." We cite below their theorem establishing upper and lower bounds on the locality measure L_m for Hilbert space-filling curves in two dimensions. Looser bounds have also been shown for more general curves.

THEOREM 1. *For a two-dimensional Hilbert space-filling curve C traversing a uniform grid of 4^k points, the locality metric $L_m(C) = \max_{i,j \in C, i<j} d(\mathbf{x_i}, \mathbf{x_j})/|i-j|$ is bounded as*

$$6\left(1 - O\left(2^{-k}\right)\right) \leq L_m(C) \leq 6\frac{2}{3}. \tag{1}$$

4. DATA STRUCTURE SCHEME

In the case of distributed memory architectures (MPP systems), two levels of data storage can be distinguished. The first level is local to a processor, and each processor maintains its own data structure describing its own subdomain. The second level is the collection of local data structures defining the global structure, the parallel mesh database (PMDB). This second level may or may not exist physically. It can be simulated by creating a virtual shared space, and provide mechanisms to operate on remote data without explicit message passing [9,10]. In our design the second level (see Figure 4) is obtained by a simple partitioning of the index space (key set) associated with the element objects. The first level is implemented using structures that facilitate the dynamic memory management necessary for adaptive meshing.

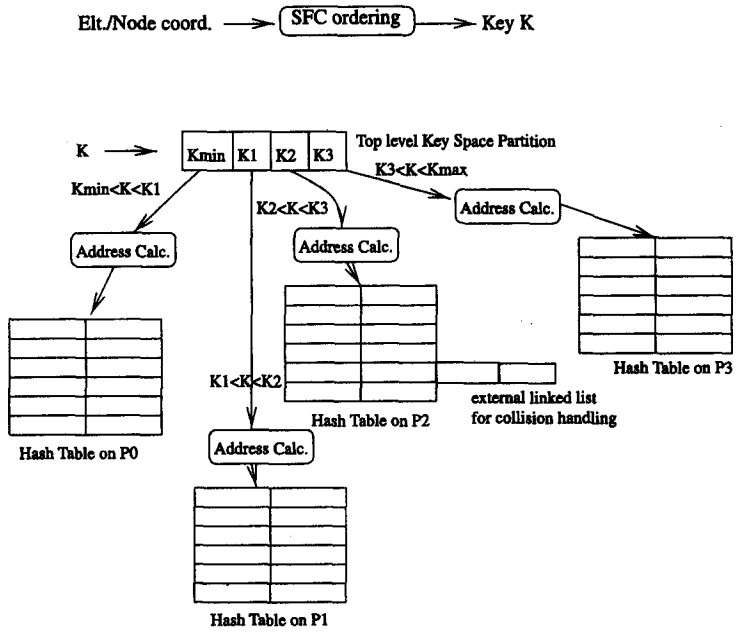


Figure 4. Distributed dynamic data structure—Global index space based on SFC partitioned for data distribution.

Of all the fundamental data structures, namely stacks, queues, lists, graphs, tables, binary, and multiway trees [16], only trees and tables are clearly suitable for adaptive meshes. Stacks, queues, and lists are clearly disadvantageous since data is often accessed in an unpredictable and irregular pattern. Graphs can be useful if the mesh is represented by a graph, although data structures based on graphs are hard to design and maintain. We have implemented the remaining two types of data structures, i.e., tables and trees.

In our parallel mesh database, each processor maintains two data structures for storing element and node objects. The mesh data related to nodes and elements are structured into object data types defined in the C++ language. These objects contain all data associated with the node/element. Furthermore, they provide functions to operate on the data. The local data structures holding these entities have been implemented using both hash tables and B-trees. During the experiments both (node and element) data structures used the same scheme; however,

hybrid implementation is also possible, and requires further consideration. We now describe the two schemes. Note that tree/hash structure is local to a processor and stores all the data belonging to a processor (see Figure 4). Redistribution will involve deleting objects from one processor's data structure and inserting it in another processor's data structure.

4.1. B-Tree

A *B-tree* [16,22] is a special type of tree with properties that make it useful for sorting and retrieving large sets of information. A tree-based data structure stores the data in its nodes and a predefined relation (parent-child) exists among these nodes. As shown in Figure 5, finding an element stored in a B-tree requires searching only a single path between the root (parent node) and a leaf (node without any children).

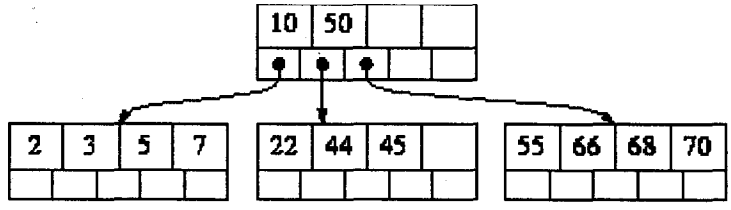


Figure 5. B-tree with order 5.

The insertion and deletion algorithms guarantee that the longest path between the root and the leaf is $O(\log_2 N)$. It should be noted, adaptive meshing inherently leads to tree-like structure. Each node may contain $m-1$ data and may have m children. Because of this, and because it is well balanced, the B-tree provides the possibility of very short path lengths for accessing very large collections of elements. Furthermore, related records are kept on the same disk or memory page which takes advantage of the geometric locality. Thus, the B-tree can be adjusted to account for hierarchical storage (CPU, Cache, RAM, disk, etc.) and can provide high cache efficiency. In our implementation, a B-tree node contains the keys of its records, and references (pointers) to the node/element objects.

4.2. Hash Tables

In a hash table structure, data elements are kept in an array-based data structure, called the hash table [16,22]. The table is divided into buckets (slots) which are basically the elements of a one-dimensional array. Based on a key, the address calculator (hashing) function determines in which slot of the array the data should be located. The process of accessing a record by mapping a key value to a position is called *hashing*. In other words, the hashing function (h) is a mapping from the key space to the physical memory address space.

$$\text{memory\_address} = h(\text{key}).$$

Theoretically, the scheme has very fast data retrieval ($O(1)$ if there is no collision), data insertion, and deletion properties. However, collisions (two data items hash to the same key) must be appropriately handled. Slots in the table are wasted since collision minimization requires some sparsity resulting in poor memory and cache efficiency compared to trees. The efficiency of the mapping is critical for good performance; i.e., the function should be able to give different slots for different keys. In order to achieve this, clearly the number of slots must be more than the number of existing keys. However, satisfying this criteria is in general still not enough to avoid collisions. Moreover, in the case of AMR, the number of objects increases during the simulation. Collision occurs when the particular slot given by h is already occupied by another data member. Several methods exist to handle collisions. To deal with the problem, we use separate linked lists

to store all the items with a conflict. This is called *external chaining* (Figure 6). Other possible schemes are *linear rehashing*, or *double hashing* using more complex strategies.

