SYSTEMS ENGINEERING FOR HIGH PERFORMANCE COMPUTING SOFTWARE: THE HDDA/DAGH INFRASTRUCTURE FOR IMPLEMENTATION OF PARALLEL STRUCTURED ADAPTIVE MESH REFINEMENT

MANISH PARASHAR AND JAMES C. BROWNE*

Abstract. This paper defines, describes and illustrates a systems engineering process for development of software systems implementing high performance computing applications. The example which drives the creation of this process is development of a flexible and extendible program development infrastructure for parallel structured adaptive meshes, the HDDA/DAGH package. The fundamental systems engineering principles used (hierarchical abstractions based on separation of concerns) are well-known but are not commonly applied in the context of high performance computing software. Application of these principles will be seen to enable implementation of an infrastructure which combines breadth of applicability and portability with high performance.

Key words. Software systems engineering, Structured adaptive mesh-refinement, High performance software development, Distributed dynamic data-structures.

1. Overview. This paper describes the systems engineering process which was followed in the development of the Hierarchical Dynamic Distributed Array/Distributed Adaptive Grid Hierarchy (HDDA/DAGH) program development infrastructure (PDI) for implementation of solutions of partial differential equations using adaptive mesh refinement algorithms. The term "systems engineering" was carefully chosen to distinguish the development process we propose as appropriate for development of high performance computing software from the conventional "software engineering" development process. The term "systems engineering" is not widely used in the vernacular of high performance computing. Indeed, formal structured development processes are not commonly used in development of high performance computing (HPC) software. This may be because conventional software engineering processes do not address many of the issues important for HPC software systems. This paper uses development of the HDDA/DAGH PDI as a case study to present a structured development process which addresses the issues encountered in development of high performance computing software systems. While HDDA/DAGH is a PDI for applications rather than an application, the issues addressed by the systems engineering process we describe are common to all types of high performance computing software systems. We propose this systems engineering process as one which is generally appropriate for high performance computing applications. Conventional software engineering [1,2] focuses on the management aspects of the development process for very

* Department of Computer Science & TICAM, University of Texas at Austin, Austin, Texas 78712 {browne,parashar}@cs.utexas.edu

large systems which have many components, multiple developers involved in system development, and structures and processes which enable effective management of the development process. These large systems are often focused on information management for commercial or defense applications. (Large embedded controllers for medical instruments, power systems, etc. are also targets for a somewhat different family of software engineering methods.) The usual requirements for these information management systems include: high availability, good response for interactive transactions and maintainability over long lifetimes. To achieve these goals over very large systems the work of many developers must be coordinated to yield a coherent system structure. These systems are typically implemented for commodity hardware based execution environments using commodity software systems as the implementation infrastructure. There is a substantial body of "conventional wisdom" concerning how to realize efficient instantiations of these systems. (Although the rise of distributed or client/server implementations has introduced a new set of performance concerns.) The primary source of complexity is primarily sheer system size. Conventional software engineering methods and processes are structured by this set of requirements and issues.

High performance systems are typically quite different from these information management systems. They are often of modest size by commercial standards but typically have a high degree of internal complexity. HPC applications are usually developed by small teams or even individuals. There is no commodity implementation infrastructure to be used. The execution environments are state-of-the-art, rapidly changing, and frequently parallel computer systems. The underlying hardware is a often novel architecture for which there is little "conventional wisdom" concerning development of efficient programs. These execution environments change much more rapidly than is the case for large commercially-oriented systems. The end-user requirements for HPC software systems typically evolve even more rapidly because they are used in research environments rather than in production environments. Time for end-to-end execution (absolute performance) is usually the most critical property with adaptability to a multiplicity of applications and portability across the rapidly evolving platforms being other important issues. Reuse of previously written code is also often desired. The complexity of HPC systems primarily arises from the data management requirements of the applications. We conclude that traditional methods of software engineering are not appropriate for development of high performance computing software. However, high performance computing software development can benefit from the application of a well-structured development process.

The systems engineering process we propose targets issues and requirements underlying the development of high performance computing software systems. In what follows we describe the systems engineering process which we followed in the development of the HDDA/DAGH system and demonstrate that the result is a system which is performant, adaptable and portable. Application of well-structured development processes to high performance computing software will be beneficial to the field in general. If HPC is to become an effective discipline we must document good practice so that best practice can be identified. This is particularly true for developers of infrastructure systems which are intended to be used by a broad community of users. This paper uses the development of HDDA/DAGH as a vehicle to put forward what we think is one example of good design/development process for HPC software systems.

2. Systems Engineering of High Performance Software Systems. The systems engineering process for high performance computing software development described here has four elements:

1. A process for the translation of application requirements into system design requirements. This step is often complex, iterative and is actually never finished since the applications requirements for research-oriented systems typically evolve rapidly and continually.
2. A design model founded on the principles of hierarchical abstraction and separation of concerns. Use of hierarchical abstractions in software system development was formalized by Dijkstra in 1968 [3]. But development of effective abstraction hierarchies is not simple. We propose that definition and realization of effective abstraction hierarchies should be based on the principle of separation of concerns [4,5]. Construction of abstraction hierarchies based on separation of concerns is discussed in detail in Section 2.2. Satisfaction of the requirements for absolute performance, adaptability and portability are grounded in the structure of the design model, and in defining abstractions which enable selection of efficient algorithms.
3. Selection of implementation algorithms which meet the goals of system performance in the context of the design model.
4. An implementation model which preserves the structure and properties of the design model in the implementation.

