SC97 • Technical Paper

TOC Authors Sessions Abstracts PostScript

A Common Data Management Infrastructure for Adaptive Algorithms for PDE Solutions

Manish Parashar
Department of Computer Sciences and
Texas Institute for Computational and Applied Mathematics
University of Texas at Austin
2.400 Taylor Hall
Austin, TX 78712
Tel: (512) 471-3312; Fax: (512) 471-8694
parashar@cs.utexas.edu
http://www.ticam.utexas.edu/~parashar/public_html/

James C. Browne
Department of Computer Sciences and
Texas Institute for Computational and Applied Mathematics
University of Texas at Austin
2.400 Taylor Hall
Austin, TX 78712
Tel: (512) 471-9579; Fax: (512) 471-8885
browne@cs.utexas.edu
http://www.cs.utexas.edu/users/browne/

Carter Edwards
Texas Institute for Computational and Applied Mathematics
University of Texas at Austin
2.400 Taylor Hall
Austin, TX 78712
Tel: (512) 471-3312; Fax: (512) 471-8694
carter@ticam.utexas.edu
http://www.ticam.utexas.edu/~carter/

Kenneth Klimkowski
Texas Institute for Computational and Applied Mathematics
University of Texas at Austin

2.400 Taylor Hall
Austin, TX 78712
Tel: (512) 471-3312; Fax: (512) 471-8694
`ken@ticam.utexas.edu`
`http://www.ticam.utexas.edu/~ken/`

Abstract:

This paper presents the design, development and application of a computational infrastructure to support the implementation of parallel adaptive algorithms for the solution of sets of partial differential equations. The infrastructure is separated into multiple layers of abstraction. This paper is primarily concerned with the two lowest layersof this infrastructure: a layer which defines and implements dynamic distributed arrays (DDA), and a layer in which several dynamic data and programming abstractions are implemented in terms of the DDAs. The currently implemented abstractions are those needed for formulation of hierarchical adaptive finite difference methods, hp-adaptive finite element methods, and fast multipole method for solution of linear systems. Implementation of sample applications based on each of these methods are described and implementation issues and performance measurements are presented.

Keywords:

Problem Solving Environment, Parallel Adaptive Algorithm, Distributed Dynamic Data Structures, Adaptive Mesh-Refinement, hp-Adaptive Finite Elements, Fast Multipole Methods.

1 Introduction

This paper describes the design and implementation of a common computational infrastructure to support parallel adaptive solutions of partial differential equations. The motivations for this research are:

1. Adaptive methods will be utilized for the solution of almost all very large-scale scientific and engineering models. These adaptive methods will be executed on large-scale heterogeneous parallel execution environments.
2. Effective application of these complex methods on scalable parallel architectures will be possible only through the use of programming abstractions which lower the complexity of application structures to a tractable level.
3. A common infrastructure for this family of algorithms will result in both, enormous savings in coding effort and a more effective infrastructure due to pooling and

focusing of effort.

The goal for this research is to reduce the intrinsic complexity of coding parallel adaptive algorithms by providing an appropriate set of data structures and programming abstractions. This infrastructure has been developed as a result of collaborative research among computer scientists, computational scientists and application domain specialists working on three different projects: An DARPA project for hp-adaptive computational fluid dynamics and two NSF sponsored Grand Challenge projects, one on numerical relativity and the other on composite materials.

1.1 Conceptual Framework

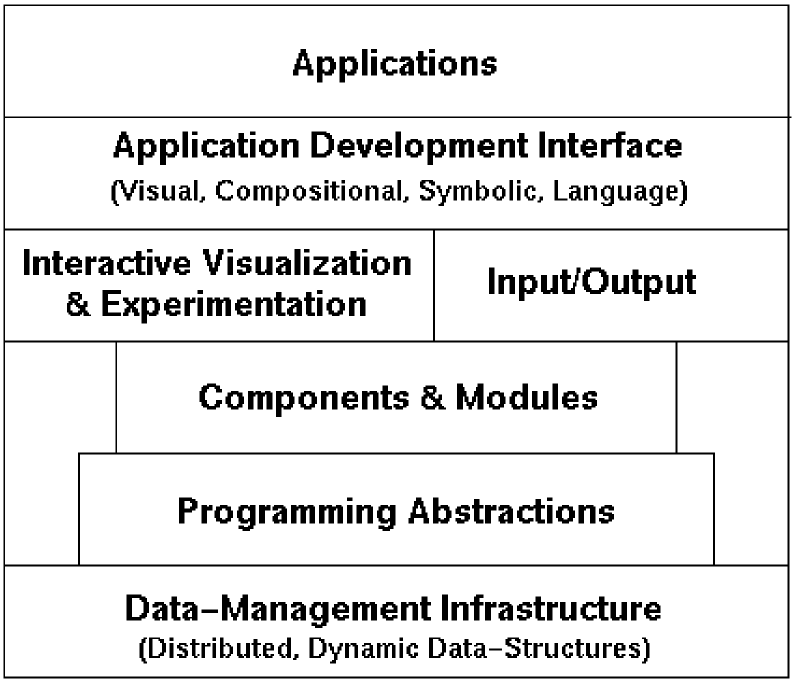


Figure 1.1: Hierarchical Problem Solving Environment for Parallel Adaptive Algorithms for the Solution of PDEs

Figure 1 is a schematic of our perception of the structure of a problem solving environment (PSE) for parallel adaptive techniques for the solution of partial differential equations. This paper is primarily concerned with the lowest two layers of this hierarchy and how these layers can support implementation of higher levels of abstraction. The bottom layer of the hierarchical PSE is a data-management layer. The layer implements a Distributed Dynamic Array (DDA) which provides array access semantics to distributed and dynamic data. The next layer is a programming abstractions layer which adds application semantics to DDA objects. This layer implements data abstractions such as grids, meshes and trees which underlie different solution methods. The design of the PSE is based on a separation of concerns and the definition of hierarchical abstractions based on the separation. Such a clean separation of concerns [1] is critical to the success of an infrastructure that can provide a foundation for several different solution methods. In particular the PSE presented in this paper supports finite difference methods based on adaptive mesh refinement, hp-adaptive finite element methods, and adaptive fast multipole methods.

1.2 Overview

This paper defines the common requirements of parallel adaptive finite difference and finite element methods for solution of PDEs and fast multipole solution of linear systems, and demonstrates that one data management system based on Distributed Dynamic Arrays (DDA) can efficiently meet these common requirements. The paper then describes the design concepts underlying DDAs and sketches implementations of parallel adaptive finite difference, finite element, multipole solutions using DDAs based on a common conceptual basis as the common data management system. Performance evaluations for each method are also presented.

The primary distinctions between the DDA-based data management infrastructures and other packages supporting adaptive methods are (1) the separation of data management and solution method semantics and (2) the separation of addressing and storage semantics in the DDA design. This separation of concerns enables the preservation of application locality in multi-dimensional space when it is mapped to the distributed one-dimensional space of the computer memory and the efficient implementation of dynamic behavior.

The data structure which has traditionally been used for implementation of these problems has been the multi-dimensional array. Informally an array consists of: (1) an index set (a lattice of points in an n-dimensional discrete space), (2) a mapping from the n-dimensional index set to one dimensional storage, and (3) a mechanism for accessing storage associated with indices.

A DDA is a generalization of the traditional array which targets the requirements of adaptive algorithms. In contrast to regular arrays, the DDA utilizes a recursively defined hierarchical index space where each index in the index space may be an index space. The storage scheme then associates contiguous storage with spans of this index space. The relationship between the application and the array is defined by deriving the index space directly from the n-dimensional physical domain of the application.

2 Problem Description

Adaptive algorithms require definition of operators on complex dynamic data structures. Two problems arise: (1) the volume and complexity of the bookkeeping code required to construct and maintain these data structures overwhelms the actual computations and (2) maintaining access locality under dynamic expansion and contraction of the data requires complex copying operations if standard storage layouts are used. Implementation on parallel and distributed execution environments adds the additional complexities of partitioning, distribution and communication. Application domain scientists and engineers are forced to create complex data management capabilities which are far removed from the application domain. Further, standard parallel programming languages do not provide explicit support for dynamic distributed data structures. Data management requirements for the three different adaptive algorithms for PDEs are described below.

2.1 Adaptive Finite Difference Data Management Requirements

Finite difference methods approximate the solution of the PDE on a discretized grid overlaid on the n-dimensional physical application domain. Adaptation increases the resolution of the discretization in required regions by refining segments of the grid into finer grids. The computational operations on the grid may include local stencil based operations at all levels of resolution, transfer operations between levels of resolution and global linear solves. Thus the requirement for the data-management system for adaptive finite difference methods is seamless support of these operations across distribution and refining and coarsening of the grid. Storage of the dynamic grid as an array where each point of the grid is mapped to a point in a hierarchical index space is natural. Then the refinements and coarsening of the grid become traversal of the hierarchy of the hierarchical index space.

2.2 Adaptive Finite Element Data Management Requirement

The finite element method requires storage of the geometric information defining the mesh of elements which spans the application domain. Elements of the linear system arising from finite element solutions are generally computed *on the fly* so that they need not be stored. HP-adaptive finite element methods adapt by partitioning elements into smaller elements (h refinement) or by increasing the order of the polynomial approximating the solution on the element (p refinement). Partitioning adds new elements and changes relationships among elements. Changing the approximation function enlarges the descriptions of the elements. The data-management requirements for hp-adaptive finite elements thus include storage of dynamic numbers of elements of dynamic sizes. These requirements can be met by mapping each element of the mesh to a position in a hierarchical index space which is associated with a dynamic span of the one-dimensional storage space. Partitioning an element replaces a position in the index space by a local index space representing the expanded mesh.