When the table becomes full, a new array can be created with double the size of the original (*extendible hashing*). For insertion and deletion, the place in the array is located by the hashing function. If collision occurs, the true location is obtained from the collision handler. Now the data can be inserted or deleted, but depending on the collision handling, further steps might be required (e.g., establish/change pointers to next and previous data members in the case of external chaining).

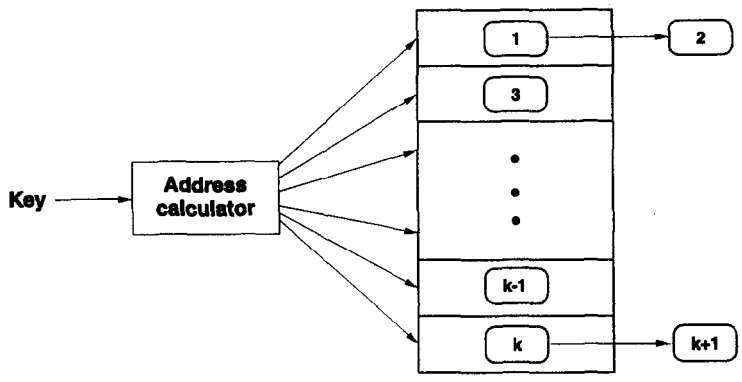


Figure 6. Hashing with external chaining.

In the case of multiple collisions, traversal of the hash table can be expensive when compared to the traversal of the trees. In our implementation, the hashing function uses a projection of the key space to the memory space, i.e.,

$$h = \frac{\text{key} - \text{min\_key}}{\text{max\_key} - \text{min\_key}} * \text{array\_size},$$

where key represents the object to be looked up, *min_key* is the lowest, and *max_key* is the highest key of the domain. Collision handling is solved with external chaining so the *hash_entry* structure contains two pointers to the previous and next structures in the chain. Moreover, the key of, and a pointer to, the actual object (element/node) is stored in the entry structures.

4.3. Advantages of the SFC-Based Data Structure

We may now summarize the main advantages of this data structure as the following.

- *Fast key generation.* The self-similarity property leads to a series of inexpensive computer algorithms to create the keys using basic bit interleaving operations [17].
- *Uniqueness.* Due to the uniqueness of the key, a simple indexing scheme can be maintained using minimal sorting. No complex renumbering of the objects is necessary when data is inserted or deleted. In the case of conventional array index-based data structures, renumbering is unavoidable after changes to the mesh during refinement.
- *Global address space.* Again, because of the uniqueness, no local to global mapping of the identifiers is required. Furthermore, because the elements are assigned to processors in contiguous lists, the knowledge of the key of an element and the two extremes of the local key space of all subdomains will determine in which subdomain an element is situated; i.e., given an element or node's coordinates, its key, and hence, its storage location including owning processor is very easy to obtain.
- *Data locality.* Because of the inherent geometric locality property of the ordering, a "smart" data structure that stores adjacent keys in adjacent locations in the physical memory will have the advantage of data locality and all of its attendant advantages.

- *Support for hierarchical AMR.* The coarse grid is the basis of the grid hierarchy and it is represented by a simple list of keys. As the simulation evolves, newly created elements are incorporated within the hierarchy as shown in Figure 7. These new elements form a new level, and can be interpreted as a sublist corresponding to the refined region. This structure is the result of the subcube property, and it is useful for maintaining data locality and connection between the different levels. This further results in easy implementation of coarsening (derefinement) strategies. The feature has been explored by Parashar and Browne in [9]. Moreover, a tree-like data structure can be implemented based on the hierarchy. The children elements are the siblings and the parent is the root of the subtree, resulting in a quadtree in two- (one to four refinement) and an octree in three-dimensions (one to eight refinement).

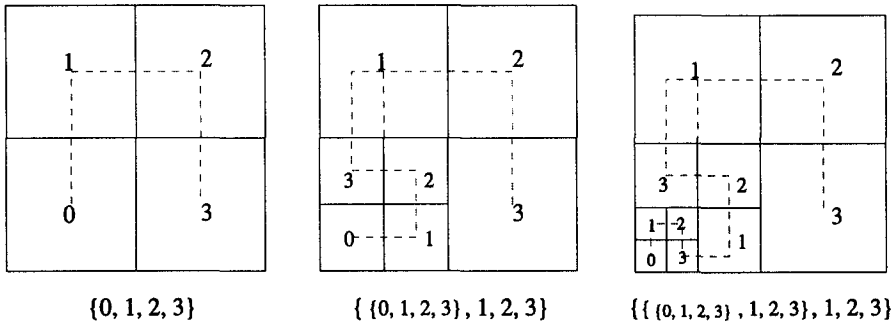


Figure 7. Hierarchical AMR.