Each of these steps will be described in more detail below and illustrated application to the development of HDDA/DAGH in Section 3.

2.1. Translation of Application Requirements to System Design Requirements. This is an iterative process which is rendered more complex by the cultural, vocabulary and experiential differences between computational scientists/computer scientists who are the typical system developers, and physicists and engineers who are the typical application developers for HPC software systems. It is often the case that the application-level developers have not done (and indeed cannot do) a systematic a priori analysis of requirements. The applications which are being supported often involve new problems for which solution methods are not known, and new algorithms which are being used by application scientists for the first time.

It is therefore impossible for them to define the requirements precisely, and unreasonable for the computational or computer scientist to expect a static and complete requirements specification before beginning development. Consequently the requirements specification for an HPC software system is an evolving document. What must be agreed upon is the process by which the end-users and the software system developers actively collaborate. Usually an initial requirements statement is negotiated and an initial design and implementation of the PDI is created. Application developers then try this initial implementation and come back with an additional set of requirements based on their experience using the software system to attempt problem solution (and to experiment with new ideas recently discovered).

There are, however, some generic requirements for the implementation of infrastructures such as the HDDA/DAGH PDI. The application programming interface of the PDI should be as close as possible to the direct representation of operations in the algorithms of the solution method. The desired application programming interface usually includes the ability to reuse existing Fortran or C modules, to make both dynamic data structuring and distribution of data across multiple address spaces transparent and, at the same time, to lose nothing in the way of efficiency compared to a low-level, detailed application-specific implementation. Secondary requirements include portability of the resulting system across a wide variety of experimental platforms and scalability from small problems to very large problems.

2.2. The Design Model. The design model which we have adopted is the usual one of deriving requirements from the top down but designing the system from the bottom up as a set of hierarchically structured layers of abstractions. Critical factors underlying the development of effective hierarchical abstractions are:

1. Separation of Concerns - Create a clean separation of semantic content between levels.
2. Keep the semantic distance between levels as small as is consistent with not introducing too much overhead.
3. Direct Mapping - Define layers which implement the requirements of the higher levels as directly as is consistent with efficiency. Avoid complex protocols across levels.

Figure 2.1 is a schematic of the design model for the HDDA/DAGH PDI. Each layer can be thought of as a set of abstract data types which implements operations against instances of the structures they define.

The lowest level of the abstraction hierarchy of the HDDA/DAGH PDI defines a hierarchical dynamic distributed array or HDDA which is a generalization of the familiar static array of programming languages. The HDDA is purely an array data type and only has the operations of creation, deletion, array expansion and contraction, and array element access

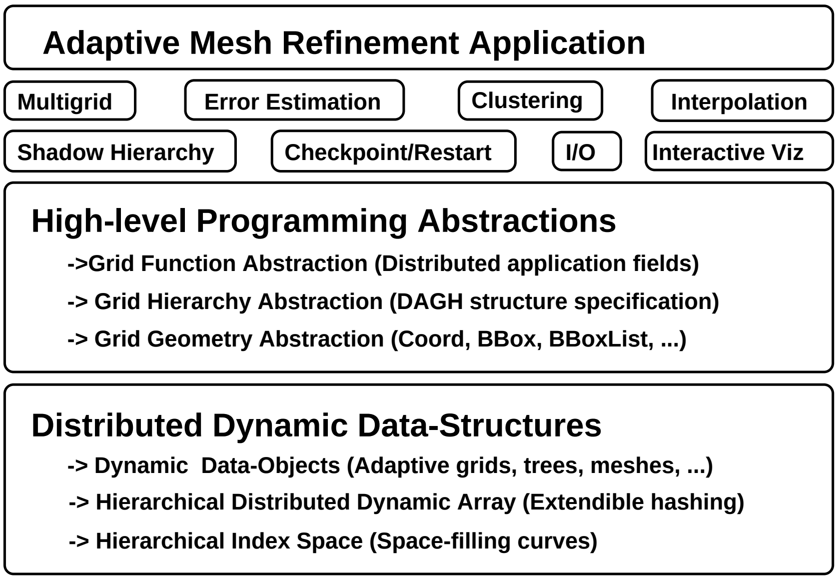

FIG. 2.1. *Design model for the HDDA/DAGH Program Development Infrastructure*

and storage defined on it. Further, since the use of computational kernels written in C and Fortran is a requirement, partitioning, communication, expansion and contraction must be made transparent to these computational kernels. Separation of concerns is illustrated by the fact that we define a separate level in the hierarchy (above the HDDA) hierarchy to implement grids and/or meshes. We shall see that defining the HDDA as a separate abstraction layer gives material benefit by making definition of multiple types of grids and meshes simple and straightforward.