2.3 Adaptive Fast Multipole Data Management Requirements

Fast multipole methods partition physical space into subdomains. The stored representation of the subdomain includes a *charge configuration* of the subdomain and various other descriptive data. Adaptation consists of selectively partitioning subdomains. The elements are often generated *on the fly* so that storage of the values of the forces and potential need not be stored. The requirement for data management capability is therefore similar to that of adaptive finite element methods. A natural mapping is to associate each subdomain with a point in a hierarchical index space.

2.4 Requirements Summary

It should be clear that an extended definition of an array where each element can itself be an array and where the entity associated with each index position of the array can be an object of arbitrary and variable size provides one natural representation of the data management requirements for all of adaptive finite element, adaptive finite difference and adaptive fast multipole solvers. The challenge is now to demonstrate an efficient implementation of such an array and to define implementations of each method in terms of this storage abstraction. The further and more difficult challenge is an efficient parallel/distributed implementation of such an array.

3 Distributed Dynamic Data-Management

The distributed dynamic data-management layer of the PSE implements Distributed Dynamic Arrays (DDAs). This layer provides pure array access semantics to dynamically structured and physically distributed data. DDA objects encapsulate distribution, dynamic load-balancing, communications, and consistency management, and have been extended with visualization and analysis capabilities.

There are currently two different implementations of the data management layer, both built on the same DDA conceptual framework: (1) the Hierarchical Dynamic Distributed Array (HDDA) and (2) the Scalable Dynamic Distributed Array (SDDA). The HDDA is a hierarchical array in that each element of the array can recursively be an array; it is a dynamic array in that each array (at each level of the hierarchy) can expand and contract at run-time. Instead of hierarchical arrays the SDDA implements a distributed dynamic array of objects of arbitrary and heterogeneous types. This differentiation results from the differences in data management requirements between structured and unstructured meshes. For unstructured meshes it is more convenient to incorporate hierarchy information into the programming abstraction layer which implements the unstructured mesh. These two implementations derived from a common base implementation and can and will be re-integrated into a single implementation in the near future.

The arrays of objects defined in the lowest layer are specialized by the higher layers of the PSE to implement application objects such as grids, meshes, and tree. A key feature of a DDA based on the conceptual framework described following is its ability to extract out the data locality requirements from the application domain and maintain this locality despite its distribution and dynamic structure. This is achieved through the application of the principle of separation of concerns [1] to the DDA design. An overview of this design is shown in Figure 3.1. Distributed dynamic arrays are defined in the following subsection.

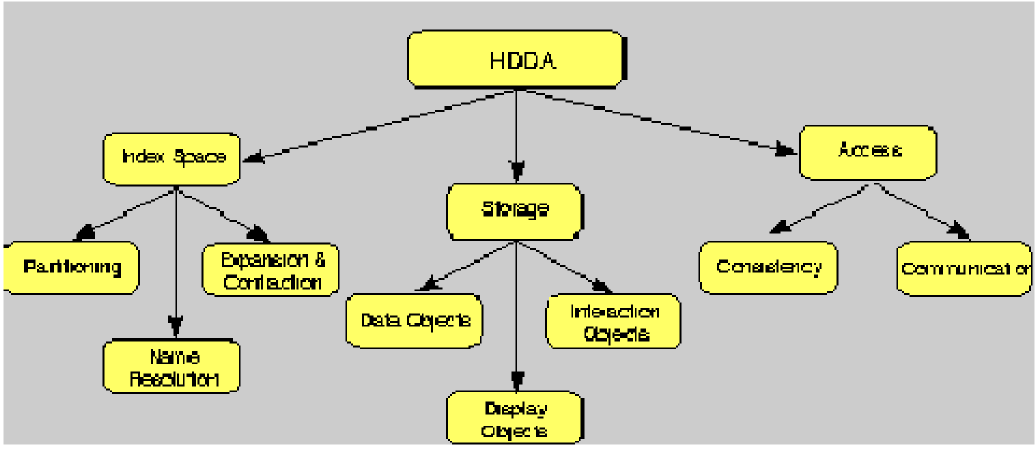


Figure 3.1: DDA Design - Separation of Concerns -> Hierarchical Abstractions

3.1 Distributed Dynamic Array Abstraction

The SDDA and HDDA are implementations of a **distributed dynamic array**. The distributed dynamic array abstraction, presented in detail in [2], is summarized as follows.

- In general, an **array** is defined by a **data set** *D*, an **index space** *I*, and an injective function $F : D \rightarrow I$
- An array is **dynamic** if elements may be dynamically inserted into or removed from its data set *D*.
- An array is **distributed** if the elements of its data set *D* are distributed.

A data set *D* is simply a finite set of data objects. An index space *I* is a countable set of **indices** with a well-defined linear ordering relation, for example the set of natural numbers. Two critical points of this array abstraction are (1) that the cardinality of the data set *D* is necessarily equal to or less than the cardinality of the index space *I* and (2) that each element of the data set uniquely maps to an element of the index space.

3.2 Hierarchical Index Space and Space-Filling Curves

An application partitions its *N*-dimensional problem domain into a finite number of points and/or regions. Each region can be associated with a unique coordinate in the problem domain. Thus the "natural" indexing scheme for such an application is a discretization of these coordinates. Such an index space can defined as: $I_1 \times I_2 \times \cdots \times I_N$ where each I_j corresponds to the discretization of a coordinate axis.

An index space may be hierarchical if the level of discretization is allowed to vary, perhaps in correspondence with a hierarchical partitioning of the problem domain. The I_j components of hierarchical index space could be defined as $(d,(i_1,i_2,\ldots,i_d))$, where d is the depth of the index.

Recall that an index space requires a well-defined linear ordering relation. Thus the "natural" *N* dimensional hierarchical index space must effectively be mapped to a linear, or one-dimensional, index space. An efficient family of such maps are defined by space-filling curves (SFC) [3].

One such mapping is defined by the Hilbert space-filling curve, illustrated in Figure 3.2. In this mapping a bounded domain is hierarchically partitioned into regions where the regions are given a particular ordering. In theory the partitioning "depth" may be infinite and so any finite set of points in the domain may be fully ordered. Thus an SFC mapping defines a hierarchical index space, of theoretically infinite depth, for any application domain which can be mapped into the SFC "bounding box".

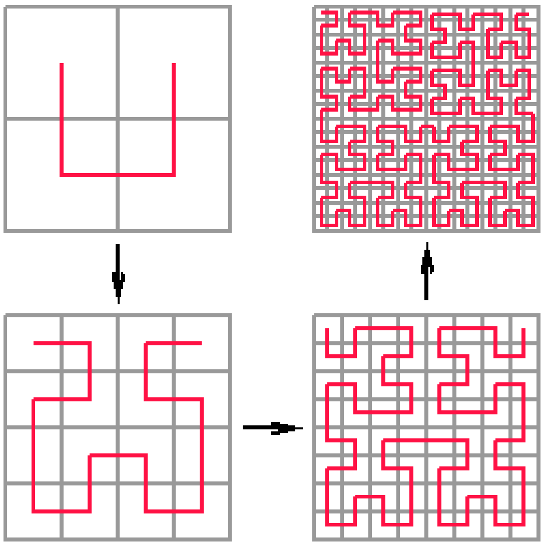


Figure 3.2: Hilbert Space-Filling Curve

The "natural" index space of the SFC map efficiently defines a linear ordering for all points and/or subregions of the application's problem domain. Given such a linear ordering the these subregions can be distributed among processors by simply partitioning the index space. This partitioning is easily and efficiently obtained from a partitioning the linearly ordered index space such that the computational load of each partition is roughly equal. Figure 3.3, Space-Filling Curve Partitioning, illustrates this process for an irregularly partitioned two dimensional problem domain. Note in Figure 3.3 that the locality preserving property of the Hilbert SFC map generates "well-connected" subdomains.

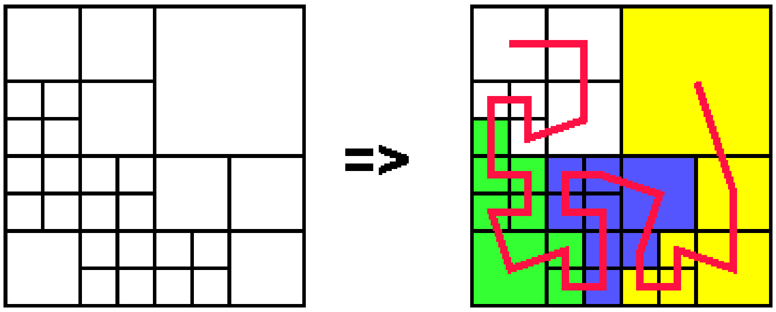


Figure 3.3: Space-Filling Curve Partitioning

3.3 DDA Implementation

An **array implementation** provides storage for objects of the data set *D* and access to these objects through a converse function $F^{-1} : F(D) \rightarrow D$. The quality and efficiency of an array implementation are determined by the correlation between storage locality and index locality and by the expense of the converse function F^{-1}. For example a conventional one-dimensional FORTRAN array has both maximal quality and efficiency.