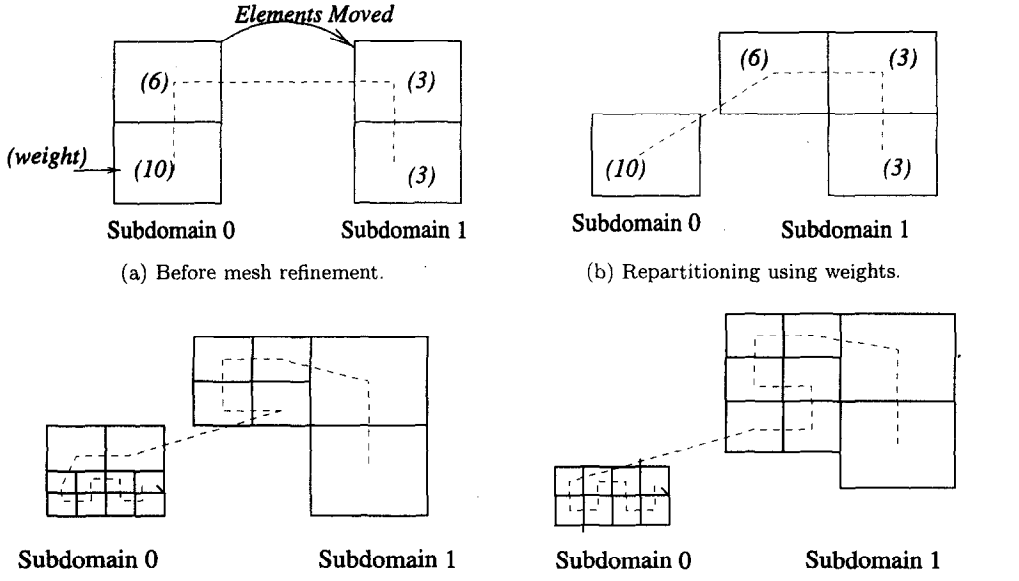
5. LOAD BALANCING AND SUPPORT FOR IRREGULAR COMPUTATIONS

The second part of effective data management for parallel adaptive computation is the balancing of load as the computation evolves. Adaptive mesh refinement can result in highly irregular patterns of computational load that change as the mesh is modified. Thus, dynamic load balancing is an important issue in the case of this application. As outlined earlier, our basic data distribution strategy is a partitioning of the key space (see Figure 4). Static partitioning schemes have been presented earlier (see for instance [18,23]).

We outline below two simple algorithms to implement dynamic load balancing. More complex strategies may be seen in the work of Biswas and Oliker [24], and Schloegel, Karypis and Kumar [25]. In all our approaches, we use a strategy outlined in [26] to minimize data migration during repartitioning. We repartition coarse grids based on a weight (typically error estimate) and then create the refined elements on target processors; i.e., we repartition before refinement. Figure 8 outlines the strategy. After the creation of the new elements, a few smoothing iterations may be used to remove any small imbalances. In previous work [26], we have also conducted numerical experiments with more complex graph-based partitioning tools and complex weighting schemes (error as vertex weight, polynomial order as edge weight, etc.) and concluded that while partition quality is usually somewhat better in such approaches, the simplicity and code development ease of the approaches outlined here makes them quite competitive.

5.1. Incremental Repartitioning

Simple incremental repartitioning algorithms are used to transfer elements between the subdomains until required load balance is achieved. Since we want to maintain the contiguous list of keys in a subdomain, elements from the end and/or the beginning of the local key space are shipped to adjacent subdomains (Figure 8). This algorithm can be used for local improvement between (overlapping) pairs of processors and it can also be applied as a global rebalancing strategy.



(a) Before mesh refinement.

(b) Repartitioning using weights.

(c) Repartitioning and mesh refinement.

(d) Repartitioning, mesh refinement, and smoothing.

Figure 8. Incremental load balancing based on the SFC.

In the first case, neighboring subdomains ensure local load balance, and iterative application can lead to global balance. In the second case, elements are relocated to/from a certain subdomain until it reaches the ideal work load per processor (one subdomain is mapped to one processor). Although an SFC-based balancer does not give as high quality partitions as a graph partitioner, it is easy to implement, the algorithm runs fast in parallel, and it is incremental.

5.2. Scratch-Remap Repartitioning

In this strategy, instead of starting with an existing distribution and adjusting partitions, we compute a new distribution from scratch. This strategy is particularly suitable when the mesh size changes rapidly (e.g., by 50% or more elements). Such changes can be seen when we use adaptive strategies that attempt to obtain a mesh for a certain error in one refinement cycle, e.g., [27]. Using a set of global weights, we first determine a final destination processor for every element, and then move the element directly to the destination. The first question to be answered is how to determine the destination of an element after repartitioning. This is achieved easily and in parallel by having every processor set up a current global weight picture (cgwl), and an expected weight picture (egwl) for the entire grid. From these two pictures, every processor can determine the final destination of every element/node object it owns. Since the elements/nodes are stored in a sorted table, this can be done quite simply by going through the table and marking the elements. Each processor can then package and move elements to target processors (see Figure 9).

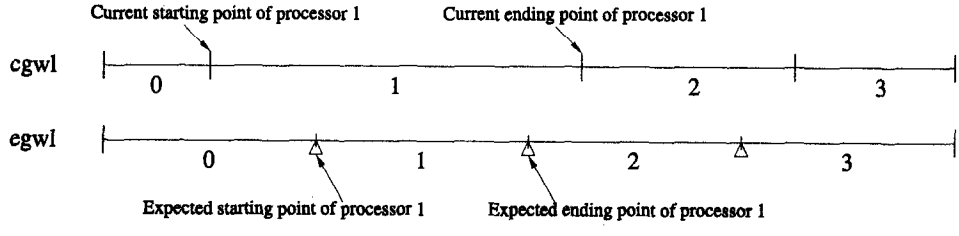


Figure 9. Scratch-remap load balancing based on the SFC.

6. SOLUTION ALGORITHM

The adaptive *hp* FEM schemes discussed in this work give rise to sets of linear systems that are irregularly sparse and poorly conditioned. The irregular sparsity (as opposed to the nice band structure of classical FEM) is a consequence of the different sizes and connectivities of different elements as the polynomial order is changed from element to element. Efficient algorithms for parallel solution of these irregularly sparse systems are not readily available. The poor conditioning (large ratio of maximum and minimum eigenvalues for symmetric positive definite systems) is a result of the simultaneous refinement and enrichment process. Theoretical and numerical evidence suggests that the condition number increases as $O(p^4/h^2)$ [5] in two space dimensions. Numerical roundoff error in the solution process and convergence for iterative solvers are degraded as the conditioning deteriorates. A final difficulty that is encountered in the design of a solver for use across a large class of parallel architectures is the choice of solution method for best efficiency. Different algorithms have very different efficiencies on different architectures. Thus, one must either change algorithms (code) based on architecture or design a solver that is able to adjust to the architecture. In recent years, much effort has been dedicated towards the design of efficient solvers for such grids [28–32] using domain decomposition techniques. Special solver-specific storage techniques must also be developed to store the irregularly sparse matrices. We have experimented with standard variable block row and compressed sparse row storage schemes (see [33] for more details). The locality preserving orderings employed appear to produce smaller envelope sizes and allow fairly large-sized problems to be solved on small memory.