The next abstraction level implements grids by instantiating arrays as a component of a larger semantic concept, that of a grid. A grid adds definition of a coordinate system and computational operators defined in that coordinate system. The definition of a grid includes the operations of creation, deletion, expansion and contraction which are directly translated to operations on instances of the HDDA, and also defines computational (stencil) operators, partitioning operators, geometric region operators, refinement and coarsening operators, etc. Creation of a hierarchical grid is directly mapped to creation of a set of arrays. Since arrays are implemented separately from grids, it is straightforward to separately implement many different variants of grids using the same array abstractions which are provided. Thus separation of concerns spreads vertically across higher levels of the abstraction hierarchy leading to simpler, faster and more efficient implementations.

If the HDDA maintains locality and minimizes overheads then the DAGH level can be focused on implementing a wide span of grid variants. Since each grid variant can be defined independently of the other grid types without redundancy, and must implement only the computational operations unique to its specific grid type, each grid variant can have a simple and efficient implementation.

Hierarchical abstractions are a recursive concept. The HDDA is itself a hierarchy of levels of abstractions.

2.3. Algorithms for Implementation of the Design Model. Each level of the design model will be implemented as a set of abstract data types. Therefore algorithms for implementing the operations of each abstract type on its instances must be selected. Separation of concerns enables selection and/or definition of the simplest algorithms which can accomplish a given requirement. Separation of concerns in the design model thus leads to performant, scalable, adaptable and portable code.

The critical requirement for the HDDA/DAGH package is to maximize performance at the application level. Performance at the application level requires locality of data at the data management level. Locality not only minimizes communication cost on parallel systems but also maximizes cache performance within processors. Since the application levels operators (operations on the grids) are defined in an n-dimensional application space, it is critical that the locality of the data in the one-dimensional distributed physical storage space maintains the locality defined by the geometry of the problem in the n-dimensional coordinate space in which the solution is defined. Therefore we must choose or define storage management algorithms which lead to preservation of the multi-dimensional geometric locality of the solution in the physical layout of data in storage. A second factor in obtaining high performance is minimization of overhead such copying of data, communication, etc. Therefore our algorithm choices for the HDDA must focus on minimizing these overheads.

2.4. Structure for the Implementation Model. The implementation model must preserve the structure and properties of the design model clearly in the implementation. The implementation model which we chose is a C++ class hierarchy where a judicious integration of composition and inheritance is used to lead to a structure which captures and accurately reflects the hierarchical abstractions in the design model. This structure will be seen in the next section to closely follow the design model.

3. Case Study - Design and Implementation of the HDDA/DAGH Infrastructure. The foundations for HDDA/DAGH originated in the development of a similar infrastructure for support of hp-adaptive finite element computational methods which was begun in 1991 [6]. Thus development of application requirements extends over some seven years. The mapping to design requirements also owes intellectual debts to other projects and contributors as noted in the acknowledgments.

3.1. Translation of Application Requirements to System Design Requirements. The HDDA/DAGH infrastructure was initially developed to support the computational requirements of the Binary Black Hole (BBH) NSF Grand Challenge project beginning in 1993. The BBH project had already settled on using the Berger-Oliger AMR algorithm [7] as its means of coping with the rapid changes in the solution of Einstein's equations in the vicinity of a black hole. Support for several variants of multigrid solution methods was also a requirement. HDDA/DAGH has later been adapted and extended to support several other applications. A summary of these extensions and adaptations will be given in Section 4. The breadth of these extensions and the ease with which they were made is a vindication of the time and care spent in the conceptualization and early design phases of development of the HDDA/DAGH package. The initial requirements were developed by a sequence of meetings between the physicists at the University of Texas at Austin formulating the solution of Einstein's equations for the evolution of a BBH and the authors of this paper. These meetings were held more or less regularly for about a year, and were later expanded to include representatives of the other research groups in the BBH consortium. The requirements specification process actually took place over a period of about three years and spanned several releases of the software. A number of major new requirements developed as the user community worked with the software. For example the need for shadow hierarchies to enable error analysis was not discovered until the physicists began coding with the early version of HDDA/DAGH. In fact, it was not until about February, 1997 that definition of the core capabilities of HDDA/DAGH was truly finalized. (And we are sure that this definition will not be valid for any extended period of time.) Translation of support for Berger-Oliger AMR and multigrid into definition of hierarchical layers of abstract data types with highly efficient execution and convenience features such as built-in visualization and checkpointing and restart define the highest level of application requirements. The specific application requirements for a parallel implementation of Berger-Oliger adaptive mesh refinement based solutions of partial differential equations is support for dynamic hierarchical grids and in particular dynamic hierarchical grids which may be distributed across multiple address spaces. The grid must be adapted based on the estimated error in the solution. Both coarsening and refinement is required. The implementation of dynamic data management must be sufficiently efficient so that the benefits of adaptivity are not out-weighted by the overheads of dynamic storage management. Efficient implementation of dynamic and distributed data management implies that the locality properties of the application geometry be preserved in the storage layout across distributed and expansion and contraction. Many different grid types and computational operators must be supported. Reuse of existing Fortran and C coded computational routines was desired.