A DDA implements distributed dynamic arrays where storage for objects in the data set *D* is distributed and dynamic, and the converse function F^{-1} provides global access to these objects. A DDA's storage structure and converse function consists of two components: (1) local object storage and access and (2) object distribution. The HDDA and SDDA implementations of a DDA use extendible hashing [4] & [5] and red-black balanced binary trees [6] respectively for local object storage and access.

An application instructs the DDA as to how to distribute data by defining a partitioning of index space *I* among processors. Each index in the index space is uniquely assigned to a particular processor $i \rightarrow P$. The storage location of a particular data object is now determined by its associated index $d \rightarrow i \rightarrow P$. Thus the storage location of any dynamically created data object is well-defined.

Each DDA provides global access to distributed objects by transparently caching objects between processors. For example, when an application applies a DDA's converse function ($F^{-1}(i) \rightarrow d$) if the data object d is not present on the local processor the data object is transparently copied from its owning processor into a cache on the local processor.

3.4 Locality, Locality, Locality!

A DDA's object distribution preserves locality between object storage and the object's global indices. Given the "natural" index space of the space-filling curve map and the corresponding domain partitioning, a DDA's storage locality is well-correlated with geometric locality, as illustrated in Figure 3.4.

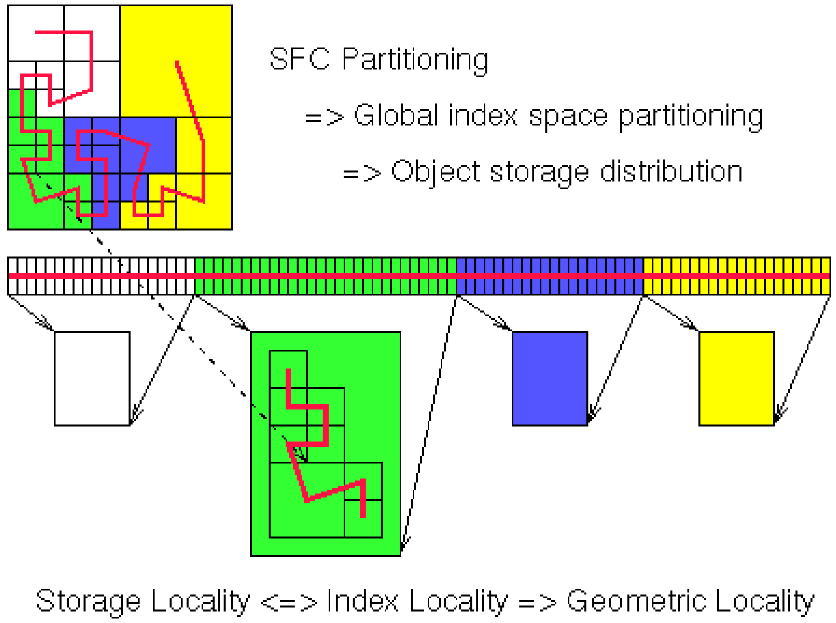


Figure 3.4: Locality, Locality, Locality!

In Figure 3.4 an application assigns SFC indices to the data objects associated with each subregion of the domain. A DDA stores objects within a span of indices in the local memory of a specified processor. Thus geometrically local subregions have their associated data objects stored on the same processor.

3.5 Application Programming Interface (API)

The DDA application programming interface consists a small set of simple methods which hide the complexity of the storage structure and make transparent any required interprocessor communication. These methods include:

GET	$d \leftarrow F^{-1}(i)$
INSERT	$D \leftarrow D + d_{new}$
REMOVE	$D \leftarrow D - d_{old}$
ITERATE	$\{ d : i <= F(d) <= j \}$
REPARTITION	Forcing a redistribution of objects
PUT/LOCK/UNLOCK	Cache coherency controls

Implementation of these methods varies between the SDDA and HDDA; however, the abstractions for these methods are common to both versions of the DDA.

3.6 DDA Performance

The DDA provides object management services for an application's distributed dynamic data structures. This functionality introduces an additional overhead cost when accessing local objects. For example, local object access could be accomplished directly via 'C' pointers instead of DDA indices; however, note that the pointer to a data object may be invalidated under data object redistribution.

This overhead cost is measured for the SDDA version of the DDA, which uses a red-black balanced binary tree algorithm for local object management. The overhead cost of the local get, local insert, and local remove method is measured on an IBM RS6000. The local get method retrieves a local data object associated with an input index value, i.e. the converse function. The local insert method inserts a new data object associated with its specified index $(d_{new}, F(d_{new}))$ into the local storage structure. The local remove method removes a given data object from the local storage structure. The computational time of each method is measured given the size of the existing data structure, as presented in Figure 3.5, SDDA Overhead for Local Methods.

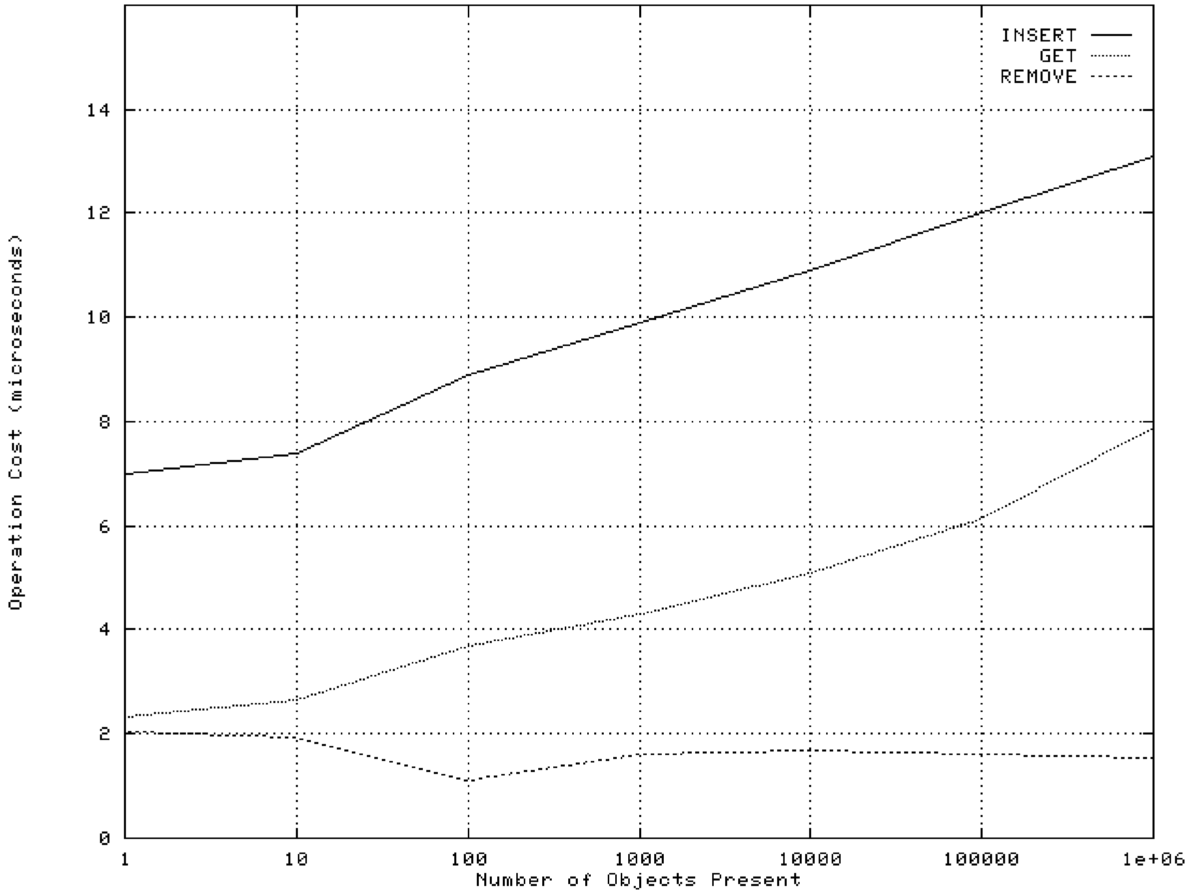


Figure 3.5: SDDA Overhead for Local Methods

The overhead cost of local get method, as denoted by the middle line in Figure 3.5, increases logarithmically from 2 microseconds for an empty SDDA to 8 microseconds for an SDDA containing one million local data objects. This slow logarithmic growth is as expected for a search into the SDDA's balanced binary tree. The local insert method performs two operations: (1) search for the proper point in the data structure to insert the object and (2) modification of the data structure for the new object. The cost of the insert method is a uniform "delta" cost over the get operation. Thus the overhead cost of modifying the data structure for the new data object independent of the size of the SDDA. Note that the overhead cost of the local remove method, as denoted in Figure 3.5 by the lowest line, is also independent of the size of the SDDA.

4 Method Specific Data & Programming Abstractions

The next level of the PSE specializes DDA objects with method specific semantics to create high-level programming abstractions which can be directly used to implement parallel adaptive algorithms. The design of such abstractions for three different classes of adaptive solution techniques for PDEs, hierarchical, dynamically adaptive grids, hp-adaptive finite elements and dynamic tree (dynamic trees are the data abstractions upon which the fast multipole methods are implemented) are descried below.