To address these concerns, we have developed a class of special solvers based on multilevel iterative substructuring described in [18,32]. The basic idea of our special solver is to combine direct and iterative solution methodologies. This mix is achieved by organizing the unknowns into a multilevel hierarchy. The bubble, face, and edge functions associated with higher-order elements provide the higher levels of the hierarchy while lower levels can be obtained by clustering patches of elements. Higher levels in this hierarchy are closely coupled and usually assigned to the same local memory. Using substructuring, we can then eliminate these using a direct solver approach. The lower levels are then solved using an iterative solver. The iterative solution is accelerated using a coarse grid type preconditioner. The procedure is detailed in [5].

We have also developed interfaces to popular solver library PETSC (portable extensible toolkit for scientific computation [33]). PETSC provides a wide variety of Krylov space-type iterative solvers with a range of preconditioners including popular domain decomposition preconditioners like the additive Schwartz methods and the algebraic incomplete LU-type preconditioners.

7. NUMERICAL TESTS

7.1. Test Problem

We use two test problems in linear elastostatics. The first is inspired by the problem of analyzing the response of a dental implant to biting loads, and the second is a three-dimensional version of the classic L-shaped domain.

$$\begin{aligned} \sigma_{ij,j} + f_i = 0 \in \Omega \subset R^n, \qquad & n = 2, 3, \\ \sigma_{ij} = 2\mu\epsilon_{ij} + \lambda\epsilon_{kk}, \quad & \epsilon_{ij} = \frac{1}{2}(u_{,i} + u_{,j}), \\ \mathbf{u} = g, \qquad \text{on } \partial\Omega_D, \qquad & \sigma.\mathbf{n} = t, \qquad \text{on } \partial\Omega_N, \end{aligned} \tag{2}$$

where σ_{ij} is the Cauchy stress, f_i is the body force, ϵ_{ij} is the corresponding strain, μ, λ are the Lamè parameters, u is the displacement field, and $\partial\Omega_D$ is the part of the boundary on which Dirichlet boundary conditions are applied and $\partial\Omega_N$ is the part of the boundary on which Neumann boundary conditions are applied. Figure 10 illustrates the first problem.

Table 1. Time in seconds to traverse full data set and perform ordering of the unknowns on eight processors of the Cray T3E and the Origin 2000.

Ordering	Cray T3E				SGI O2000			
Total dof	Hashing		B-tree		Hashing		B-tree	
	8pr	16pr	8pr	16pr	8pr	16pr	8pr	16pr
8598	0.99	0.40	0.12	0.31	0.15	0.23	0.07	0.32
33328	0.71	0.79	0.65	0.66	0.35	0.43	0.25	0.49
133300	12.69	4.34	13.18	4.50	3.30	1.67	3.49	1.89
534082	210.38	61.51	212.16	62.5	49.61	13.93	49.18	14.29

Table 2. Time in seconds to refine elements and propagate constraints on eight and 16 processors of the Cray T3E and the Origin 2000.

Refinement	Cray T3E				SGI O2000			
Elements Refined	Hashing		B-tree		Hashing		B-tree	
	8pr	16pr	8pr	16pr	8pr	16pr	8pr	16pr
1046	0.39	0.31	0.39	0.22	0.11	0.10	0.13	0.15
4184	1.16	0.63	1.716	0.82	0.34	0.20	0.51	0.49
16736	8.24	2.98	11.27	4.38	2.32	0.95	2.83	1.19
66944	101.14	30.63	114.93	36.92	32.10	9.60	33.85	8.82

Table 3. Time in seconds to enrich elements and propagate constraints on eight processors of the Cray T3E and the Origin 2000.

Refinement	Cray T3E				SGI O2000			
Elements Enriched	Hashing		B-tree		Hashing		B-tree	
	8pr	16pr	8pr	16pr	8pr	16pr	8pr	16pr
4184	0.07	0.06	0.13	0.12	0.03	0.02	0.05	
16736	0.21	0.12	0.59	0.29	0.11	0.08	0.22	
66944	0.88	0.48	7.32	1.28	0.59	0.37	0.98	1.23
267776	6.72	3.68	11.67	5.67	5.16	2.59	4.37	1.98

Table 4. Time in seconds to perform dynamic load balancing on eight processors of the Cray T3E and the Origin 2000.

Load Balancing	Cray T3E				SGI O2000			
Elements Moved	Hashing		B-tree		Hashing		B-tree	
	8pr	16pr	8pr	16pr	8pr	16pr	8pr	16pr
648	0.50	0.52	0.57	0.37	0.27	0.29	0.09	0.09
3025	2.9	3.12	1.62	1.53	1.42	1.29	0.51	0.43
11674	16.44	14.69	19.01	6.88	6.67	5.94	4.52	2.23
44582	130.65	84.60	194.82	67.22	43.60	26.17	59.58	21.51

Two major kinds of experiments were carried out. In the first, the performance of the two data storage schemes are compared on two different supercomputer architectures. The second illustrates dynamic load balancing based on the space-filling curve ordering.

7.2. Data Management

Sample results from implementing the above-described data structures in a parallel adaptive *hp* FEM code are presented. The time taken for several representative data management operations

are measured, namely:

(1) time taken for a full data set traversal in order (for visualization or local to global mappings);
(2) time taken to refine/enrich elements;
(3) time taken to rebalance the partitions using data migration.

