Integrated Data-Management for Computational Steering

Manish Parashar and James C. Browne
Department of Computer Sciences
TICAM, Taylor Hall 2.400
University of Texas at Austin
Austin, TX 78712
Tel: 512-471-2160; Fax: 512-471-8694
{parashar, browne}@cs.utexas.edu

Contents

Integrated Data-Management for Computational Steering

Manish Parashar and James C. Browne

Department of Computer Sciences

University of Texas at Austin

{parashar, browne}@cs.utexas.edu

Abstract

This paper describes a computational steering system for parallel adaptive applications where the computation, visualization and steering are driven from a common data-management infrastructure, and where the operation of visualization and interaction are executed (and parallelized) as far as possible on the computational engines used for the application itself. The paper consists of three components: (1) a description of the interaction and steering requirements for a family of parallel adaptive, scientific and engineering applications, (2) a systems engineering methodology (based on a separation of concerns across hierarchically defined abstractions) for high performance computing (HPC) software that enables integration of visualization, analysis, interaction and steering within computational data objects, and (3) application of this methodology to the design and implementation of HDDA/DAGH, a data-management infrastructure for parallel adaptive mesh-refinement and multigrid techniques.

1 Overview

This paper first describes the significance and requirements for interactive experimentation and computational steering for a family of parallel adaptive algorithms that underlie most large-scale scientific and engineering simulations. These methods use dynamic adaptation of the computational domain to concentrate additional computational effort and resources to only those regions in the domain that require it. The methods yield highly advantageous cost/accuracy ratios and can enable larger simulations at higher resolutions compared to static methods for a given set of resources. This paper concentrates on adaptive mesh-refinement (AMR) techniques from this family of adaptive algorithms, although the design approach presented is applicable to other techniques from the family [1]. Example AMR based applications include simulation of black-hole interactions, geophysical modeling of the whole earth, and oil reservoir and porous media simulations.

The paper then describes a systems engineering methodology for high-performance computing (HPC) application software development that can be used to design common distributed and dynamic data-management that can efficiently support the integration of parallel adaptive computations with interaction, visualization, analysis, and steering. Execution of interaction, visualization, and analysis operations on the same data-structures as the computations that are being controlled through steering (whether external or a mixture of external and algorithmic) offers many advantages. Runtime control operations can now be directly performed on applications objects such as grids, meshes and trees through the view provided by the their visualization. Further, parallelization of interaction and analysis operations becomes straightforward since data already resides in distributed data-structures.

A critical requirement for enabling the definition and execution of multiple types of operations on a common data management infrastructure is an underlying abstraction hierarchy which enables definition of multiple classes of methods on a single instance of stored data. The systems engineering methodology presented in this paper achieves this by defining hierarchical abstractions based on a clean separation of concerns across system functionality.

The final portion of this paper describes the application of the systems engineering methodology to the design and development of the Hierarchical Distributed Dynamic Array/Distributed Adaptive Grid Hierarchy (HDDA/DAGH) program development infrastructure (PDI) for implementation of solutions of partial differential equations using adaptive mesh-refinement algorithms. HDDA/DAGH integrates computations with interaction and visualization and is being used to enable a wide spectrum of scientific and engineering applications. Application of the interactive steering capabilities of the HDDA/DAGH PDI to large-scale simulations is currently under evaluation and will be included in the final paper.