4.1 Hierarchical Adaptive Mesh-Refinement

Problem Description

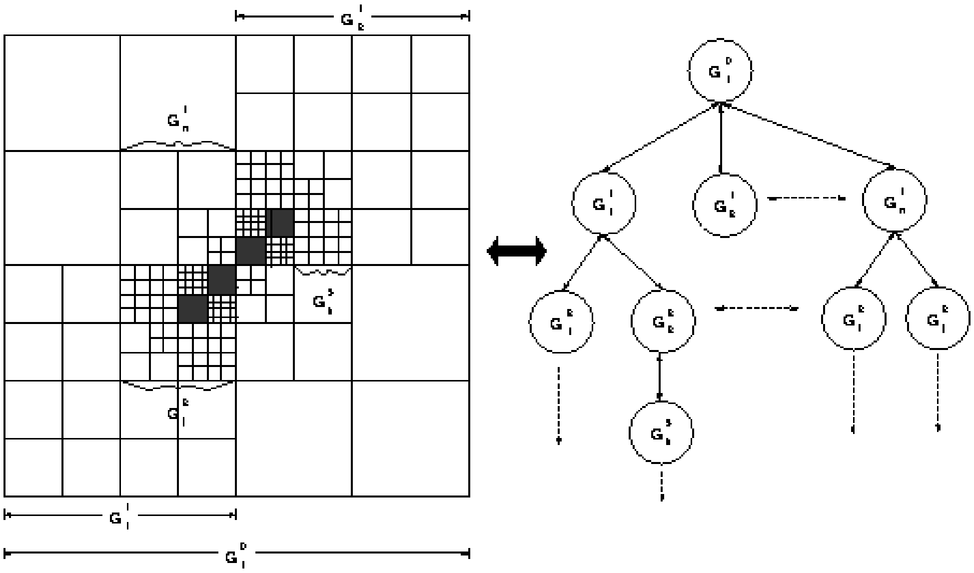
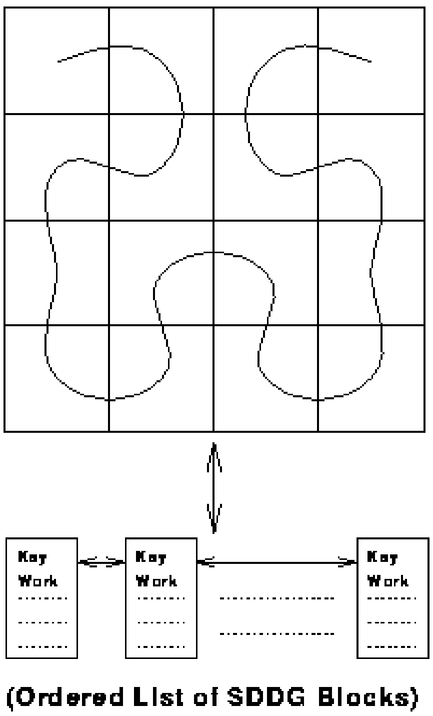


Figure 4.1: Adaptive Grid Hierarchy - 2D (Berger-Oliger AMR Scheme)

Dynamically adaptive numerical techniques for solving differential equations provide a means for concentrating computational effort to appropriate regions in the computational domain. In the case of hierarchical adaptive mesh refinement (AMR) methods, this is achieved by tracking regions in the domain that require additional resolution and dynamically overlaying finer grids over these regions. AMR-based techniques start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain requiring additional resolution are tagged and finer grids are overlayed on the tagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring more resolution are similarly tagged and even finer grids are overlayed on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The adaptive grid hierarchy corresponding to the AMR formulation by Berger & Oliger [7] is shown in Figure 4.1. **Distributed Data-Structures for Hierarchical AMR** Two basic distributed data-structures have been developed, using the fundamental abstractions provided by the HDDA, to support adaptive finite-difference techniques based on hierarchical AMR: (1) A Scalable Distributed Dynamic Grid (SDDG) which is a distributed and dynamic array, and is used to implement a single component grid in the adaptive grid hierarchy; and (2) A Distributed Adaptive Grid Hierarchy (DAGH) which is defined as a dynamic collection of SDDGs and implements the entire adaptive grid hierarchy. The SDDG/DAGH data-structure design is based on a linear representation of the hierarchical, multi-dimensional grid structure. This representation is generated using space-filling curves described in Section 3 and exploits the self-similar or recursive nature of these mappings to represent a hierarchical DAGH structure and to maintain locality across different levels of the hierarchy. Space-filling mapping functions are also used to encode information about the original multi-dimensional space into each space-filling index. Given an index, it is possible to obtain its position in the original multi-dimensional space, the shape of the region in the multi-dimensional space associated with the index, and the space-filling indices that are adjacent to it. A detailed description of the design of these data-structures can be found in [8].



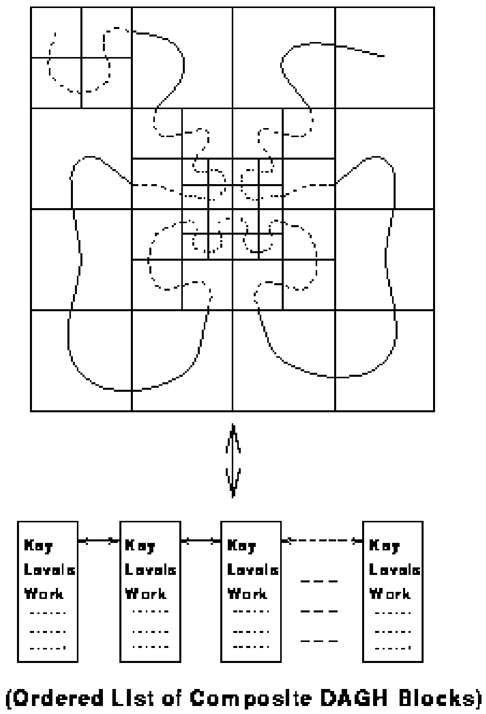


Figure 4.2: SDDG Representation - Figure 4.3: DAGH Composite Representation

SDDG Representation:

A multi-dimensional SDDG is represented as a one dimensional ordered list of SDDG blocks. The list is obtained by first blocking the SDDG to achieve the required granularity, and then ordering the SDDG blocks based on the selected space-filling curve. The granularity of SDDG blocks is system dependent and attempts to balance the computation-communication ratio for each block. Each block in the list is assigned a cost corresponding to its computational load. Figure 4.2 illustrates this representation for a 2-dimensional SDDG.

Partitioning a SDDG across processing elements using this representation consists of appropriately partitioning the SDDG block list so as to balance the total cost at each processor. Since space-filling curve mappings preserve spatial locality, the resulting distribution is comparable to traditional block distributions in terms of communication overheads.

DAGH Representation:

The DAGH representation starts with a simple SDDG list corresponding to the base grid of the grid hierarchy, and appropriately incorporates newly created SDDGs within this list as the base grid gets refined. The resulting structure is a composite list of the entire adaptive grid hierarchy. Incorporation of refined component grids into the base SDDG list is achieved by exploiting the recursive nature of space-filling mappings: For each refined region, the SDDG sub-list corresponding to the refined region is replaced by the child grid's SDDG list. The costs associated with blocks of the new list are updated to reflect combined computational loads of the parent and child. The DAGH representation therefore is a composite ordered list of DAGH blocks where each DAGH block represents a block of the entire grid hierarchy and may contain more than one grid level; i.e. inter-level locality is maintained within each DAGH block. Figure 4.3 illustrates the composite representation for a two dimensional grid hierarchy.

The AMR grid hierarchy can be partitioned across processors by appropriately partitioning the linear DAGH representation. In particular, partitioning the composite list to balance the cost associated to each processor results in a composite decomposition of the hierarchy. The key feature of this decomposition is that it minimizes potentially expensive inter-grid communications by maintaining inter-level locality in each partition.

Data-Structure Storage:

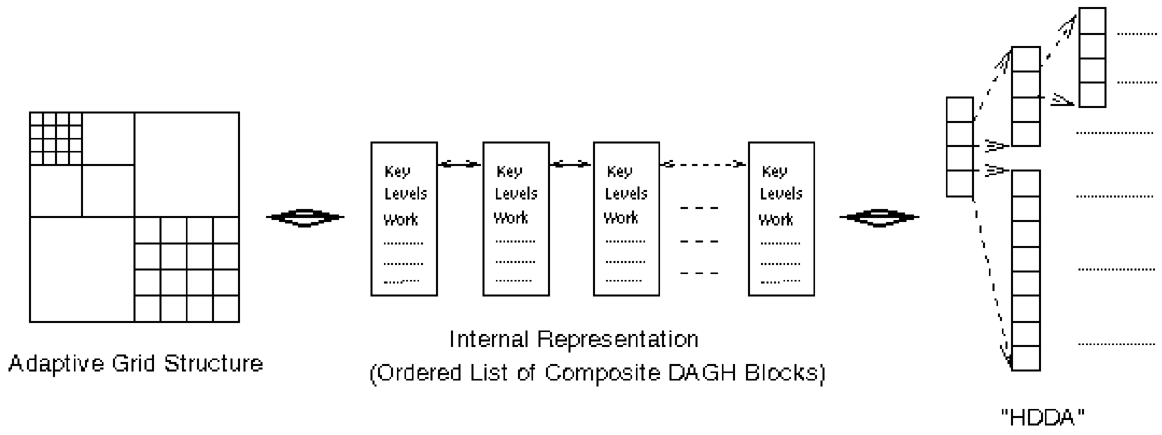


Figure 4.4: SDDG/DAGH Storage

Data-structure storage is maintained by the HDDA described in Section 3. The overall storage scheme is shown in Figure 4.4.