3.2. Instantiation of the Design Model for HDDA/DAGH. Figure 3.1 is a schematic of the layers of the HDDA/DAGH abstraction hierarchy in the context of an application. The hierarchy descends from left to right (compare with Figure 2.1) and thus the functionality becomes more generic from left to right. Each level of the hierarchy is given in more detail from top to bottom. An application consists of application specific components (stencil operators, solvers, interpolation operators, etc.). These application specific components are mapped to operations on an appropriate subtype on its right in the programming abstractions layer. The grid subtypes are mapped to the types implemented by the dynamic data management layers to their right. Definitions both across and within layers have been strongly influenced by the principle of separation of concerns.The ovals with lighter shading are specific instances of the higher level with darker shading. The Grid Structure abstractions defines hierarchical grid and implement standard operations on these grids. These abstractions represent the structure geometry of the computational domain and add grid semantics to the instances of the HDDA in which the computational data is stored. It is straightforward to define multiple types and instances of Grid Hierarchies such as a SAMR hierarchy or a multigrid hierarchy. The Grid Function abstractions define applications fields on the Grid Structure. These abstraction define the data storage associated with a grid structure and can be defined on the cells, vertices or faces of each grid in the hierarchy. This separation of Grid Function from Grid Hierarchy enables the structure of the computational domain to be defined and manipulated independent of the computational data. This makes it simple for users to employ computational operators for a wide spectrum of computational methods. The Grid Geometry abstractions represent regions in grid space are independent of grid type. These abstractions provide a uniforms means for interacting with grids and addressing and directing computations to regions on grid. Separation of geometric specifications from computational operations allows a single implementation of geometry operators to be applied to all grids.

The requirements for the HDDA are derived from the requirements for the dynamic hierarchical grids:

1. There must be a connected hierarchy of arrays to represent a grid hierarchy.
2. The hierarchy of arrays must be dynamic.
3. The arrays must be partitioned across separate address spaces.
4. The partitioning must result in a balanced computational load across processors.
5. Pure static single address space array semantics must be maintained on a local basis since Fortran and C coded computational routines must continue to execute correctly. Consequently access

to array element values must be transparent to their distribution across address spaces and their dynamics.

6. Computational operations must be efficient even when the grid is dynamic and partitioned across many processors.

Separation of concerns applies vertically within the definition of the HDDA 3.2. The first three requirements suggest separation of logical structure from physical structure. Partitioning, expansion and contraction and access are defined on the logical structure of the HDDA (the index space) and mapped to the physical structure implementing storage. The HDDA is therefore composed of three abstractions: index spaces, storage and access. Index spaces define the logical structure of the hierarchical array while storage defines the layout in physical memory. Access virtualizes the concept of accessing the value stored at a given index space location across hierarchy and partitioning.

Definitions:.

Index Space An index space is a lattice of points in an n-dimensional discrete space. We need a recursively defined hierarchical index space where each position in an index space may be an index space.

Storage A mapping from the n-dimensional index space to a one dimensional physical storage.

Access A set of operations for returning the values associated with positions in the index space from the associated storage.

The connection between the application and the array is defined by a mapping from points in the n-dimensional continuous physical coordinate space in which the solution is defined, to points in the n-dimensional discrete index space. Expansion, contraction and partitioning are implemented as operations on the index space. Storage maps the index space to storage space. Note that separation of the index space from storage allows for multiple mappings from index space to physical space where each mapping defines a different semantics for the same data. For example, application of the computational operators is defined by one mapping while the data which must be communicated among partitions is defined by another mapping. The objects of visualization are defined by yet a third mapping. This instance of separation of concerns has enabled visualization to be driven by the same data as is used for the computation. Access, the third abstraction, implements the operations required for transparency of access across physically distributed address spaces. It must be disjoint from storage and index spaces because the implementation must vary with execution environment. This is the second application of separation of concerns in the design of the HDDA.

3.3. Algorithms for the Abstractions of the Design Model. The requirement at this point is to identify algorithms:

1. for mapping the n-dimensional continuous space of the solution to an n-dimensional hierarchical index space with preservation of

application locality,
2. for mapping the hierarchical n-dimensional index space to a one-dimensional physical storage space with preservation of the locality in the index space to locality in the storage space,
3. for segmenting the storage space into efficiently manageable blocks which can be accessed, expanded and contracted efficiently with preservation of locality,
4. for partitioning of the grid hierarchy among processors and communication of the overlapping regions of the grid among processors, and
5. for refinement and coarsening of the grid.

3.3.1. Hierarchical, Extendible Index Space. The hierarchical, extendible index space component of the HDDA is derived directly from the application domain using space-filling mappings [8] which are computationally efficient, recursive mappings from N-dimensional space to 1-dimensional space. Figure 3.3 illustrates a 2-dimensional Peano-Hilbert curve. The solution space is first partitioned into segments. The space filling curve then passes through the midpoints of these segments. Space filling mapping encode application domain locality and maintain this locality though expansion and contraction. The self-similar or recursive nature of these mappings can be exploited to represent a hierarchical structure and to maintain locality across different levels of the hierarchy. Space-filling mappings allow information about the original multi-dimensional space to be encoded 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. The index-space is used as the basis for application domain partitioning, as a global name-space for name resolution, and for communication scheduling.

3.3.2. Mapping to Address Space. The mapping from the multi-dimensional index space to the one-dimensional physical address space is accomplished by mapping the positions in the index space to the order in which they occur in a traversal of the space filling curve. This mapping can be accomplished with simple bit-interleaving operations to construct a unique ordered key. This mapping produces a unique key set which defines a global address space. Coalescing segments of the linear key space into a single key, blocks of arbitrary granularity can be created.

