A Common Computational Infrastructure for Adaptive Algorithms for PDE Solutions

J.C. Browne, C. Edwards, R. van de Geijn, K. Klimkowski, M. Parashar and J. Singer

Department of Computer Sciences & TICAM*

University of Texas at Austin

Austin, TX 78712

Technical Paper Submission

Corresponding & Presenting Author:

Manish Parashar

Department of Computer Science

TICAM, Taylor Hall 2.400

University of Texas at Austin

Austin, TX 78712

parashar@cs.utexas.edu

Tel: 512-471-1841; Fax: 512-471-8694

Abstract

This paper defines, describes and illustrates the use of an infrastructure for implementation of parallel adaptive algorithms for solution of sets of partial differential equations. The infrastructure is separated into two layers of abstraction: a layer which defines an implements a scalable dynamic distributed array (SDDA) and a layer in which several dynamic data abstractions are implemented in terms of the SDDA. The currently implemented data structures are: those needed for formulation of hp-adaptive finite element methods, those needed for hierarchical adaptive finite element methods and those needed for the fast multipole method for solution of linear systems. Implementation issues and performance measurements are given. The applications from which the data structure requirements were derived are described.

1 Introduction

1.1 Overview and Motivation

This paper defines, describes and demonstrates the application of a computational infrastructure which is an effective basis for development of parallel adaptive methods for the solution of systems of partial differential equations. While space restrictions limits the depth of coverage of the several elements of the infrastructure we felt it to be important to describe the full infrastructure and the motivating applications. URLs for web sites where additional information on each subject can be found are given in the discussions of each element of the infrastructure.

The motivations for this research are:

1. Adaptive methods will be utilized for the solution of almost all very large-scale science and engineering models. These adaptive methods will be executed upon the very large-scale heterogeneous parallel execution environment.

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 but also a more effective infrastructure due to pooling and focusing of effort.

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 ARPA project for hp-adaptive computational fluid dynamics and two NSF sponsored Grand Challenge projects, one for numerical relativity and another on composite materials. The computational infrastructure is being used at several sites of the numerical relativity Grand Challenge project.

1.2 Conceptual Framework

Figure 1 is a schematic of our perception of the structure of solution codes for sets 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. These two elements are a data management layer which implements an array abstraction and a programming abstraction layer which implements data abstractions such as grids, meshes and trees which underlie different solution methods. *The clean separation of array semantics from the higher level operations is critical to the success of this approach in providing a foundation for several different solution methods.* Each project used a different adaptive solution method. The data management infrastructure is founded on the observation that several different adaptive methods for solution of partial differential equations have a common set of requirements for management of the dynamic distributed data structures which ensue from the adaptation and parallelization of the solution. In particular it will be shown that finite difference methods based on adaptive mesh refinement, hp- adaptive finite element methods and adaptive fast multipole methods for solution of linear systems can readily be implemented on the scalable dynamic distributed array (SDDA) data structure.

Properties of the SDDA include:

1. Single processor semantics are preserved - The dynamism and distribution of the array and the communication requirements engendered by distribution are transparent to computational operations

2. The index space of the array is directly derived from the geometry of the problem.

3. The mapping of the n-dimensional index space to the one- dimensional space of linear memory preserves the locality of the problem geometry.

4. The arrays are hierarchical and dynamic in both the number of levels in the hierarchical definition and with respect to size at any level.

5. Performance of applications built on the SDDA are competitive, in fact, often superior to those programmed in conventional languages such as C and Fortran 90.

1.3 Related Work

There is a rich spectrum of previous work, some focusing on support for specific adaptive solution methods [1] and several systems which implement distributed arrays, grids or matrices ([2, 3, 4, 5, 6]). The two lowest levels of the computational infrastructure, the SDDA and the programming abstractions, are an evolutionary development which owe significant intellectual debts to the previous systems. The extensions of the infrastructure presented herein over previous implementations are:

1. broader support for dynamic structuring,

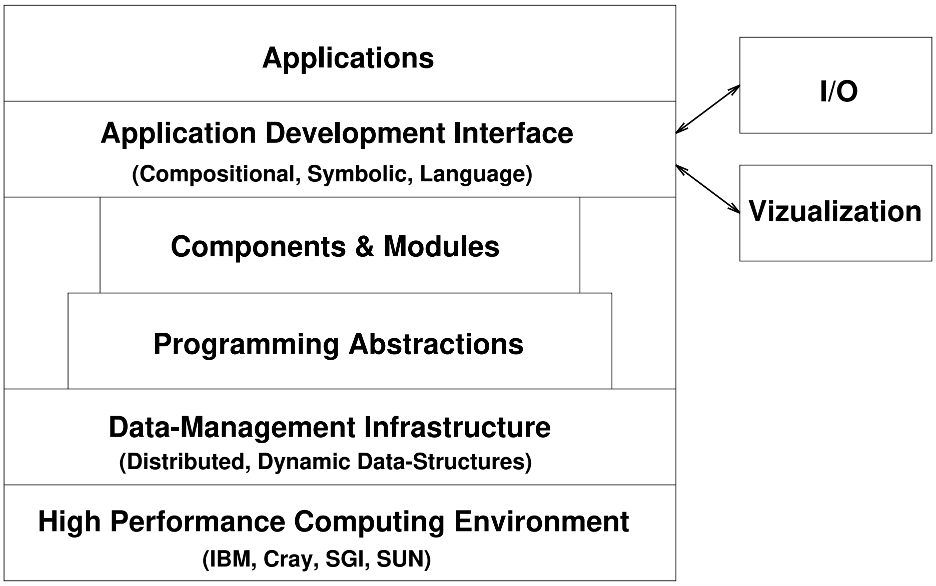

Figure 1: Hierarchical Structure for Solution Systems for Partial Differential Equations

2. preservation of geometric locality of the problem in the storage structure and
3. more importantly, a clean separation of data management issues from programming abstraction issues abstraction issues.