The initial mesh of the experiments is presented in Figure 10. The results of the experiments are presented in tabular and simple plot format. Tables 1–4 compare the current implementation of the two storage schemes performing the operations listed above. The experimental runs were carried out on a Cray T3E and an SGI Origin 2000 supercomputer. These machines have significant architectural differences (the SGI is a distributed shared memory with cache coherent nonuniform memory while the Cray T3E is a simple distributed memory) and the effect of the architecture on our schemes is of interest. Table 1 demonstrates the time taken to order the unknowns. At the end of the local, subdomain ordering, synchronization and communication are essential to construct the global ordering. Clearly, this communication dominates the whole operation, since there is no remarkable difference between the performances of the B-tree and hashing. However, there is a significant difference between the results on the two architectures, the T3E showing poorer efficiency. Tables 2 and 3 consider the adaptation processes. This experiment traverses the element data structure and refines the mesh repeatedly. During the process, synchronization/communication is necessary for updating information across subdomains. The fact that there are no significant differences between the timings leads to the conclusion that most of the time is spent in communication (very fast runs on the supercomputers do not give reliable results). It is speculated that the small differences are due to the slower traversal, but faster search of the hashing scheme. It should be also noted that since the bit level interleaving operations are architecture dependent, the distributions of the key space are inconsistent between the two architectures. During adaptation, the Origin 2000 generated more closely packed keys than the ones generated by the T3E. This degraded the hashing scheme on the SGI, making both the traversal and search algorithms inefficient in the case of enrichment.

Table 4 demonstrates the timings of the dynamic load-balancing algorithm. This process requires: the location of the elements with lowest/highest key; deletion from the old subdomain data structure; interprocessor transfer; and finally, insertion in the new subdomain data structure of the elements/nodes. Eight processor experiments show the advantage of the hashing, mainly because deletion from a B-tree comes with substantial overhead. However, in the case of the 16 processor runs, the hashing scheme degrades due to slower traversal. This is especially observable when the table is very sparse (small number of elements/nodes).

To further explore the sensitivity of the hashing to the size of the table, several more measurements were carried out with half and one-fourth of the size of the original 40000 element and 160000 node buckets. The results are presented in Tables 5–7. These results suggest that all the processes and especially the enrichment are sensitive to the hash table size.

Preliminary three-dimensional results are presented in Table 8. Time taken to adapt and repartition are presented for different mesh sizes. Data migration times appear to be bounded as the problem size and number of processors goes up. Refinement and enrichment times are also quite small for the problem sizes and are a very small fraction of solution times. Note that the number of elements migrated appears to be very large since these results are for a uniform refinement of the whole mesh, i.e., the worst case.

Further, we wish to emphasize here that the total time spent in different data management operations in our structure is typically smaller than the time spent in the solution process. Table 9 shows the time spent on the solver outlined in the previous section on a 2D test problem for different maximum polynomial orders on different numbers of processors of an IBM SP. In another test, the systems arising from the 3D mesh shown in Figure 11 took 180.53 seconds on 16 processors for 90453 unknowns.

Table 5. Effect of the hash table size on the refinement, on 16 processors of the Cray T3E.

Refinement	Cray T3E			
Elements Refined	B-tree	Hashing-1	Hashing-$\frac{1}{2}$	Hashing-$\frac{1}{4}$
1046	0.222 s	0.305 s	0.155 s	0.154 s
4184	0.819 s	0.632 s	0.514 s	0.523 s
16736	4.376 s	2.984 s	3.263 s	3.719 s
66944	36.922 s	30.363 s	37.37 s	46.425 s

Table 6. Effect of the hash table size on the enrichment, on 16 processors of the Cray T3E.

Enrichment	Cray T3E			
Elements Enriched	B-tree	Hashing-1	Hashing-$\frac{1}{2}$	Hashing-$\frac{1}{4}$
4184	0.121 s	0.063 s	0.043 s	0.034 s
16736	0.291 s	0.118 s	0.127 s	0.137 s
66944	1.278 s	0.486 s	0.616 s	0.923 s
267776	5.677 s	3.680 s	6.571 s	13.844 s

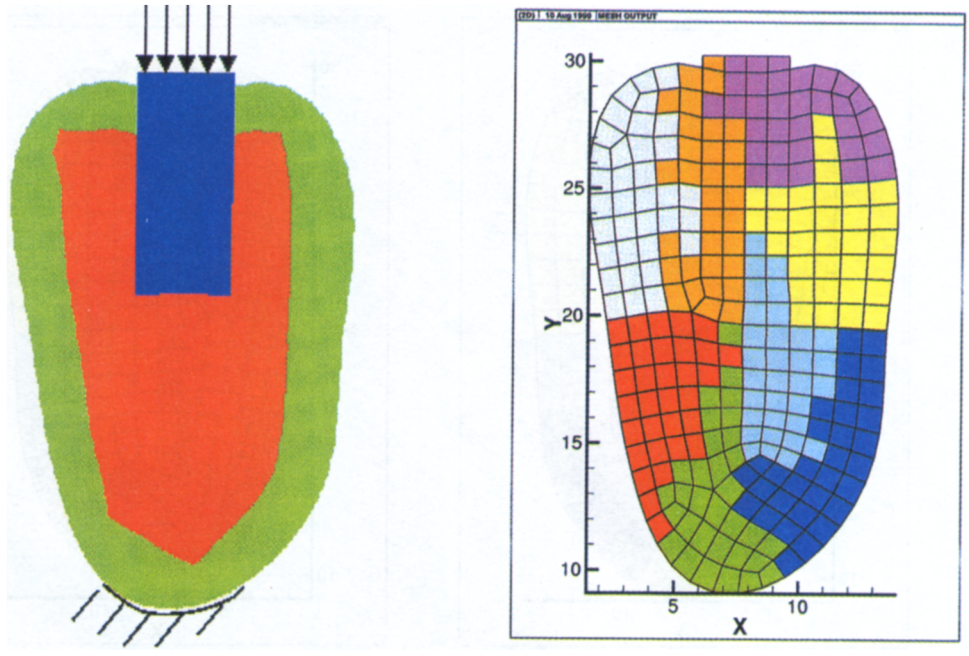
Table 7. Effect of the hash table size on the dynamic load balancing, on 16 processors of the Cray T3E.