3.3.3. Storage and Access. Data storage is implemented using extendible hashing techniques [9] to provide a dynamically extendible, globally indexed storage (see Figure 3.4). The keys for the Extendible Hash Table are contractions of the unique keys defined as described preceding. Entries into the HDDA correspond to DAGH blocks. Expansion and contraction are local operations involving at most two buckets. Locality of data is preserved without copying. The HDDA data storage provides a means for efficient communication between DAGH blocks. To communicate data to another DAGH blocks, the data is copied to appropriate locations in the HDDA. This information is then asynchronously shipped to the appropriate processor. Similarly, data needed from remote DAGH blocks is received on-the-fly and inserted into the appropriate location in the HDDA. Storage associated with the HDDA is maintained in ready-to-ship buckets. This alleviates overheads associated with packing and unpacking. An incoming bucket is directly inserted into its location in the HDDA. Similarly, when data associated with a DAGH block entry is ready to ship, the associated bucket is shipped as is. The overall HDDA/DAGH distributed dynamic storage scheme is shown in Figure 3.5.

3.3.4. Partitioning and Communication. An instance of a DAGH is mapped to an instance of the HDDA. The granularity of the storage 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. In case of an AMR scheme, computational load is determined by the number of grid elements contained in the block and the level of the block in the AMR grid hierarchy. The former defines the cost of an update operation on the block while the latter defines the frequency of updates relative to the base grid of the hierarchy. Note that in the representation described above, space-filling mappings are applied to grid blocks instead of individual grid elements. The shape of a grid block and its location within the original grid is uniquely encoded into its space-filling index, thereby allowing the block to be completely described by a single index.

Partitioning a DAGH across processing elements using this representation consists of appropriately partitioning the DAGH key 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.

3.3.5. Refinement and Coarsening of the Grid. The DAGH representation starts with a single HDDA corresponding to the base grid of the grid hierarchy, and appropriately incorporates newly created grids within this list as the base grid gets refined. The resulting structure is a composite key space of the entire adaptive grid hierarchy. Incorporation of refined component grids into the base grid key space is achieved by exploiting the recursive nature of space-filling mappings. For each refined region, the key list corresponding to the refined region is replaced by the child grid's key 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. Each DAGH block in this representation is fully described by the combination of the space-filling index corresponding to the coarsest level it contains, a refinement factor, and the number of levels contained

3.4. Implementation Model for HDDA/DAGH. The success of a system design model is ultimately determined by the ability to preserve this design through implementation. Quite often, important design features like modularity and extensibility are lost in naive monolithic implementations. The design approach based on the principles of separation of concerns and hierarchical abstractions enables a direct coupling of the design with object-oriented software development technology to preserve all features of the design in the implementation. Hierarchical abstraction defines the structure of the object-oriented class hierarchy, while the separation of concerns across the abstractions leads to clean interfaces in the class structure. The actual implementation uses *C++* and builds the class hierarchy from bottom up using inheritance and composition to specialize more general base classes.

A subset of the HDDA/DAGH abstraction shown in Figure 3.6 is used to illustrate the preservation of design through implementation. This figure shows the portion of the hierarchy that designs a *Grid Function* which is an application field such as pressure or temperature defined on the computational *Grid Hierarchy*. The first application of the principle of separation of concerns separates the structure of the grid hierarchy from the storage associated with the field. The structure of the grid hierarchy is then defined as the combination of the hierarchy index space (which is derived from the application domain using space-filling mappings) and the *Grid Geometry* operators such as regions (bounding boxes) and points (coordinates). The storage is implemented as an HDDA which is separated into the hierarchical index space and extendible hashing bucket storage.

The actual *C++* class hierarchy corresponding to this abstraction hierarchy is shown in Figure 3.7. The classes that make up the base of the hierarchy include:

- A *Buckets* class structure that implements generic data buckets and bucket iterators. Buckets are specialized into single (or stand alone) buckets and packed buckets which combine multiple single buckets.
- An *Index-Space* class structure which implements the hierarchical extendible, index space based on space-filling mappings. The class structure starts with a simple bit vector of arbitrary length (class *BitVec*) and specializes this class into a space-filling index (class *sfcIndex*) by applying bit interleaving to it. Classes *PeanoHilbert* and *Morton* apply bit transformation defined by the Peano-Hilbert and Morton space-filling mapping algorithms respectively to the sfcIndex to generate the appropriate index-space.
- A *Grid Geometry* class structure which implements points and re-

gions in the computational domain. Class *Coords* implements an arbitrary point in the domain while class *BBox* implements a region as a combination of a lower bound Coords and an upper bound Coords.

Storage for the Grid Function is built on the HDDA structure which is implemented as a composition of the index-space and bucket class structures. HDDA objects are then specialized via inheritance to implement *GridData* objects by adding grid access semantics. The *Grid Structure* is implemented as the *GridHierarchy* class which specializes individual *GridComponents*. The GridComponent class implements a single component grid in the grid hierarchy as a span(s) of the index-space which geometry operators defined on it. GridHierarchy combines multiple component grids into the hierarchical SAMR grid structure and defines operators on this structure. Finally, storage (*GridData*) and structure (*GridHierarchy*) are combined by the *GridFunction* class to implement application grid functions. It can be seen from Figures 3.6 and 3.7 that design structure derived using our design model based on separation of concerns and hierarchical abstractions directly complements its implementation class hierarchy, thereby preserving all the attributes of the design in the implementation.