1.4 Problem Specification

Adaptive algorithms require definition of operators on complex dynamic data structures. Two problems arise: (i) the volume and complexity of the bookkeeping code required to format the data structures overwhelms the actual computations and (ii) maintaining locality under expansion and contraction of the arrays 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. Additionally the standard parallel programming languages do not provide explicit support for dynamic distributed data structures.

1.5 Goal and Approach

The goal for this research is to reduce the intrinsic complexity of coding of parallel adaptive algorithms by providing an appropriate set of data structures and programming abstractions.

Several versions of this infrastructure are available on the Web. It will be made available as an "STL-like" class library.

This research has been executed as collaborations among computer scientists, computational scientists and application domain researchers on three different projects. This collaboration is of critical importance to the development of the infrastructure. It has been repeated found during the development to date that the applications cannot foresee their requirements for implementation of a given method prior to implementation and that the computer and computational scientists cannot predict what is needed for the implementation of a given method.

2 Implementation Design for the SDDA

The components of the SDDA are:

- a hierarchical dynamic index space,
- a data storage structure (using extendable hashing) which preserves data locality under expansion and contraction, and
- a data distribution mechanism.

Applications configure the SDDA by deriving their own hierarchical dynamic index space from a discretization of an application's problem geometry. This configuration process is best explained by a simple illustration of the formulation of an index space from a problem geometry.

2.1 Hierarchical Dynamic Index Space

An application partitions its N-dimensional problem domain into a finite number of regions. Each region can be associated with a unique coordinate in the problem domain. A discretization of these coordinates yields a finite set of N-dimensional indices within an N-dimensional index space.

Such an N-dimensional index space can be mapped to a 1-dimensional index space. A well known family of such maps are known as space-filling curves [7, 8, 9, 10, 11]. This mapping is used to fully order the finite set of N-dimensional indices associated with an application's partition of the N-dimensional problem domain.

Figure 2, Space-Filling Curve Ordering of a Partitioned Domain, illustrates this process for an irregularly partitioned two dimensional problem domain. In this illustration a Hilbert space-filling curve is used to fully order the subregions. This particular space-filling curve has the property that points which are local along the one-dimensional curve map to points which are local in the N-dimensional space. This locality preserving property of the mapping is an essential feature of an application's configuration of the SDDA via the hierarchical dynamic index space.

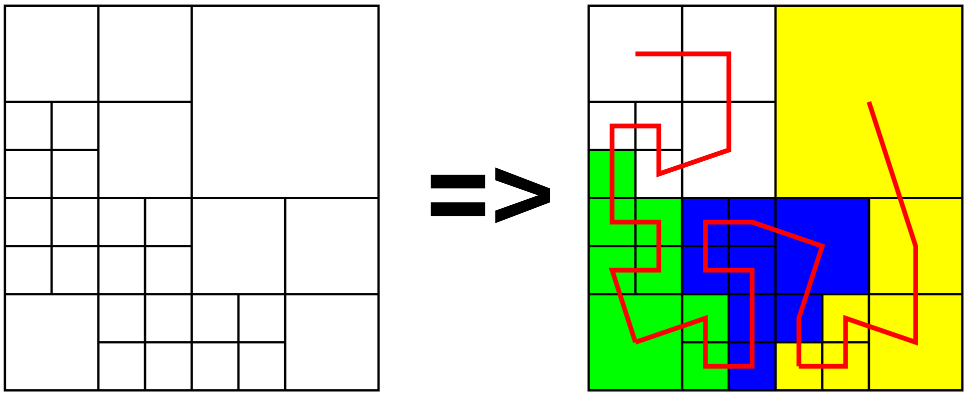

Figure 2: Space-Filling Curve Ordering of a Partitioned Domain

The space-filling curve maps have a recursive property, known as self-similarity, which leads to a computationally efficient implementation of the map. The level of recursion in a space-filling curve's map directly corresponds to a discretization granularity for the N-dimensional problem domain. Thus both the 1-dimensional and N-dimensional index spaces are hierarchical, and dynamic as the application can vary the map's level of recursion, or granularity, at run-time.

Once an application has defined an index space the application may easily partition its problem among processors by simply partitioning the index space. The color (or shade) coded regions in Figure 2 denote a partitioning of the problem geometry among four processors. This problem partitioning is easily and efficiently obtained by partitioning the 1-dimensional index space such that the computational load of each partition is roughly equal. Note that this approach, by virtue of the locality preserving property of the space-filling curve, results in a "good" partitioning of the problem.

2.2 Storage Structure: Locality, Locality, Locality!

The SDDA provides a storage structure, derived from extendable hashing [12, 13], which provides storage locality for an application's data objects which are assigned local indices in the index space. Thus the "array" semantics where index locality implies storage locality.