Load Balancing	Cray T3E			
Elements Moved	B-tree	Hashing-1	Hashing-$\frac{1}{2}$	Hashing-$\frac{1}{4}$
648	0.372 s	0.515 s	0.359 s	0.283 s
2872	1.530 s	3.122 s	1.936 s	1.377 s
10608	6.885 s	14.694 s	9.722 s	7.518 s
39218	67.223 s	84.596 s	63.589 s	66.655 s

Finally, it should be concluded that our current implementation of the two schemes gives comparable results with better overall performance on the SGI. It should be also noted that the data structure operations of both schemes occupy a very small part of the total simulation time, especially when compared to the time taken by the solver algorithm. Hashing can be tuned for efficiency but in the case of adaptive techniques, this is very difficult. Since at the beginning of the simulation, the number of objects in the final mesh is not known, extendible hashing or prediction is necessary. Furthermore, since the address locator is based on the two extreme keys in the domain, the objects of the subdomains occupy only a small fraction of their table. On the other hand, the use of the global extremes is necessary because load balancing continuously changes the local key space. Finally, because adaptive techniques refine the mesh at certain locations of interest, the uneven distribution of the keys in the key space leads to poor efficiency. However, frequent runtime rearrangement of the table in order to resolve these problems and deliver constant efficiency can be expensive. As a result, future work should be directed towards further improving the efficiency of the B-tree, that needs much less tuning and has constant, repeatable performance.

7.3. Load Balancing

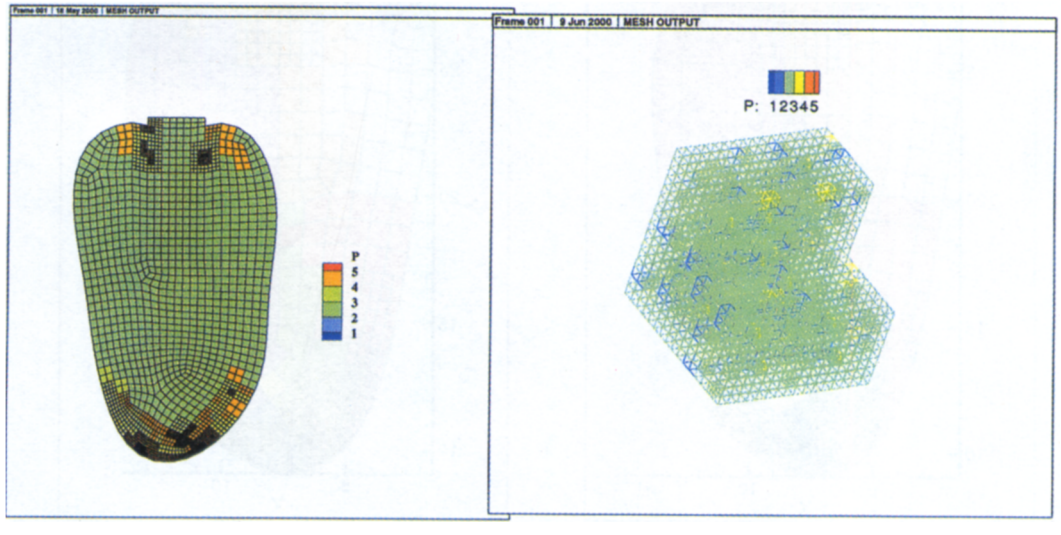
Figures 10–13 demonstrate the dynamic load balancing of the partitions as the computation and the adaptation evolve. The model of the dental implant problem is defined in part (a) of Figure 10. Different colors of this plot indicate the three different materials of the model: the dental implant, the hard outer shell of the bone, and the soft trabecular bone inside. The initial



(a) The model.

(b) Partitioned initial uniform mesh of the model (hli = 7.5%).

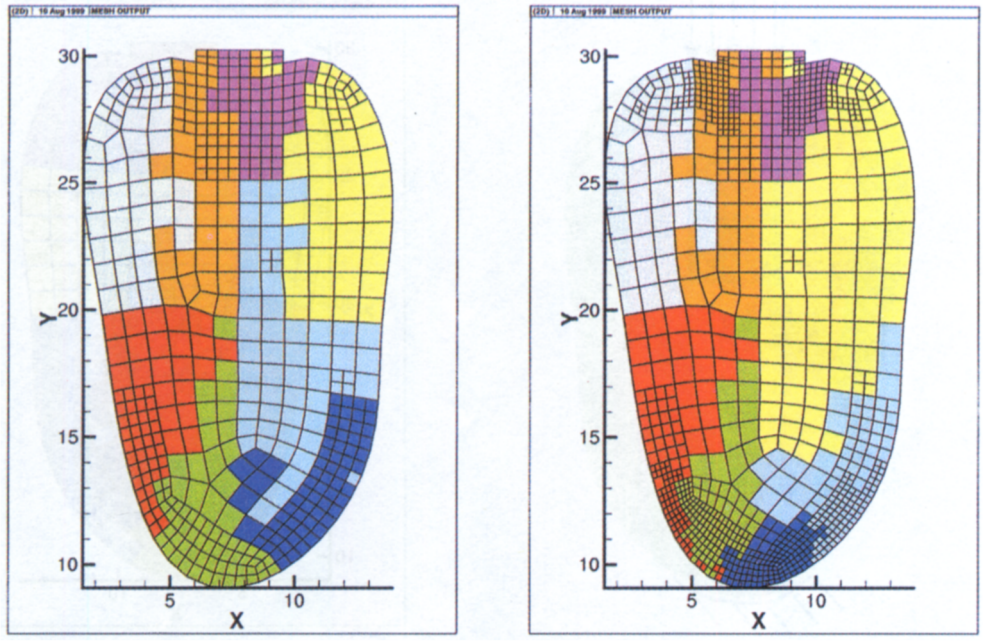
Figure 10. The model and its initial mesh.



(a) Final *hp* mesh for 2D bone-implant model.