4. Applications of HDDA/DAGH. Figure 4.1 illustrates the spectrum of application codes and infrastructures enabled by HDDA/DAGH. Three different infrastructures targeting computational grand challenges use HDDA/DAGH as their foundation: (1) a computational infrastructure for the binary black-hole grand challenge, (2) a computational infrastructure for the neutron star grand challenge and (3) IPARS: a problem solving environment for parallel oil reservoir simulation. Applications codes developed using the HDDA/DAGH data-structures and programming abstractions includes general relativity codes for black-hole, boson star and neutron star interactions, coupled hydrodynamics and general-relativity codes, laser plasma codes, and geophysics codes for adaptive modeling of the whole earth. HDDA/DAGH is also used to design a multi-resolution data-base for storing, accessing and operating on satellite information at different levels of detail. Finally, base HDDA objects have been extended with visualization, analysis and interaction capabilities. The capabilities are then inherited by application objects derived from HDDA objects and provide support for a framework for interactive visualization and computational steering where visualization, analysis and interaction is directly driven from the actual computational objects.

5. Related Work. The authors are not aware of any closely related work. There are hundreds if not thousands of substantial and effective programs which successfully implement "high performance computations". But there are relatively few infrastructures for support of implementation of high performance parallel computations. Of course, the designers and developers of other packages [10,11,12] for support of implementation of adaptive mesh refinement methods have all had to face and overcome the issues and concerns of designing to performance and designing for extensibility. But for the most part the process of design for these systems has largely gone unrecorded. If HPC is to be an effective discipline we must document good practice so that best practice can be identified. The most closely related body of research is that of "Software Architectures" [13]. Our design models are closely related to software architectures. But little attention has been paid to "software architectures" for high performance computing. Smith and Browne [14] and Smith [15] have defined a discipline of performance engineering for information management software systems. There are thousands of books and papers on conventional software engineering. Fundamental concepts such as hierarchical structuring and orderly process are well covered in the standard books ([1,2] among others). But the particular concerns of HPC software systems are not covered.

6. Conclusions. It has been shown that at least for the HDDA/DAGH PDI that following a development model and process which targets the issues important for software systems for high performance software systems contributed to attainment of a system which has been demonstrated to meet its requirements and to have desirable properties with respect to extensibility and portability. We suggest that formulation and application of appropriate development models and reporting on the results of use of good development models will benefit the community concerned with development of high performance software systems.

7. Acknowledgments. There are many contributors to the development of HDDA/DAGH. Carter Edwards was an major contributor to the concepts of the HDDA. The authors are grateful the community of users who endured the painful process of developing the requirements and using early versions of admittedly incomplete versions of HDDA/DAGH. In particular we are grateful to Richard Matzner, Matt Choptuik, Ed Seidel, Paul Walker, Joan Masso, Greg Cook, Tom Haupt, and Geoffrey Fox. Financial support has come from NSF ACS/PHY grant 9318152 (ARPA supplemented) and from DARPA/CSTO contract 3531427 through a subcontract from Syracuse University and from Argonne National Laboratory through the Enrico Fermi Fellowship to Manish Parashar.

REFERENCES

[1] Ian Sommerville, *Software Engineering*, Addison Wesley, 1996.
[2] Roger Pressman, *Software Engineering: A Practitioner,s Approach*, MacGraw-Hill, NY, NY, 1987.
[3] E. W. Dijkstra, "The Structure of the THE Operating System", *CACM*, pp. 341–346, Nov. 1968.
[4] James C. Browne, "A Language for Specification and Programming of Reconfigurable Parallel Computation Structures", *Proceedings of the Internation Conference on Parallel Processing, Bellaire, MI*, Aug. 1982.

[5] RAJU PANDE AND JAMES C. BROWNE, "A Compositional Approach to Concurrent Object-Oriented Programming", *Proceedings of the Internation Conference on Compilers and Languages, Paris, France*, May 1994.

[6] CARTER EDWARDS AND JAMES C. BROWNE, "Scalable Distributed Dynamic Array and its Application to a Parallel hp-Adaptive Finite Element Code", *Presentation at Parallel Objec-Oriented Methods and Applications Workshop, Santa Fe, NM*, Feb. 1996.

[7] MARSHA J. BERGER AND JOSEPH OLIGER, "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations", *Jounal of Computational Physics*, pp. 484–512, 1984.

[8] HANS SAGAN, *Space-Filling Curves*, Springer-Verlag, 1994.

[9] R. FAGIN, "Extendible Hashing - A Fast Access Mechanism for Dynamic Files", *ACM TODS*, **4**:315–344, 1979.

[10] 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.

[11] S. J. FINK, S. R. KOHN, AND S. B. BADEN, "Flexible Communication Mechanisms for Dynamic Structured Applications", *Proceedings of IRREGULAR '96, Santa Barbara, CA*, Aug. 1996.

[12] 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.

[13] MARY SHAW AND DAVID GARLAN, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[14] C. SMITH AND JAMES C. BROWNE, "The Structure of the THE Operating System", *Proceedings of NCC'82, AFIPS Press, NY, NY*, pp. 217–224, 1982.