Figure 3, Locality Preserving Property, illustrates the end-to-end locality preserving property of the SDDA when configured with an index space derived from a space-filling curve. An

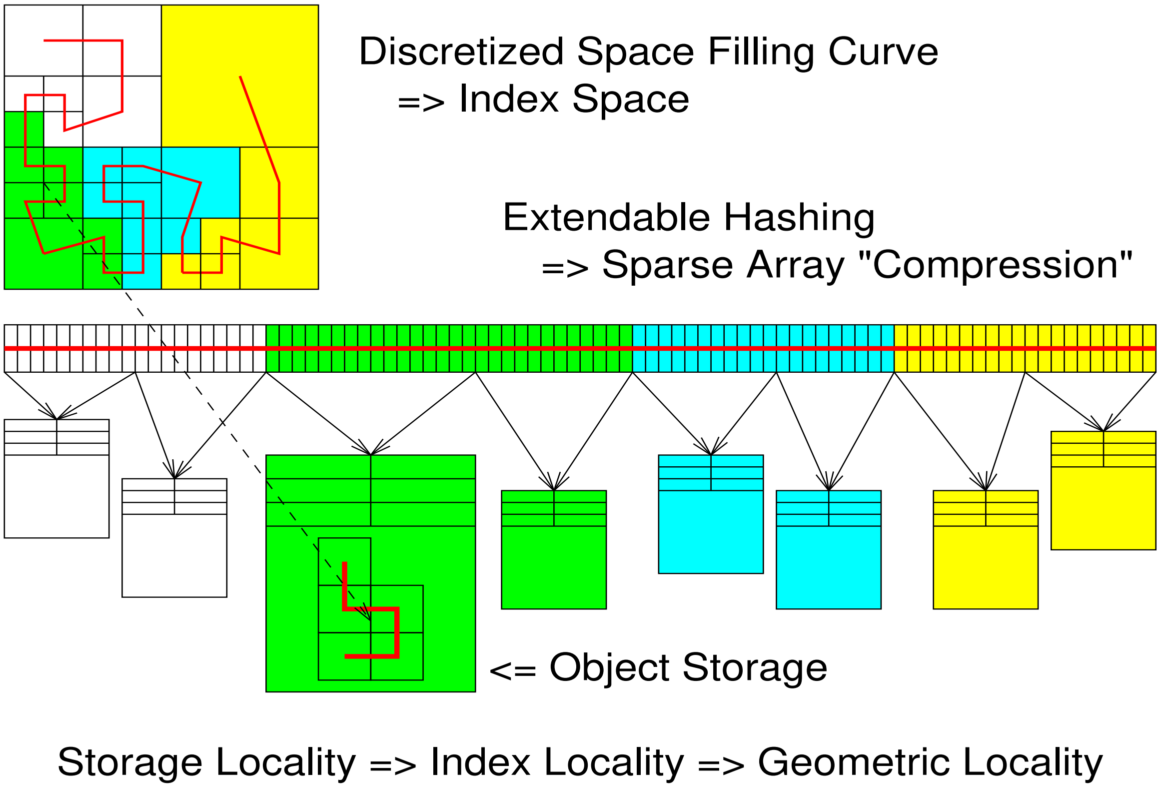

Figure 3: Locality Preserving Property

application assigns indices to subregions via the space-filling curve mapping. The SDDA stores objects within a span of indices in contiguous blocks of memory, as illustrated. Thus subregions which have locality in the appliation's problem domain have locality in their associated data object's storage.

2.3 Distribution Mechanism

Recall that the application's partitioning of the index space defined a partitioning of the application's problem among processors. This index space partitioning also defines a distribution of the application's data objects among processors. The SDDA automatically distributes the application's data among processors, as illustrated by the color (or shade) coding of data storage blocks in the above illustration. The SDDA then provides global access to the application's data by transparently caching these storage blocks between processors.

2.4 Application Progamming Interface

The SDDA interface consists of a simple set of methods. These simple methods, as denoted below, hide the complexity of the storage structure and make transparent any required interprocessor communication.

Method	arguments	description
put	(IN: Index, IN: Object)	store object
get	(IN: Index, OUT: Object)	retrieve object
next	(IN OUT: Index, OUT: Object)	iterate over objects
delete	(IN: Index)	delete object
repart	(IN: Index Array)	repartition and redistribute

2.5 Performance

Two aspects of SDDA performance are evaluated: 1) time required for local put/get operations and 2) time required for interprocessor caching. The performance of these operations is dependent upon the computational environment, the SDDA's storage block size, and the size of the application's data objects. The performance results presented here where obtained from a network of RS6000/3BT workstations using the MPICH communication library from Argonne National Labs.

Overhead for Local Operations

Figures 4 and 5 illustrate the overhead associated with inserting and deleting objects in the SDDA. These results were obtained by inserting 5000 objects with random indices into a single processor SDDA and then deleting all 5000 objects. These operations introduce two overhead costs: 1) inserting/deleting the object from a storage block and 2) dynamically expanding and contracting the set of storage blocks. Thus the per-operation cost includes an amortization of the storage expansion/contration performed periodically during the 5000 inserts and deletes.

The data object size and storage block size are varied to determine their effect on SDDA performance. The overhead cost of inserting and deleting is dominated by the storage block size. This is an expected result as larger storage blocks require greater memory movement for expansion and contraction of the SDDA's storage. The "acceptability" of this overhead is a function of the work performed with an object. Each data object stored in the SDDA should have a sufficient lifetime and data content to acceptably amortize its creation and deletion overhead.