Programming Abstractions for Hierarchical AMR

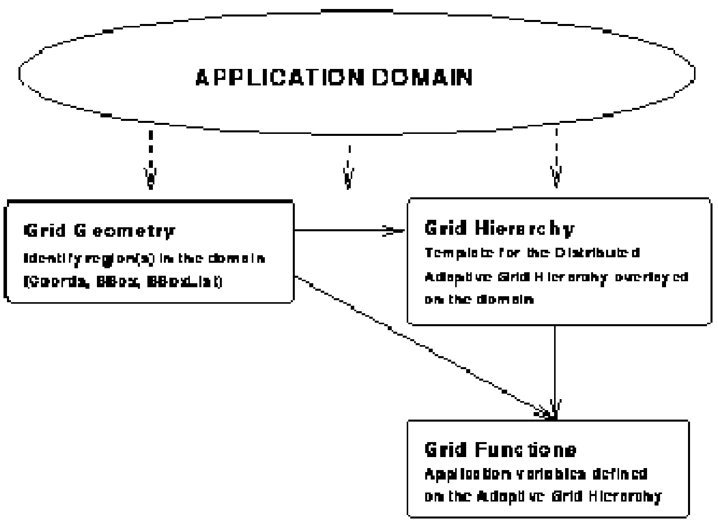


Figure 4.5: Programming Abstraction for Parallel Adaptive Mesh-Refinement

We have developed three fundamental programming abstractions using the data-structures described above that can be used to express parallel adaptive computations based on adaptive mesh refinement (AMR) and multigrid techniques (see Figure 4.5). Our objectives are twofold: first, to provide application developers with a set of primitives that are intuitive for expressing the application, and second, to separate data-management issues and implementations from application specific operations.

Grid Geometry Abstractions:

The purpose of the grid geometry abstractions is to provide an intuitive means for identifying and addressing regions in the computational domain. These abstractions can be used to direct computations to a particular region in the domain, to mask regions that should not be included in a given operation, or to specify region that need more resolution or refinement. The grid geometry abstractions represent coordinates, bounding boxes and doubly linked lists of bounding boxes.

Coordinates: The coordinate abstraction represents a point in the computational domain. Operations defined on this class include indexing and arithmetic/logical manipulations. These operations are independent of the dimensionality of the domain.

Bounding Boxes: Bounding boxes represent regions in the computation domain and is comprised of a triplet: a pair of *Coords* defining the lower and upper bounds of the box and a *step array* that defines the granularity of the discretization in each dimension. In addition to regular indexing and arithmetic operations, scaling, translations, unions and intersections are also defined on bounding boxes. Bounding boxes are the primary means for specification of operations and storage of internal information (such as dependency and communication information) within DAGH.

Bounding Boxes Lists: Lists of bounding boxes represent a collection of regions in the computational domain. Such a list is typically used to specify regions that need refinement during the regriding phase of an adaptive application. In addition to linked-list addition, deletion and stepping operation, reduction operations such as intersection and union are also defined on a BBoxList.

Grid Hierarchy Abstraction:

The grid hierarchy abstraction represents the distributed dynamic adaptive grid hierarchy that underlie parallel adaptive applications based on adaptive mesh-refinement. This abstraction enables a user to define, maintain and operate a grid hierarchy as a first-class object. Grid hierarchy attributes include the geometry specifications of the domain such as the structure of the base grid, its extents, boundary information, coordinate information, and refinement information such as information about the nature of refinement and the refinement factor to be used. When used in a parallel/distributed environment, the grid hierarchy is partitioned and distributed across the processors and serves as a template for all application variables or grid functions. The locality preserving *composite distribution* [9] based on recursive *Space-filling Curves* [3] is used to partition the dynamic grid hierarchy. Operations defined on the grid hierarchy include indexing of individual component grid in the hierarchy, refinement, coarsening, recomposition of the hierarchy after regriding, and querying of the structure of the hierarchy at any instant. During regriding, the re-partitioning of the new grid structure, dynamic load-balancing, and the required data-movement to initialize newly created grids, are performed automatically and transparently.

Grid Function Abstraction:

Grid Functions represent application variables defined on the grid hierarchy. Each grid function is associated with a grid hierarchy and uses the hierarchy as a template to define its structure and distribution. Attributes of a grid function include type information, and dependency information in terms of space and time stencil radii. In addition the user can assign special (FORTRAN) routines to a grid function to handle operations such as inter-grid transfers (prolongation and restriction), initialization, boundary updates, and input/output. These function are then called internally when operating on the distributed grid function. In addition to standard arithmetic and logical manipulations, a number of reduction operations such as *Min/Max*, *Sum/Product*, and *Norms* are also defined on grid functions. GridFunction objects can be locally operated on as regular FORTRAN 90/77 arrays.

4.2 Definition of hp-Adaptive Finite Element Mesh

The hp-adaptive finite element mesh data structure consists of two layers of abstractions, as illustrated in Figure 4.6. The first layer consists of the *Domain* and *Node* abstractions. The second layer consists of mesh specific abstractions such as *Vertex*, *Edge*, and *Surface*, which are specializations of the *Node* abstraction.

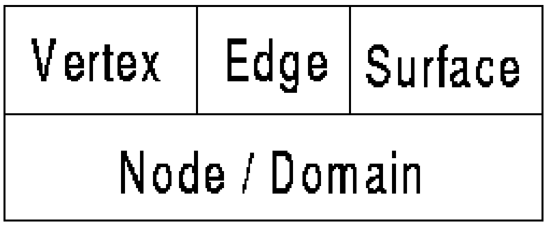


Figure 4.6: Layering of Mesh Abstraction

A mesh *Domain* is the finite element application's specialization of the SDDA. The *Domain* uses the SDDA to store and distribute a dynamic set of mesh *Nodes* among processors. The *Domain* provides the mapping from the N-dimensional finite element domain to the one-dimensional index space required by a DDA.

A finite element mesh *Node* associates a set of finite element basis functions with a particular location in the problem domain. *Nodes* also support inter-*Node* relationships, which typically capture properties of inter-*Node* locality.

Specializations of the finite element mesh *Node* for a two-dimensional problem are summarized in the following table and illustrated Figure 4.7.

Mesh Object	Reference Location	Relationships
Vertex	vertex point	
Edge	midside point	Vertex endpoints
		Element "owners"
		Irregular edge constraints
Element	centroid point	Edge boundaries
		Element refinement heirarchy

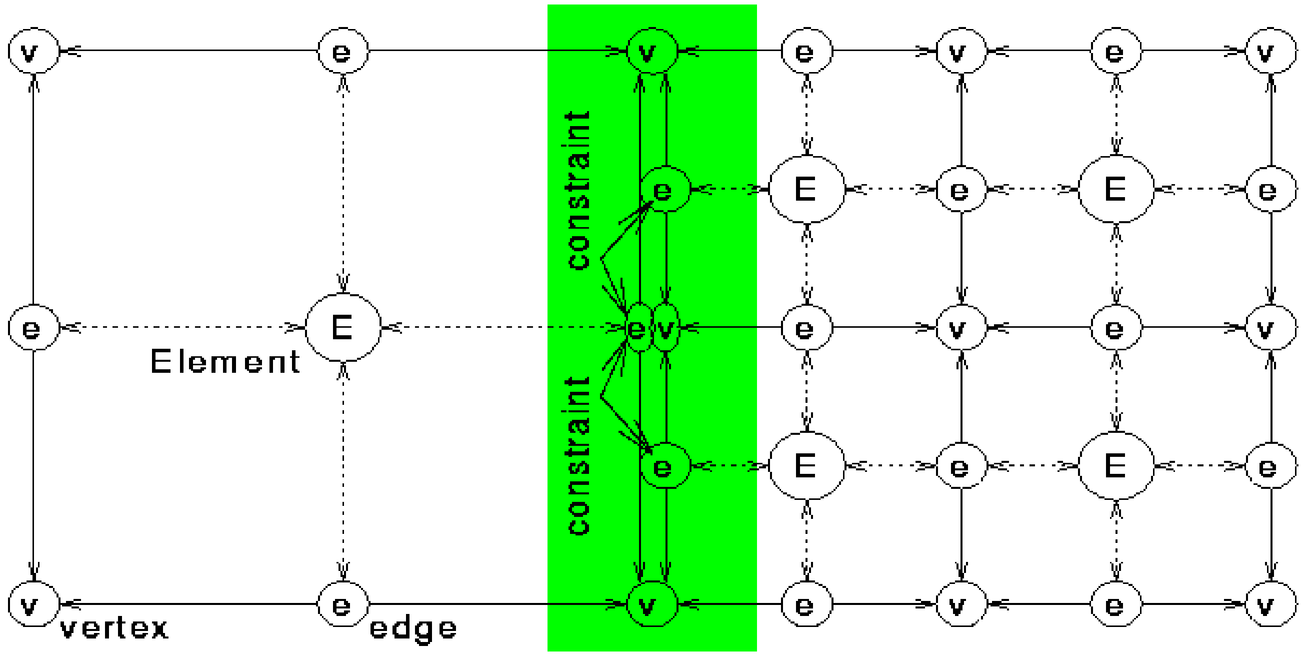


Figure 4.7: Mesh Object Relationships

Extended relationships between mesh *Nodes* are obtained through the minimul set of relationships given above. For example:

Extended Relationship	Relationship "Path"	
Element ->Vertex	Element <-> Edge -> Vertex	
Element <-> Element	Element <-> Edge <-> Element	(normal)
Element <-> Element	Element <-> Edge <-> Edge <-> Element	(constrained)

Finite element h-adaptation consists of splitting elements into smaller elements, or merging previously split elements into a single larger elements. Finite element p-adaptation involves increasing or decreasing the number of basis functions associated with the elements. An application performs these hp-adaptations dynamically in response to an error analysis of a finite element solution.