[15] C.U. SMITH, *Performance Engineering of Software Systems*, Addison Wesley, 1990.

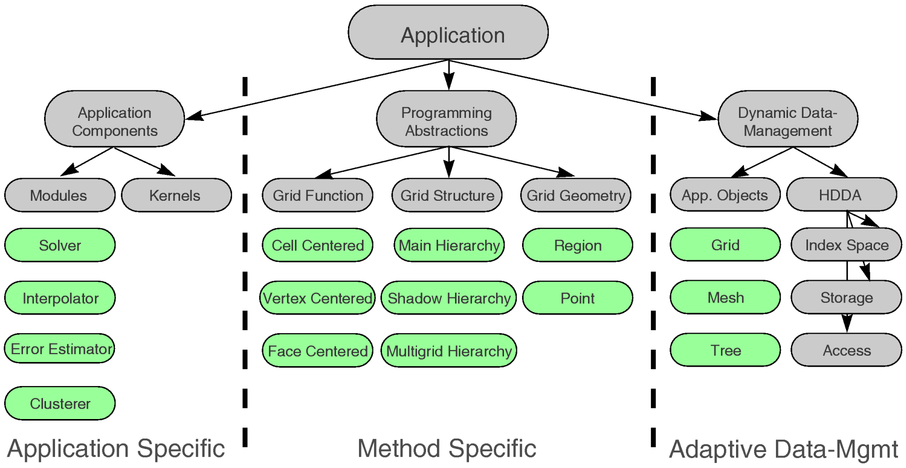

Fig. 3.1. *HDDA/DAGH Abstraction Hierarchy*

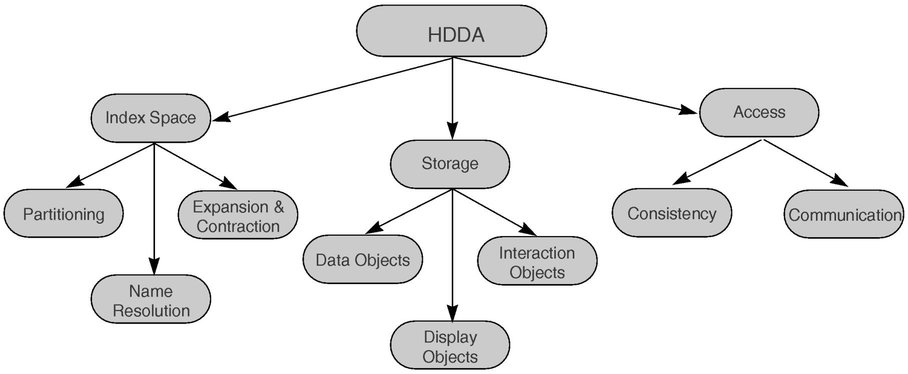

Fig. 3.2. *Hierarchical Abstractions of the HDDA*

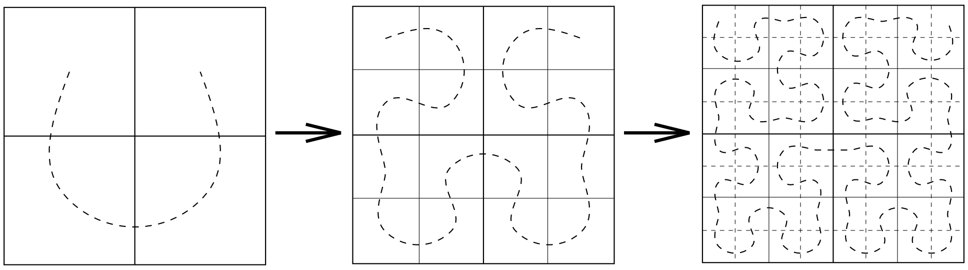

FIG. 3.3. *Hierarchical Space-Filling Mappings*

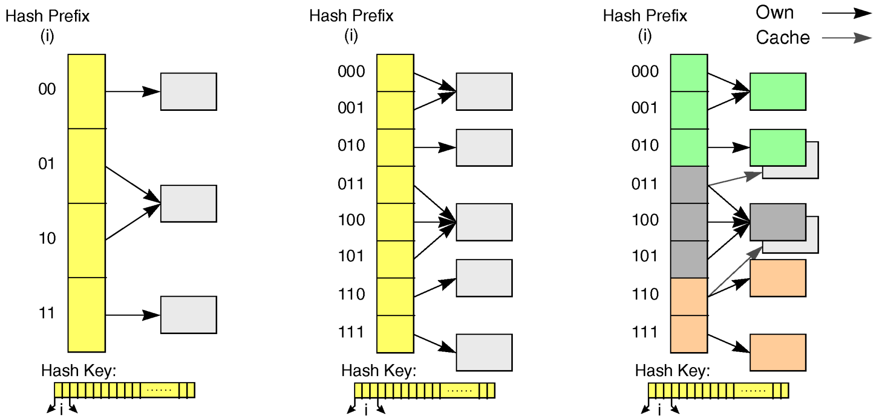

FIG. 3.4. *Extendible hashing for distributed dynamic storage and access*