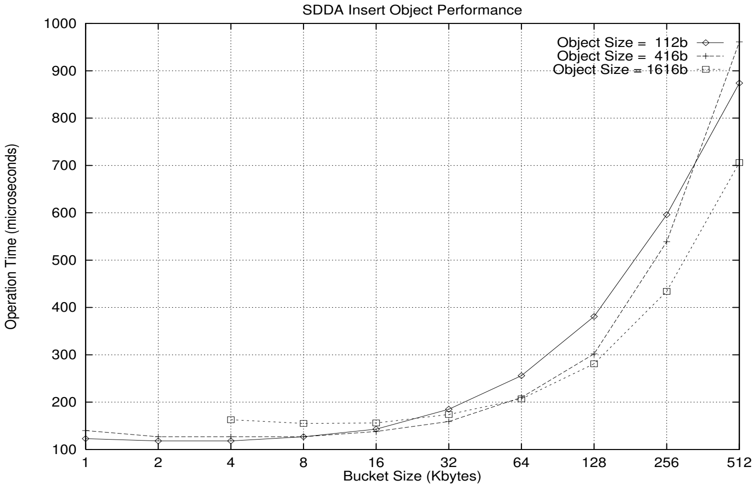

Figure 4: SDDA Insert Object Performance

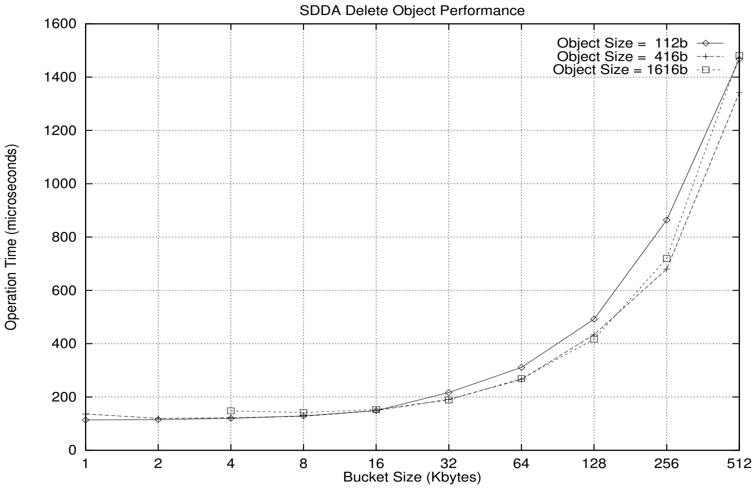

Figure 5: SDDA Delete Object Performance

Data Distribution Performance

Figure 6 illustrates the data transfer rate of the SDDA's interprocessor data caching mechanism. Interprocessor data caching occurs through a pair of asynchronous request and reply messages. Interprocessor caching is unscheduled, any processor may request data from any other processor at any time, thus the performance experiment varies the number of processor "neighbors" from which data is simultaneously cached.

As seen in Figure 6 the transfer rate improves with bucket size, up to a start-up versus bandwidth limit. An additional argument for modest size buckets is that a large bucket will likely contain numerous data objects which are not needed on the remote processor. Thus more data

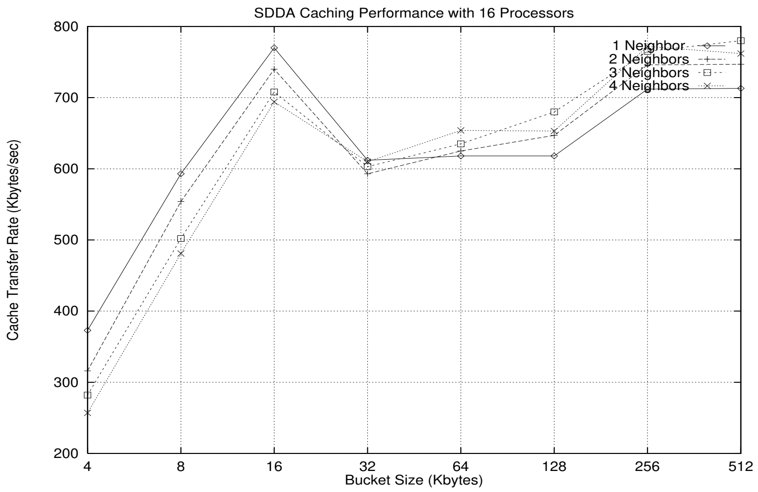

Figure 6: SDDA Caching Performance

is transfered than is actually needed.

3 Method Specific Data Abstractions

The next level above the SDDA establishes method specific data abstractions for the several different methods of solutions. The three higher level data abstractions which have been defined to date are dynamically hierarchical grids, hp-adaptive finite elements and dynamic trees. (Dynamic trees are the data abstractions upon which the fast multipole methods are implemented.)

3.1 Data Abstractions for Hierarchical Adaptive Mesh-Refinement

3.1.1 Problem Description

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

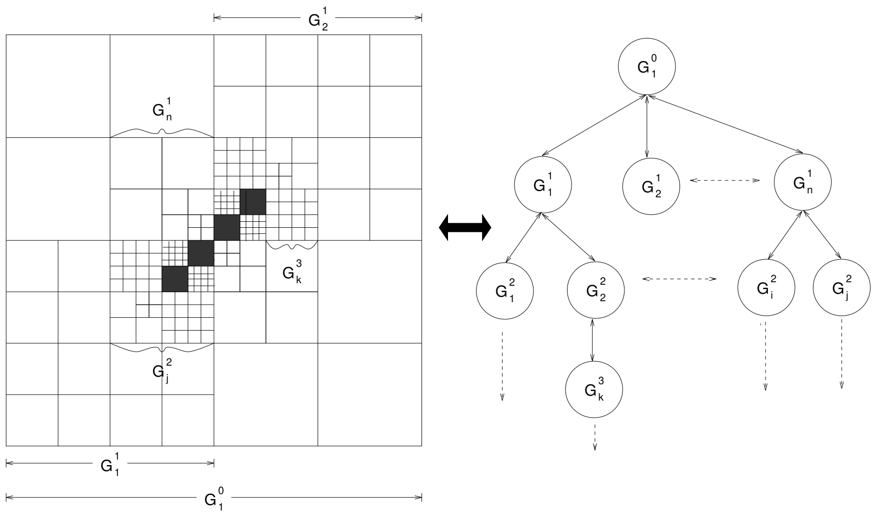

Figure 7: Adaptive Grid Hierarchy - 2D (Berger-Oliger AMR Scheme)

regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The adaptive grid hierarchy corresponding to the AMR formulation by Berger & Oliger [14] is shown in Figure 7.