HP-adaptation results in the creation of new mesh *Nodes* and specification of new inter-*Node* relationships. Following an hp-adaptation the mesh partitioning may lead to load imbalance, as such the application may repartition the problem. A DDA significantly simplifies such dynamic data structure update and repartitioning operations while insuring data structure consistency throughout these operations.

4.3 Adaptive Trees

An adaptive distributed tree requires two main pieces of information. First it needs a tree data structure with methods for gets, puts, and pruning nodes of the tree. This infrastructure requires pointers between nodes. Second an adaptive tree needs an estimation of the cost associated with each node of the tree in order to determine if any refinement will take place at that node. With these two abstractions, an algorithm can utilize an adaptive tree in a computation. At this point, we are developing a distributed fast multipole method based on balanced trees, with the goal of creating a mildly adaptive tree in the near future.

Adaptive trees could be defined in either of the DDAs. The implementation described here is done using the SDDA. All references in the tree are made through a generalization of the pointer concept. These pointers are implemented as indices into the SDDA, and access is controlled by accessing the SDDA data object with the appropriate action and index. This control provides a uniform interface into a distributed data structure for each processor. Thus, distributed adaptive trees are supported on the SDDA.

The actual contents of a node includes a list of items.

1. An index of each node derived from the geometric location of the node.
2. Pointers to a parent and to children nodes.
3. An array of coefficients used by the computation.
4. A list of pointers to other nodes with which the given node interacts.

All of this information is stored in a node, called a subdomain. The expected work for each subdomain is derived from the amount of computation to be performed as specified by the data. Adaptivity can be determined on the basis of the expected work of a given node, relative to some threshold. In addition, since each node is registered in the SDDA, we can also compute the total expected work per processor. By collecting the total expected work per processor with the expected work per subdomain, a simple load balance can be implemented by repartitioning the index space.

5 Application Codes

There follow sketches of applications expressed in terms of each of the parallel adaptive mesh refinement method, the parallel hp-adaptive method and the parallel many-body problem each built on programming abstractions built upon a DDA.

5.1 Numerical Relativity using Hierarchical AMR

A distributed and adaptive version of *H3expresso* 3-D numerical relativity application has been implemented using the the data-management infrastructure presented in this paper. The *H3expresso* 3-D numerical relativity application is developed at the National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana, has *H3expresso* (developed at National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana) is a ‘‘concentrated’’ version of the full *H* version 3.3 code that solves the general relativistic Einstein’s Equations in a variety of physical scenarios [10]. The original *H3expresso* code is non-adaptive and is implemented in FORTRAN 90.

Representation Overheads

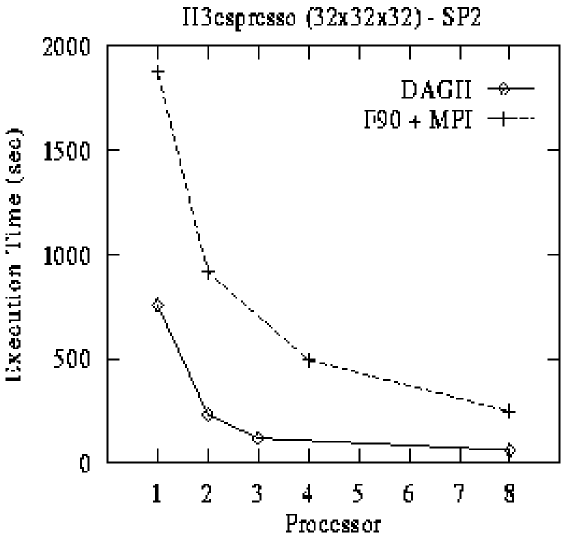
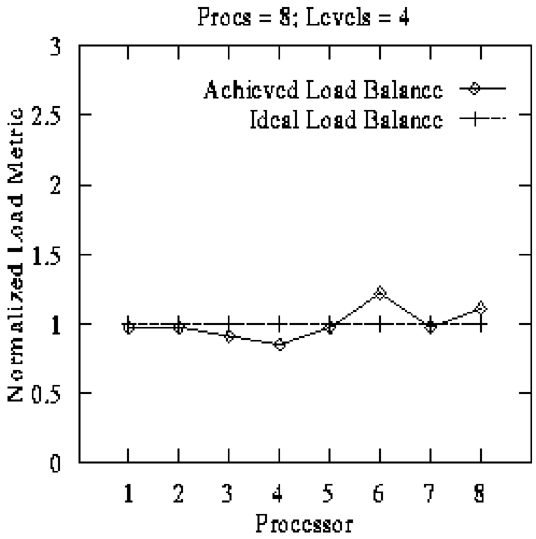


Figure 5.1: DAGH Overhead Evaluation

The overheads of the proposed DAGH/SDDG representation are evaluated by comparing the performance of a hand-coded, unigrid, Fortran 90+MPI implementation of the *H3expresso* application with a version built using the data-management infrastructure. The hand-coded implementation was optimized to overlap the computations in the interior of each grid partition with the communications on its boundary by storing the boundary in separate arrays. Figure 5.1 plots the execution time for the two codes. The DAGH implementation is faster for all number of processors.

Composite Partitioning Evaluation

The results presented below were obtained for a 3-D base grid of dimension 8 X 8 X 8 and 6 levels of refinement with a refinement factor of 2.



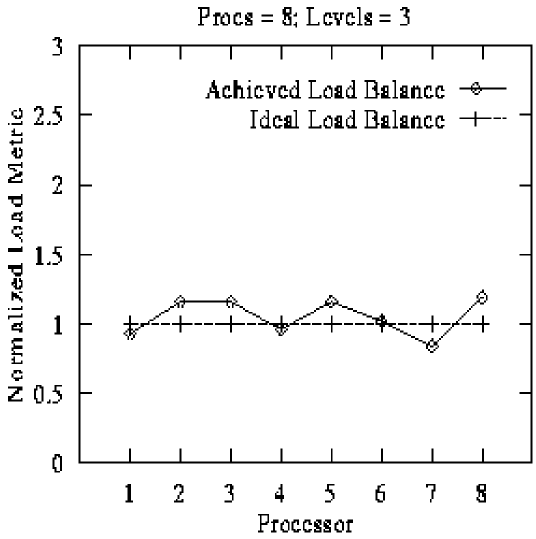
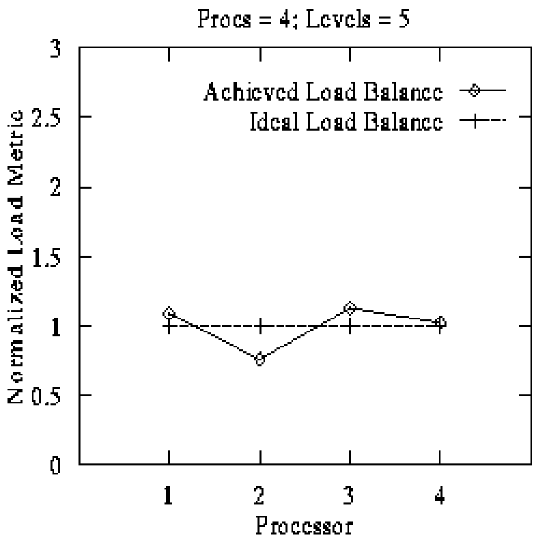


Figure 5.2: DAGH Distribution: Snap-shot I - Figure 5.3: DAGH Distribution: Snap-shot II



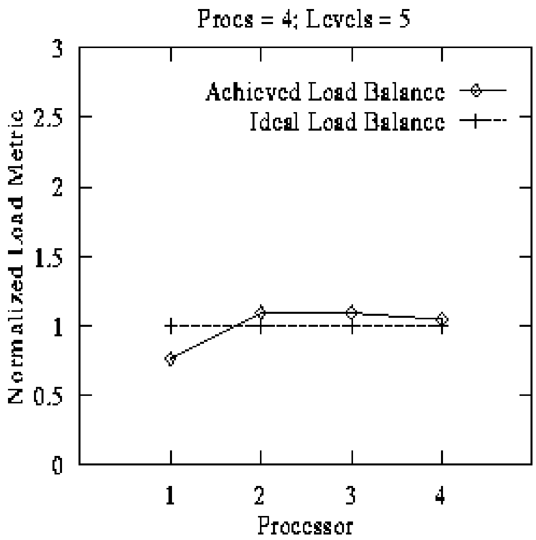


Figure 5.4: DAGH Distribution: Snap-shot III - Figure 5.5: DAGH Distribution: Snap-shot IV

Load Balance:

To evaluate the load distribution generated by the composite partitioning scheme we consider snap-shots of the distributed grid hierarchy at arbitrary times during integration. Normalized computational load at each processor for the different snap-shots are plotted in Figures 5.2-15.5. Normalization is performed by dividing the computational load actually assigned to a processor by the computational load that would have been assigned to the processor to achieve a perfect load-balance. The latter value is computed as the total computational load of the entire DAGH divided by the number of processors.