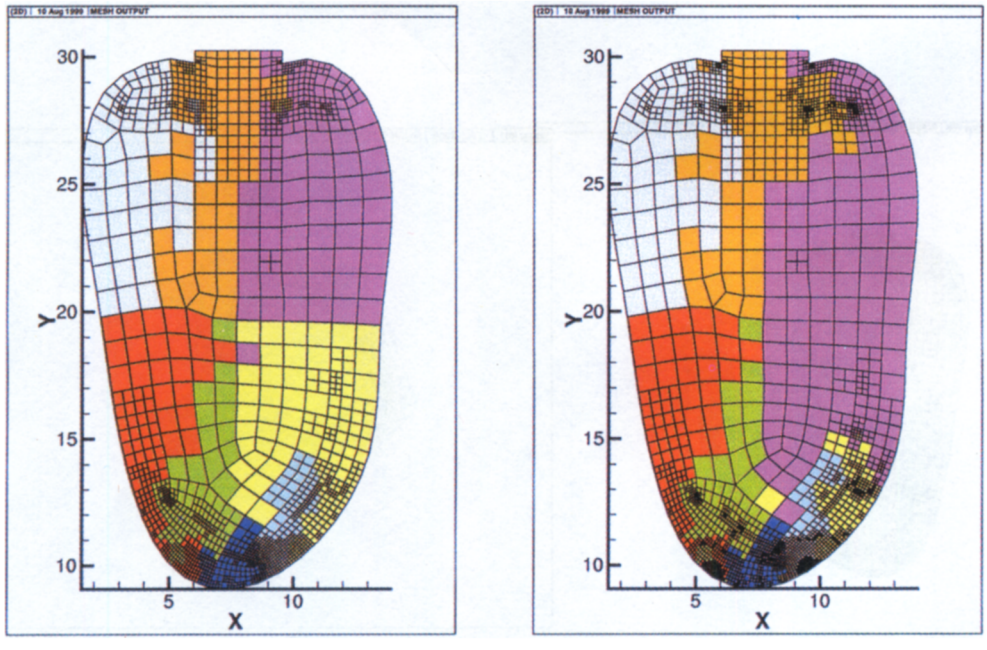
(b) Final *hp* mesh for 3D "L-shape" problem.

Figure 11. Adapted *hp* meshes.



(a) After the first stage (hli = 12%). (b) After the second stage (hli = 15%).

Figure 12. After adaptation and redistribution—I.



(a) After the third stage (hli = 22%). (b) After the fourth stage (hli = 13.5%).

Figure 13. After adaptation and redistribution—II.

Table 8. Time taken to adapt and repartition on a 3D problem on four, eight, and 16 processors of the SGI Origin 2000.

	Adaptation			Repartition	
No. Proc.	Elements Adapted	Refine	Enrich	Elements Moved	Time
4	384	0.64	0.11	384	0.11
4	3072	0.36	0.16	11108	11.99
8	384	0.26	0.09		
8	3072	2.68	0.78	19783	8.22
8	24576	78.44	3.54		
8	196608		36.03		
16	3072	2.01	0.12	16427	7.33
16	24576	79.28	0.61		

Table 9. Time in seconds to perform different solver operations on 2D test problem on different numbers of processors of the IBM SP.

Pol.	4pr		8pr		16pr		32pr		64pr	
Ord.	dof	Time	dof	Time	dof	Time	dof	Time	dof	Time
1	902	0.17	1740	0.42	3402	0.72	13202	3.29	26106	9.54
2	3402	0.38	6670	0.99	13202	1.81	52002	9.27	103282	22.8
3	7502	0.96	14792	1.98	29402	3.22	116402	11.0	231530	27.4
4	13202	2.24	26106	4.0	52002	6.18	206402	15.8	410850	31.2
5	20502	4.81	40612	7.82	81002	11.7	322002	23.2	641242	45.0
6	29402	10.3	58310	15.2	116402	22.6	463202	40.4	922706	63.4

mesh is partitioned to eight subdomains shown in the second figure. The computational load imbalance of a subdomain mesh is measured by the absolute value of the load imbalance factor, given by the difference between the ideal load of a subdomain and the actual load normalized by the ideal load

$$\text{li} = \frac{\text{ideal subdomain load} - \text{actual subdomain load}}{\text{ideal subdomain load}}.$$

The highest load imbalance factor of the mesh is denoted by hli.

The changing size and shape of the subdomains in Figures 12 and 13 indicate element movements between the processors aiming to balance the computational load represented by the number of unknowns within the partitions. It is observable how the size of the subdomains containing small, refined elements shrinks, while the ones with the bigger element sizes grow. The incremental property of the SFC-based load balancing is revealed by the fact that the color scheme does not change dramatically between the different stages. It should be also noted that even though the subdomains are load balanced well, the quality of the partitioning is not too high. This is indicated by the relatively large interface area due to the irregular geometry of the problem domain. This may be remedied by using graph-based partitioners. However, if, as in our case, the data storage itself is based on the space-filling curve ordering, the contiguous property of the local key spaces of the subdomains will be lost.

8. CONCLUSIONS

We have described here the design and implementation of an integrated set of data management and load-balancing tools necessary for the development of efficient parallel adaptive *hp* finite-element codes. Our development relies on the use of a locality preserving space-filling curve ordering to index the element/node data. We have illustrated the use of this ordering to create a

virtual shared memory by providing easily available globally accessible keys. Local tree and hash tables complete the data management schemes. Both trees and tables show good performance as evidenced by extensive numerical tests over wide ranges of h and p refinements in both two- and three-dimensional grids. However, the hash table needs problem-specific tuning parameters, and hence, is not as robust as the B-tree. Several dynamic load-balancing schemes designed to obtain good load balance while minimizing data migration have been presented. The data management schemes have been tightly integrated to both specialized solution algorithms and general purpose PETSC solver library. Good performance has been obtained in all cases. Furthermore, the tools can be easily customized to develop new application codes. All code used in this paper is available for download from `http://wings.buffalo.edu/eng/mae/acm2e/`.