3.1.2 Distributed Data-Structures for Hierarchical AMR

Two basic distributed data-structures have been developed, using the fundamental abstractions provided by the SDDA (see Section 2, 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 2 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 [15].

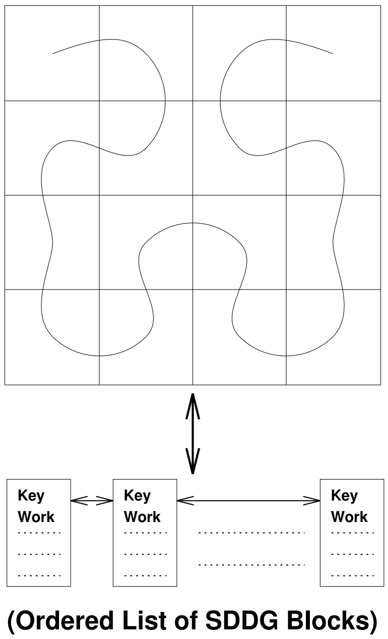

Figure 8: SDDG Representation

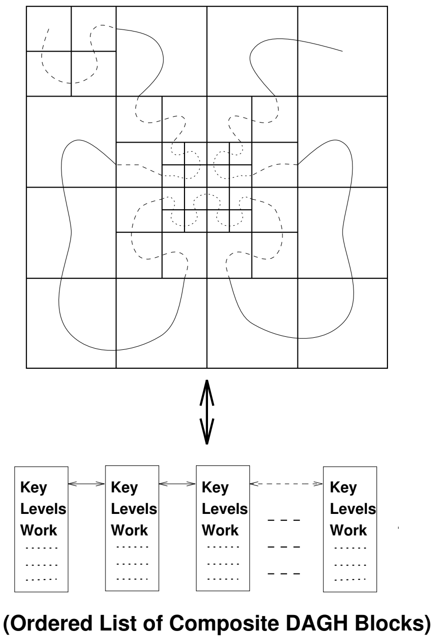

Figure 9: DAGH Composite Representation

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 8 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 9 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.

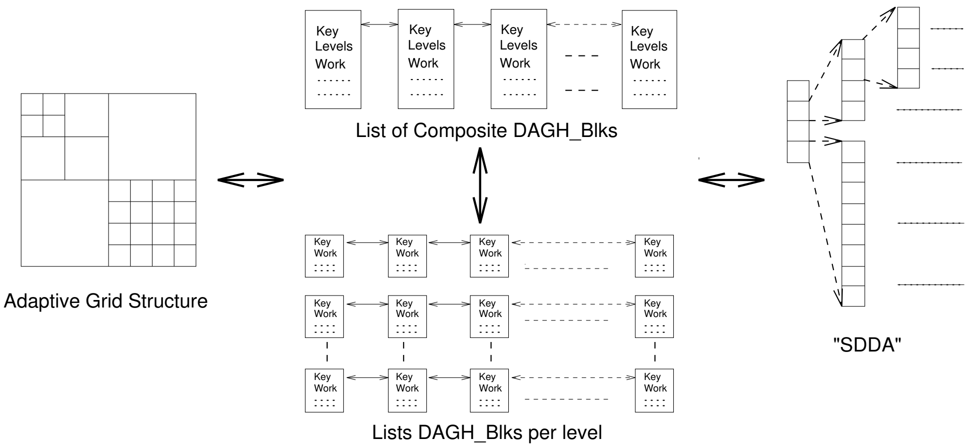

Figure 10: SDDG/DAGH Storage

Data-Structure Storage: Data-structure storage is maintained by the SDDA described in Section 2. The overall storage scheme is shown in Figure 10.

3.1.3 Programming Abstractions for Hierarchical AMR

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 11). Our objectives are twofold: firstly, to provide application developers with a set of primitives that are intuitive for implementing the problem, i.e. application objects $\equiv$ abstract datatypes. Our second objective is a separation of data-management issues and implementations from application specific computations.

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

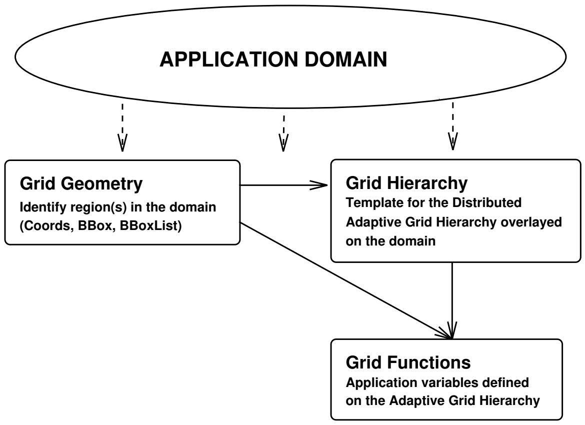

Figure 11: Programming Abstraction for Parallel Adaptive Mesh-Refinement

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* [16] based on recursive *Space-filling Curves* [17] 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.

3.2 Definition of hp-Adaptive Finite Elements

The hp-adaptive finite element mesh data structure consists of two layers of abstractions, as illustrated in Figure 12. 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.