Any residual load imbalance in the partitions generated can be tuned by varying the granularity of the SDDG/DAGH blocks. Smaller blocks can increase the regriding time but will result in smaller load imbalance. Since AMR methods require re-distribution at regular intervals, it is usually more critical to be able to perform the re-distribution quickly than to optimize each distribution.

Inter-Grid Communications:

Both prolongation and restriction inter-grid operations were performed locally on each processor without any communication or synchronization.

Partitioning Overheads

Procs	Update Time	Regriding Time
4	28.5 sec	1.84 sec
8	19.2 sec	1.58 sec

Table 5.1: Dynamic Partitioning Overhead

Partitioning is performed initially on the base grid, and on the entire grid hierarchy after every regrid. Regriding any level *l* comprises of refining at level *l* and all level finer than *l*; generating and distributing the new grid hierarchy; and performing data transfers required to initialize the new hierarchy. Table 5.1 compares the total time required for regriding, i.e. for refinement, dynamic re-partitioning and load balancing, and data-movement, to the time required for grid updates. The values listed are cumulative times for 8 base grid time-steps with 7 regrid operations.

5.2 HP-Adaptive Finite Element Code

An parallel hp-adaptive finite element code for computational fluid dynamics is in development. This hp-adaptive finite element computational fluid dynamics application has two existing implementations: (1) a sequential FORTRAN code and (2) a parallel FORTRAN code with fully a duplicated data structure. The hp-adaptive finite element data structure has a complexity so great that is was not tractable to distribute the FORTRAN data structure. As such the parallel FORTRAN implementation is not scalable due to the memory consumed on each processor by duplicating the data structure.

To make tractable the development of a fully distributed hp-adaptive finite element data structure is was necessary to achieve a separation of concerns between the complexities of the hp-adaptive finite element data structure and complexities of distributed dynamic data structures in general. This separation of concerns in the development of a fully distributed hp-adaptive finite element data structure provided the initial motivation for developing the SDDA.

The organization of the new finite element application is illustrated in Figure 5.6. At the core of the application architecture is the index space. The index space provides a very compact and succinct specification of how to partition the application's problem among processors. This same specification is used to both distribute the mesh structure through the SDDA and to define a compatible distribution for the vectors and matrices formed by the finite element method.

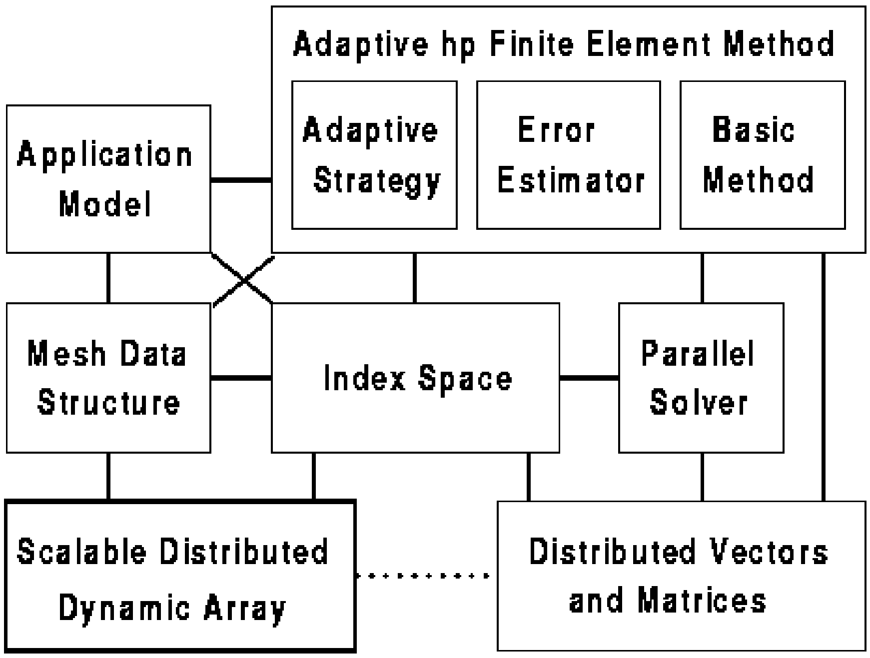


Figure 5.6: Finite Element Application Architecture

The finite element application uses a second parallel infrastructure which supports distributed vectors and matrices, as denoted in the lower right corner of Figure 5.6. The current release of this infrastructure is documented in [11]. Both DDA and linear algebra infrastructures are based upon the common abstraction of an *index space*. This commonality provides the finite element application with uniform abstraction for specifying data distribution.

5.3 N-Body Problems

General Description

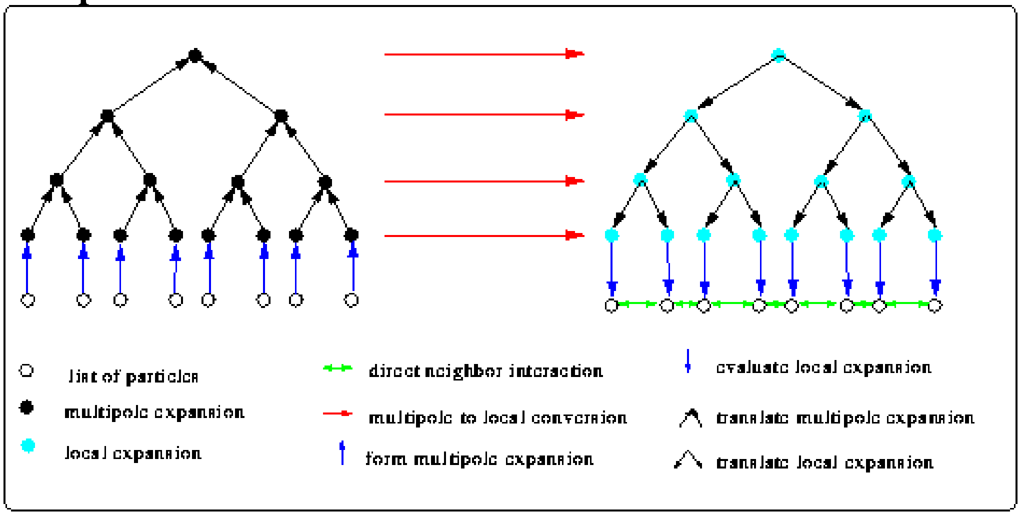


Figure 5.7: Data flow in the fast multipole algorithm

The N-body particle problems arising in various scientific disciplines appear to require an $O(N^2)$ computational method. However, once a threshold in the number of particles is surpassed, approximating the interaction of particles with interactions between sufficiently separated particle clusters allows the computational effort to be substantially reduced. The best known of these fast summation approaches is the fast multipole method [12], which, under certain assumptions, gives a method of $O(N)$

The fast multipole method is a typical divide and conquer algorithm. A cubic computational domain is recursively subdivided into octants. At the finest level, the influence of the particles within a cell onto sufficiently separated cells is subsumed into a multipole series expansion. These multipole expansions are combined in the upper levels, until the root of the oct-tree contains a multipole expansion. Then local series expansions of the influence of sufficiently separated multipole expansions on cells are formed.

Finally, in a reverse traversal of the oct-tree the contributions of cells are distributed to their children (cf. Figure 5.7). The algorithm relies on a scaling property, which allows cells on the scale of children to be sufficiently separated when they were too close on the scale of the current parents. At the finest level the influence of these sufficiently separated cells is taken into account together with the interaction of the particles in the remaining closeby cells. For a more mathematically oriented description of the shared memory implementation with or without periodic boundary conditions we refer to [13][14] and the references therein.

Figure 5.7 shows that the fast multipole algorithm is readily decomposed into three principal stages:

1. populating the tree bottom-up with multipole expansions
2. converting the multipole expansions to local expansions
3. distribute local expansions top-down

These three stages have to be performed in sequential order, but it is easily possible to parallelize each of the stages individually. For the second stage the interaction set of a cell is defined as the set of cells which are sufficiently separated from the cell, but their parents are not sufficiently separated from the cells parent cell. The local expansion about the center of a cell is found by adding up all the influences from the cells of the interaction set. For each cell these operations are found to be completely independent of each other. But notice that the majority of communication requirements between different processors is incurred during this stage. Hence, an optimization of the communication patterns during the second stage can account for large performance gains.