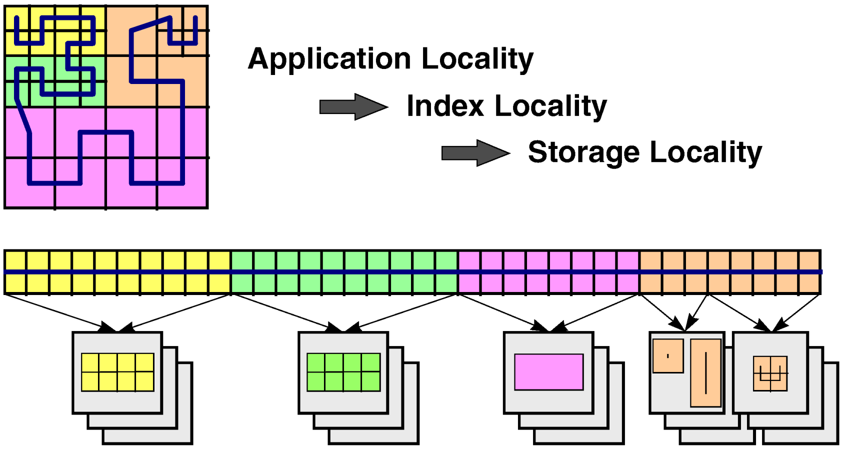

Fig. 3.5. *HDDA/DAGH distributed dynamic storage*

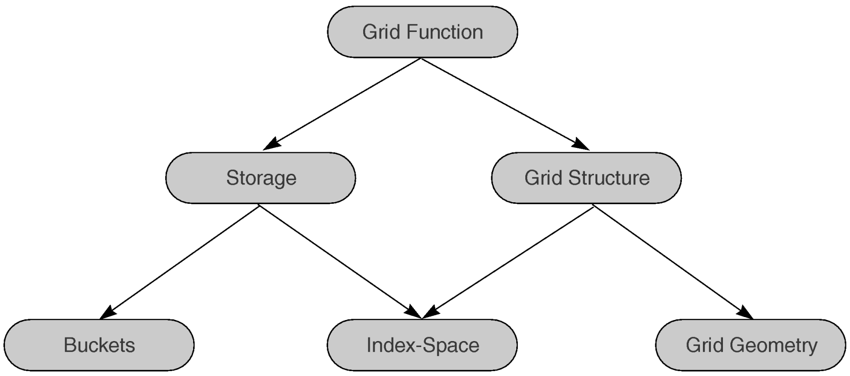

FIG. 3.6. *Preserving Design in Implementation: Hierarchical Abstraction*

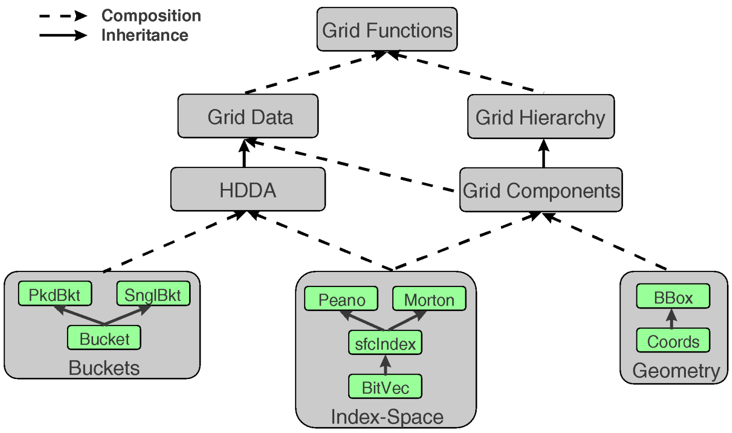

FIG. 3.7. *Preserving Design in Implementation: Object Oriented Implementation Hierarchy*

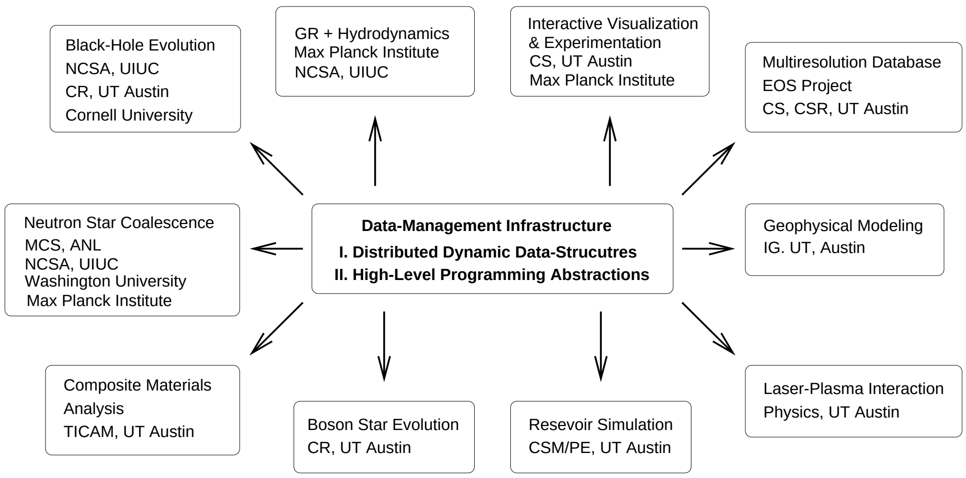

Fig. 4.1. *HDDA/Applications*