A mesh **Domain** is the finite element application's specialization of the SDDA. The **Domain**

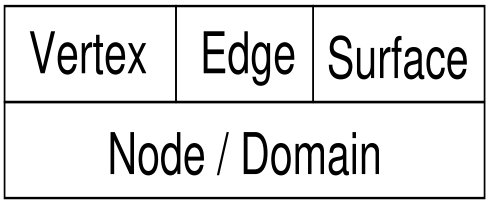

Figure 12: Layering of Mesh Abstractions

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 the SDDA.

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 13.

Name	Location	Relationships
Vertex	vertex point	
Edge	midside point	Vertex endpoints
		Element “owners”
		Irregular edge constraints
Element	centroid point	Edge boundaries
		Element refinement heirarchy

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

- 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.

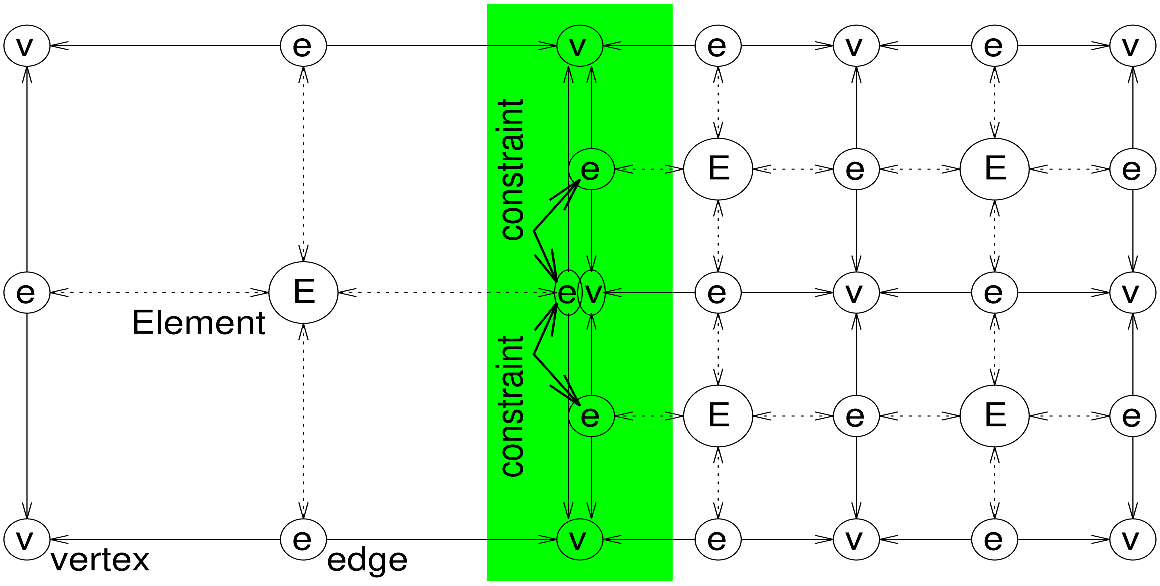

Figure 13: Mesh Object Relationships

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. The SDDA significantly simplifies such dynamic data structure update and repartitioning operations while insuring data structure consistency throughout these operations.

3.3 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.

All references in the tree are made through a generalization of the pointer concept. These pointers are implemented as keys into the SDDA, and access is controlled by accessing the SDDA data object with the appropriate action and key. 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. A key 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.

4 Example 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 on the SDDA.

4.1 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 [18]. The original *H3expresso* code is non-adaptive and is implemented in Fortran 90.

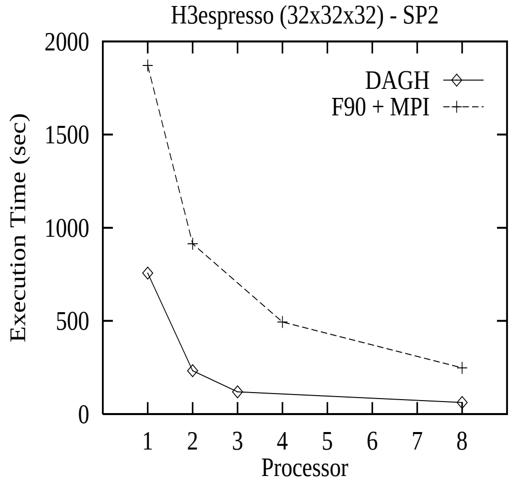

Figure 14: DAGH Overhead Evaluation

4.1.1 Representation Overheads

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 14 plots the execution time for the two codes. The DAGH implementation is faster for all number of processors.

4.1.2 Composite Partitioning Evaluation

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

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 15-18. 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.