A more detailed analysis of the (non-adaptive) algorithm reveals that optimal performance should be attained when each leaf-cell contains an optimal number of particles, thereby balancing the work between the direct calculations and the conversion of the multipole expansions to the local expansions in the second stage. **Distributed Fast Multipole Results**

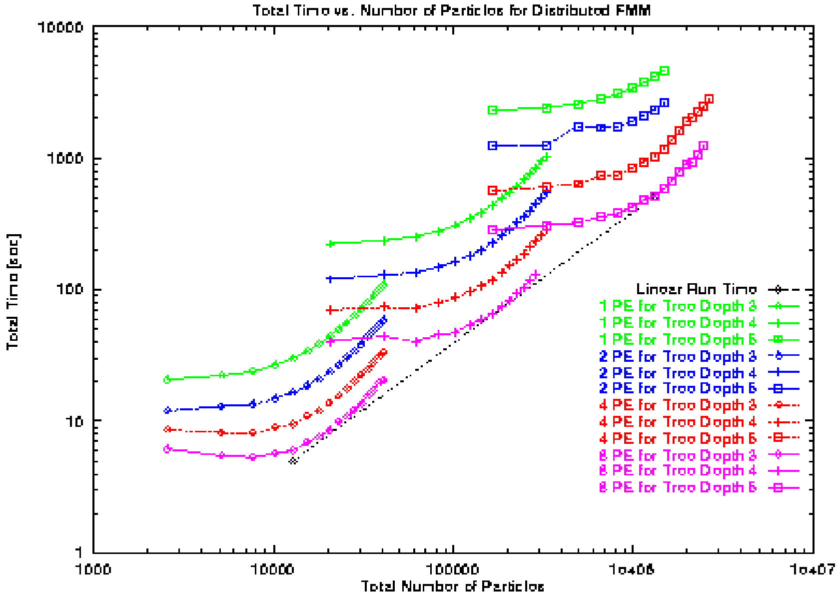


Figure 5.8: Total Execution Time on SP2 vs. Problem Size for Multiple Levels of Approximation

Next we describe preliminary performance results for the distributed fast multipole method implemented on the HDDA. We verified the fast multipole method by computing an exact answer for a fixed resolution of the computational domain. Test problems were constructed by storing one to a few particles chosen randomly per leaf cell. This comparison was repeated for several small problems until we were certain that the algorithm behaved as expected. Performance measurements were taken from a 16 node IBM SP2 parallel computer running AIX Version 4.1.

The total run times using 3, 4, and 5 levels of approximation on a variety of problem sizes for 1, 2, 4, and 8 processors are presented in Figure 5.8. We also plot the expected *O(N)* run time aligned with the 8 processor results. Each curve represents the total execution time for a fixed level of spatial resolution while increasing the number of particles in the computation. Two major components of each run time are an approximation time and a direct calculation for local particles. The approximation time is a function of the number of levels of resolution and is fixed for each curve. The direct calculation time grows as $O(N^2)$ within a given curve. The problem size for which these two times are equal represents the optimal number of particles stored per subdomain. This optimal problem size appears as a knee in the total execution time curves.

The curves for 8 processors align relatively well with the *O(N)* run time for all problem sizes. Thus, the algorithm appears scalable for the problems considered. Furthermore, these results show that 1 processor has the longest time and 8 processors have the shortest time, which indicates that some speedup is being attained. Ideally, one would expect that

8 processors achieve a speedup of 8.

We have presented results for a fast multipole method which demonstrates the *O(N)* computational complexity. These results exhibit both scalability and speedup for a small number of processors. Our preliminary results focused on validity and accuracy. The next step is to performance tune the algorithm.

6 Conclusion and Future Work

The significant conclusions demonstrated herein are:

1. That there is a common underlying computational infrastructure for a wide family of parallel adaptive computation algorithms.

2. That a substantial decrease in effort in implementation for these important algorithms can be attained without sacrifice of performance through use of this computational infrastructure.

There is much further research needed to complete development of a robust and supportable computational infrastructure for adaptive algorithms. The existing versions of DDA require extension and engineering. The programming abstractions for each solution method require enriching. It is hoped to extend the programming abstraction layer to support other adaptive methods such as wavelet methods. There is a need to do many more applications to define the requirements for the programming abstraction layer interfaces.

7 Acknowledgements

This research has been jointly sponsored by the Argonne National Laboratory Enrico Fermi Scholarship awarded to Manish Parashar, by the Binary Black-Hole NSF Grand Challenge (NSF ACS/PHY 9318152), by ARPA under contract DABT 63-92-C-0042, and by the NSF National Grand Challenges program grant ECS-9422707. The authors would also like to acknowledge the contributions of Jürgen Singer, Paul Walker and Joan Masso to this work.

8 References

1. M. Parashar and J. C. Browne, *System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement*, to be published in Structured Adaptive Mesh Refinement Grid Methods, IMA Volumes in Mathematics and its Applications, Springer-Verlag,

1997.

2. Harold Carter Edwards,*A Parallel Infrastructure for Scalable Adaptive Finite Element Methods and its Application to Least Squares C-infinity Collocation*, PhD Thesis, The University of Texas at Austin, May 1997.

3. Hans Sagan, *Space Filling Curves*, Springer-Verlag, 1994.

4. H.F. Korth, A. Silberschatz, *Database System Concepts*,. McGraw Hill. New York, 1991.

5. W. Litwin. *Linear Hashing: a New Tool for File and Table Addressing*, Proceedings of the 6th Conference on VLDB, Montreal, Canada, 1980.

6. Robert Sedgewick. *Algorithms*, Addison-Wesley, Reading, Massachusetts, 1983.

7. Marsha J. Berger, Joseph Oliger, *Adaptive Mesh-Refinement for Hyperbolic Partial Differential Equations*, Journal of Computational Physics, pp. 484-512, 1984.

8. Manish Parashar and James C. Browne, *Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement*, Proceedings of the International Conference for High Performance Computing, pp. 22-27, Dec. 1995.

9. Manish Parashar and James C. Browne, *On Partitioning Dynamic Adaptive Grid Hierarchies*, Proceedings of the 29th Annual Hawaii International Conference on System Sciences, **1**:604-613, Jan. 1996.

10. J. Masso and C. Bona, *Hyperbolic System for Numerical Relativity*, Physics Review Letters, **68**(1097), 1992.

11. Robert van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*, The MIT Press, 1997.

12. Leslie Greengard, *The rapid evaluation of potential fields in particle systems*, 1987.

13. Jürgen K. Singer, *The Parallel Fast Multipole Method in Molecular Dynamics*, PhD thesis, The University of Houston, August 1995.

14. Jürgen K. Singer, *Parallel Implementation of the Fast Multipole Method with Periodic Boundary Conditions*, East-West Journal on Numerical Mathematics, **3**(3), October 1995.

Author Biography

Manish Parahar is the current recipient of the Argonne National Laboratory Enrico Fermi scholarship and holds joint appointments as Adjunct Assistant Professor, Department of Computer Sciences, and Research Associate, Texas Institute for Computational and Applied Mathematics, both at the University of Texas at Austin. He is also a visiting researcher at the Max Planck Institute in Potsdam, Germnany. Manish's research interests include high performance parallel/distributed computing, software engineering, application development tools and problem solving environments. performance evaluation and prediction. Manish received a BE degree in Electronics and Telecommunications from Bombay University, India in 1988, and MS and PhD degrees in Computer Engineering from Syracuse University in 1994. He is a member of IEEE, IEEE Computer Society, ACM, and the Phi Beta Delta honor society. Manish will be joining Rutgers University as an Assistant Professor in Computer Engineering starting Fall 1997.

James C. Browne is Professor of Computer Science and Physics and holds the Regents Chair #2 in Computer Sciences at The University of Texas at Austin. Browne earned his Ph.D. in Chemical Physics at The University of Texas in 1960. He taught in the Physics Department at The University of Texas from 1960 through 1964. He held an NSF Postdoctoral Fellowship in 1964/65. He was, from 1965 through 1968, Professor of Computer Science at Queens University in Belfast and directed the Computer Laboratory. Browne rejoined The University of Texas in 1968 as Professor of Physics and Computer Science. Browne's research interests span parallel computations, performance measurement and analysis, operating systems and programming languages. Browne has been a member of Technical Advisory Committees for Lawrence Livermore Laboratories, Los Alamos National Laboratories, the National Bureau of Standards, the National Science Foundation--Computer Research Section, the DARPA Information Science and Technology Office and Sequent Computers. He is a fellow of the British Computer Society and of the American Physical Society. He was Chairman of the ACM Special Interest Group on Operating Systems (1974/76) and has been in the past an Associate Editor of several journals. Browne has published approximately 100 papers in computational physics and 200 papers in Computer Science.

Harold Carter Edwards is currently working for the Center for Subsurface Modeling (CSM) developing parallel solution algorithms and solvers for the CSM's multi-component / multi-physics applications. In May of 1997 he earned a Ph.D. in Computational and Applied Mathematics from the University of Texas at Austin. His dissertation included a formal development data management infrastructure abstractions and SDDA implementation presented here, as well as the development of a new adaptive finite element method for solving higher order PDEs.

From 1981 through 1991 Mr. Edwards worked in the aerospace industry analyzing mission, guidance, navigation, and control requirements and developing simulators for manned and automated spacecraft. In this previous career he earned a B.S. (1982) and M.S. (1993) in Aerospace Engineering from the University of Texas at Austin.

Kenneth Klimkowski received B.S. Electrical Engineering and B.S. Computer Science degrees in 1991 from North Carolina State University. Afterwhich he earned degrees in M.S. Engineering in 1993 and M.S. Computational and Applied Mathematics 1997 from The University of Texas at Austin. He has worked on out-of-core high performance, parallel numerical linear algebra with Dr. Robert van de Geijn at The University of Texas. Also, he has worked on Fast Multipole Methods as applied to problems of composite material analysis with Drs. Greg Rodin, Robert van de Geijn, and J.C. Browne all of The University of Texas. At this time, Mr. Klimkowski works at National Instruments in Austin, TX.