REFERENCES

1. W. Gui and I. Babuska, The h, p and hp versions of the finite element method, Part I, *Numer. Math.* **49**, 571–612, (1986).
2. W. Gui and I. Babuska, The h, p and hp versions of the finite element method, Part II, *Numer. Math.* **49**, 613–657, (1986).
3. W. Gui and I. Babuska, The h, p and hp versions of the finite element method, Part III, *Numer. Math.* **49**, 659–683, (1986).
4. L. Demkowicz, J.T. Oden, W. Rachowicz and O. Hardy, Towards a universal hp adaptive finite element strategy, Part 1, Constrained approximation and data structure, *Comput. Methods Appl. Mech. Engrg.* **77**, 79–112, (1989).
5. A. Patra and J.T. Oden, Computational techniques for adaptive hp finite element methods, *Finite Elements in Analysis and Design* **25**, 27–39, (1997).
6. M.J. Berger and J. Saltzman, Structured adaptive mesh refinement on the connection machine, In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, (March 1993).
7. C. Chang, A. Sussman and J. Saltz, Support for distributed dynamic data structures in C++, Technical Report CS-TR-3416 and UMIACS-TR-95-19, Department of Computer Science, University of Maryland, (1995).
8. S.J. Fink, S.R. Kohn and S.B. Baden, Efficient run-time support for irregular block-structured applications, *J. Parallel and Distributed Computing* **50** (1/2), 61–82, (April/May 1998).
9. M. Parashar and J.C. Browne, Distributed dynamic data-structures for parallel adaptive mesh refinement, Available from: `http://www.caip.rutgers.edu/~parashar/DAGH/`, (December 1995).
10. H.C. Edwards and J.C. Browne, *Scalable Distributed Dynamic Array (SDDA) and Its Application to a Distributed Adaptive Mesh Data Structure*, Available from: `www.ticam.utexas.edu/~carter/sdda.html`, (January 1996).
11. M. Parashar, J.C. Browne and K. Klimkowski, A common data management infrastructure for adaptive algorithms for PDE solutions, SuperComputing97, Technical Paper.
12. K. Devine and J.E. Flaherty, Parallel adaptive hp refinement techniques for conservation laws, *Appl. Numer. Math.* **20**, 367–386, (1996).
13. J.E. Flaherty, R. Loy, M. Shephard, B. Szymansky, J. Teresco and L. Ziantz, Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws, *J. Parallel Distrib. Comput.* **47**, 139–152, (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. Maths.* **26**, 241–263, (1998).
15. J.E. Flaherty, R.M. Loy, M.S. Shephard, M.L. Simone, B.K. Szymansky, J.D. Teresco and L.H. Ziantz, Distributed octree data structures and local refinement method for the parallel solution of three-dimensional conservation laws, In *Grid Generation and Adaptive Algorithms*, (Edited by M.W. Bern, J.E. Flaherty and M. Luskin), Springer, (1999).
16. R.L. Kruse and A.J. Ryba, *Data Structures and Program Design in C++*, Prentice Hall, (1999).
17. H. Sagan, *Space Filling Curves*, Springer, Berlin, (1994).
18. A. Patra and J.T. Oden, Problem decomposition strategies for adaptive hp finite element methods, *Computing Systems in Engineering* **6** (2), (1995).
19. C. Gotsman and M. Lindenbaum, On the metric properties of discrete space filling curves, *IEE Transactions on Image Processing* **5** (5), (May 1996).
20. A. Perez, S. Kamata and E. Kawaguchi, Peano scanning of arbitrary size images, In *Proc. Int. Conf. Patt. Recogn.*, pp. 565–568, (1992).
21. J. Dugundji, *Topology*, Allyn and Bacon, Boston, (1966).
22. S. Sengupta and C.P. Korobkin, *C++ Object Oriented Data Structures*, Springer-Verlag, (1996).
23. J.R. Pilkington and S.B. Baden, Partitioning with space filling curves, CSE Technical Report, Department of Computer Science and Engineering, University of California, San Diego, (1994).

24. R. Biswas and L. Oliker, Experiments with repartitioning and load balancing adaptive meshes, In *Grid Generation and Adaptive Algorithms*, (Edited by M.W. Bern, J.E. Flahery and M. Luskin), Springer, (1999).,
25. K. Schloegel, G. Karypis and V. Kumar, Multilevel diffusion algorithms for repartitioning of adaptive meshes, *Journal of Parallel and Distributed Computing* **47**, 109–124, (1997).
26. A. Patra and D.W. Kim, Efficient mesh partitioning for adaptive *hp* finite element methods, In *Proceedings of the XIth Domain Decomposition Conference*, Greenwich, U.K., July 1998, (submitted).
27. A. Patra and A. Gupta, A strategy for adaptive *hp* mesh modification using non-linear programming, *Comp. Meth. App. Mech. and Eng.* (to appear).
28. I. Babuska, A. Craig, J. Mandel and J. Pitkaranta, Efficient preconditioning for the *p* version finite element method in two dimensions, *SIAM J. Numer. Anal.* **28** (3), 624–661, (1991).
29. B.Q. Guo and W. Cao, Domain decomposition method for the *h-p* version finite element method, *Comp. Meth. Appl. Mech. Engrg.* **157**, 425–440, (1998).
30. M. Ainsworth, A preconditioner based on domain decomposition of *hp* finite element approximation on quasi-uniform meshes, Mathematical and Computer Science Technical Reports, No. 16, University of Leicester, (1993).
31. L.F. Pavarino and O. Widlund, Iterative substructuring methods for spectral element discretizations of elliptic systems I. Compressible linear elasticity, *SIAM J. Numer. Anal.* **37** (6), 353–374, (1999).
32. J.T. Oden, A. Patra and Y.S. Feng, Domain decomposition solvers for adaptive *hp* finite element methods, *SIAM J. Numer. Anal.* **34** (6), 2090–2118, (1997).
33. S. Balay, W.D. Gropp, L. Curfman McInnes and B.F. Smith, *PETSc 2.0 Users Manual*, ANL-95/11 - Revision 2.0.28, Argonne National Laboratory, (2000).
34. A. Patra, A. Laszloffy and J. Long, AFEAPI: Adaptive finite elements application interface, In *Proceedings of the 9th SIAM Conference of Parallel Processing for Scientific Computing*, (March 1999).
35. G.W. Zumbusch, Dynamic load balancing in a lightweight adaptive parallel multigrid PDE solver, In *Proceedings of the 9th SIAM Conference of Parallel Processing for Scientific Computing*, (March 1999).
36. W.Y. Crutchfield and M.L. Welcome, Object oriented implementation of adaptive mesh refinement algorithms, *Scientific Programming* **2**, 145–156, (Winter 1993).
37. B. Hendrickson and K. Devine, Dynamic load balancing in computational mechanics, *Comp. Meth. Applied Mechanics and Engineering* (to appear).
38. A. Pothen, H. Simon and K. Liou, Partitioning sparse matrices with eigenvectors of graphs, *SIAM J. Matrix Anal.* **11**, 430–452, (1990).