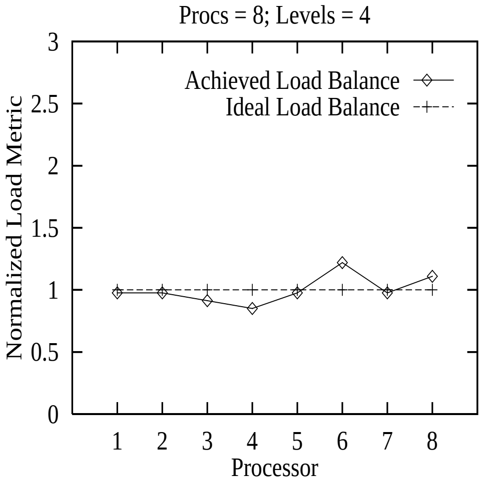

Figure 15: DAGH Distribution: Snap-shot I

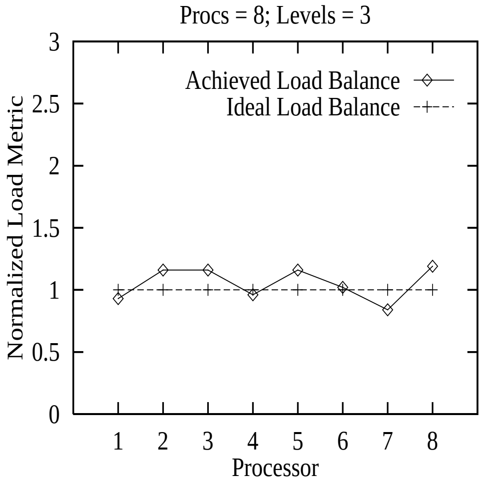

Figure 16: DAGH Distribution: Snap-shot II

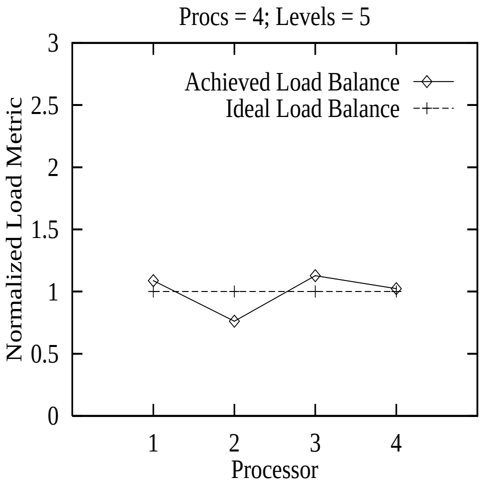

Figure 17: DAGH Distribution: Snap-shot III

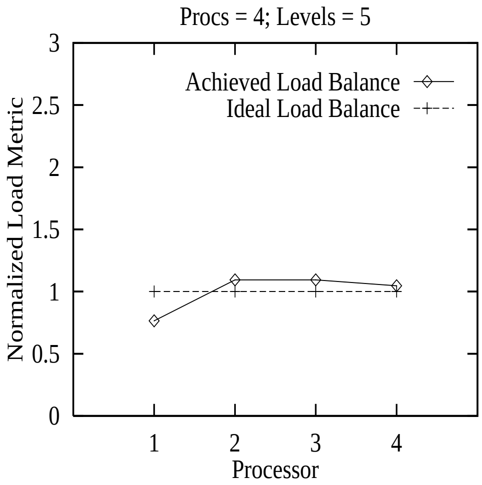

Figure 18: DAGH Distribution: Snap-shot IV

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.

4.1.3 Partitioning Overheads

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

Table 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 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.

4.2 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 motivatation for developing the SDDA.

The organization of the new finite element application is illustrated in Figure 19. 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.

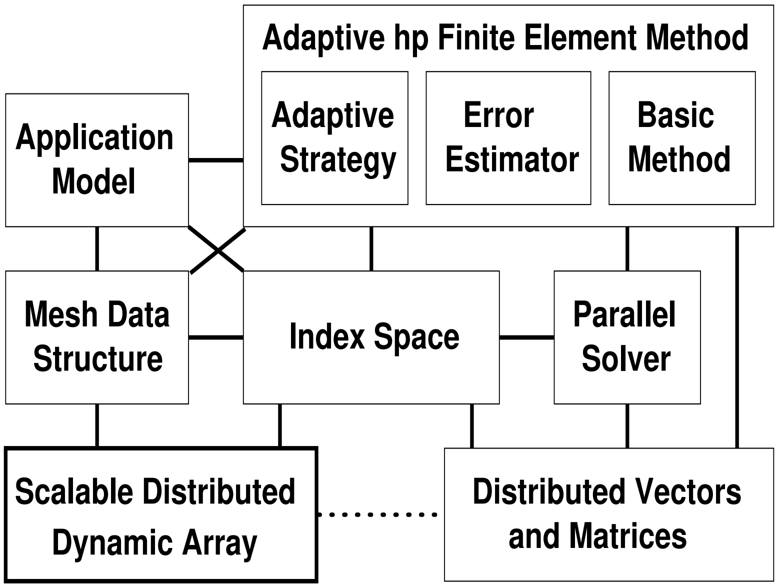

Figure 19: Finite Element Application Architecture

The finite element application uses a second parallel infrastructure which supports distributed vectors and matrices, as denoted in the lower right corner of Figure 19. The development of this infrastructure is also underway in TICAM. The current state of this infrastructure is presented in the Supercomputing'96 paper "Towards Usable and Lean Parallel Linear Algebra Libraries" by Robert van de Geijn, et. al. Both SDDA 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.

4.3 N-Body Problems

4.3.1 General Description

The N-body particle problems arising in various scientific disciplines appear to require an $\mathcal{O}(N^2)$ 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 [19], which, under certain assumptions, gives a method of $\mathcal{O}(N)$.

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

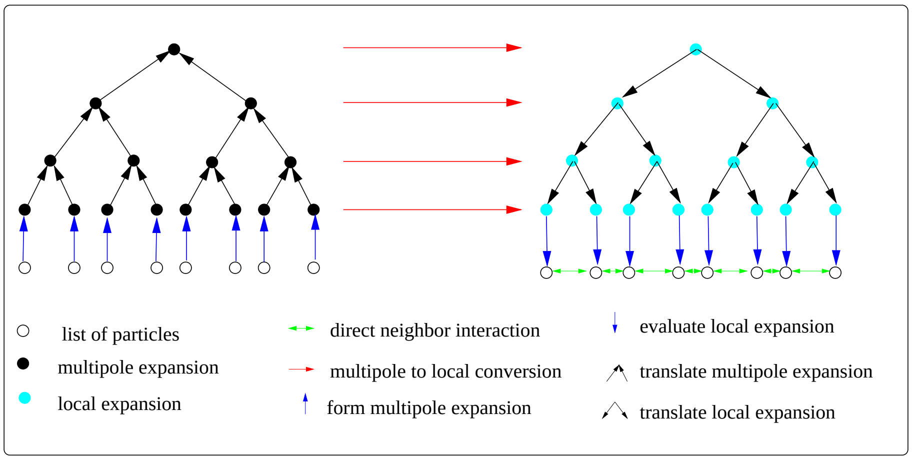

Figure 20: Data flow in the fast multipole algorithm

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 20). 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 [20, 21] and the references therein.

Figure 20 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.

4.3.2 Distributed Fast Multipole Results

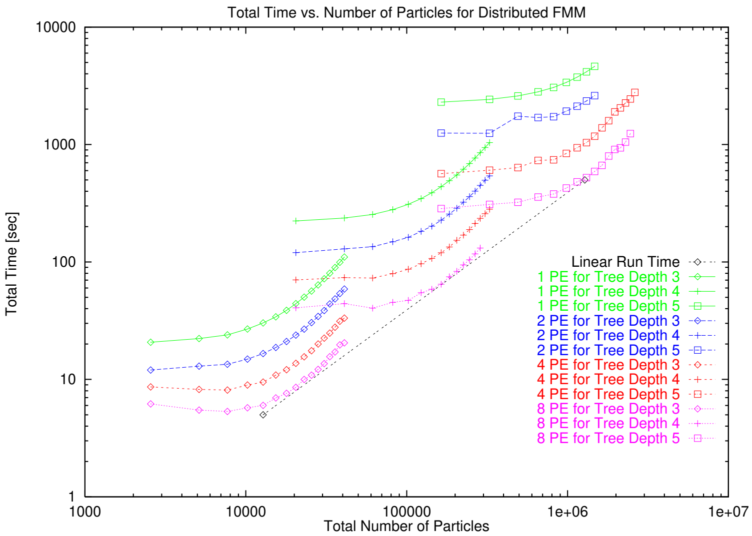

Figure 21: Total Execution Time on SP2 vs. Problem Size for Multiple Levels of Approximation

Next we describe preliminary performance results for the distributed fast multipole method implemented on the SDDA. We verified the fast multipole method by computing an exact answer for one particle chosen randomly. 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 21. 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 approximation 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 approximation and is fixed for each curve. The direct calculation time grows as O(N^2). The problem size for which these two times are equal represents the optimal number of particles stored per subdomain.

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.

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.

5 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 SDDA 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.

References

[1] M. S. Warren and J. K. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm", *Proceedings of Supercomputing'93*, Nov. 1993.

[2] Scott B. Baden, Scott R. Kohn, Silvia M. Figueria, and Stephen J. Fink, "The LPARX User's Guide v1.0", Technical report, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114 USA, Apr. 1994.

[3] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield, “Global Arrays: A Protable “Shared-Memory” Programming Model for Distributed Memory Computers”, *To be presented Supercomputing ‘94, Washington DC*, Nov. 1994.

[4] Max Lemke and Daniel Quinlan, “P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications”, *Lecture Notes in Computer Science, Parallel Processing: CONPAR 92 - VAPP V, Springer Verlag*, pp. 121–126, Sep. 1992, Proceedings of the Second Joint International Conference on Vector and Parallel Processing, Lyon, France.

[5] Rebecca Parsons, “A++/P++ Array Classes for Architecture Independent Finite Difference Computations”, *OON-SKI'94 - The Object-Oriented Numerics Conference, Sunriver, Oregon*, pp. 408–418, Apr. 1994.

[6] U. Geuder, M. Härdtner, B. Wörner, and R. Zink, “The GRIDS Parallel Programming Model for Grid-Based Scientific Applications”, *Proceedings of the Parallel Systems Fair,* 8^{th} *International Parallel Processing Symposium, Cancun, Mexico*, Apr. 1994.

[7] Hans Sagan, *Space Filling Curves*, Springer-Verlag, 1994.

[8] H. Samet, *The Design and Analysis of Spacial Data Structures*, Addison Wesley, Reading, Mass., 1989.

[9] H. Samet, *Applications of Special Data Structures*, Addison Wesley, Reading, Mass., 1990.

[10] M. S. Warren and J. K. Salmon, “A Parallel Hashed Oct-Tree N-Body Algorithm”, *in Supercomputing '93*, Portland, Oregon, Nov. 1993.

[11] John R. Pilkington and Scott B. Baden, “Partitioning with Spacefilling Curves”, CSE Technical Report CS94–349, Dept. of Computer Science and Engineering, University of California, San Diego, CA, 1994.

[12] H. F. Korth and A. Silberschatz, *Database System Concepts*, McGraw Hill, New York, 2 edition, 1991.

[13] W. Litwin, “Linear Hashing: a New Tool for File and Table Addressing”, *in Proceedings of the 6th Conference on VLDB*, pp. 212–223, Montreal, Canada, 1980.

[14] Marsha J. Berger and Joseph Oliger, “Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations”, *Jounal of Computational Physics*, pp. 484–512, 1984.

[15] 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.

[16] Manish Parashar and James C. Browne, “On Partitioning Dynamic Adaptive Grid Hierarchies”, *Proceedings of the* 29^{th} *Annual Hawaii International Conference on System Sciences*, **1**:604–613, Jan. 1996.

[17] Hanan Samet, *The Design and Analysis of Spatial Data Structures*, Addison - Wesley Publishing Company, 1989.

[18] J. Massó and C. Bona, “Hyperbolic System for Numerical Relativity”, *Physics Review Letters*, **68**(1097), 1992.

[19] Leslie Greengard, *The rapid evaluation of potential fields in particle systems.*, 1987.

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

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