2 Hierarchical Adaptive Mesh-Refinement

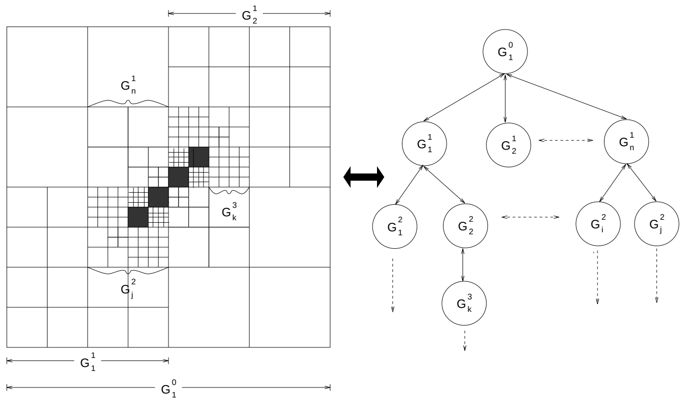

Figure 1: Adaptive Grid Hierarchy - 2D (Berger-Oliger AMR Scheme)

Parallel adaptive algorithms based on hierarchical adaptive mesh-refinement (AMR) achieve dynamic adaptivity by tracking regions in the computational domain (i.e. the grid) that require additional resolution and dynamically overlaying finer grids over these regions. AMR-based techniques start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain requiring additional resolution are tagged and finer grids are overlayed on the tagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring more resolution are similarly tagged and even finer grids are overlayed on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The adaptive grid hierarchy corresponding to the AMR formulation by Berger & Oliger [2] is shown in Figure 1. Operation on the hierarchy defined by this algorithm are outlined below:

Time Integration: Time integration is the update operation performed on each grid at each level of the adaptive grid hierarchy. Integration uses an application specific difference operator.

Inter-Level Operations: Inter-level operations are used to communicate solutions values between grids on different levels of the adaptive grid hierarchy. Two primary inter-grid operations are *Prolongations* operations defined from a coarser grid to a finer grid and *Restriction* operations defined from a finer grid to a coarser grid.

Regriding: The regriding operation consists of three steps: (1) flagging regions needing refinement based on an application specific error criterion, (2) clustering flagged points, and (3) generating the refined grid(s). An application specific error estimator is required to flag regions requiring higher resolution. Regriding can result in the creation of a new level of refinement or additional grids at existing levels, and/or the deletion of existing grids.

2.1 Interactive Experimentation, Computational Steering and AMR

2.1.1 Interaction Specification of Algorithms and Parameters

Success of the AMR technique is determined by its ability to effectively concentrate additional resources, i.e. grid refinements, to only those regions in the domain that need it. Erroneous refinements result in wasted computational cycles and storage and may lead to incorrect results. As the class of applications that use AMR are typically very computation and data intensive, spurious refinements can limit the size of the simulation performed and seriously increase simulation times. In an AMR application code, regriding is determined by a set of parameters such as an error threshold and regrid frequency, and the error estimation algorithm. The appropriate choice of these elements depends on the nature of the solution which is only know at run-time. Traditional implementation of these applications are forced to make static selections of the regriding parameters and error estimation algorithm even though the choices may not be appropriate at all stages of the solution evolution. Any changes are made after an a posteriori evaluation. Clearly, the regriding component of an AMR application can benefit from interactive experimentation and steering capabilities to tune the regriding parameter selection and error estimation algorithm to the current characteristics and requirements of the solution.

Another critical component of the AMR algorithm is the inter-level interpolation operators that couple data at different levels of resolution. It is imperative that these operators have a sufficient degree of accuracy (i.e. interpolation order) so that they do not introduce errors into the solution. Higher order interpolators can be computationally expensive and cause increased communications, while lower order ones may not be accurate enough. The appropriate choice for interpolation order and algorithm is once again dependent on the state of the solution evolution and should ideally be selected at runtime. Interactive experimentation and steering capabilities can enable this.

Finally, the ability for change, modify or tune model specific parameters and algorithms, for e.g. the well placements in an oil reservoir simulation or the apparent horizon finder in a black-hole simulation, using interactive experimentation can be very beneficial specially for very large runs.

2.1.2 Interactive Visualization and Input/Output

In addition to interactively experimenting with parameters and algorithms used by the adaptive application at runtime, the ability to selectively monitor the state of the simulation is critical, specially in simulations that can run for many months. This includes interactive visualization of the applications parameters and fields, on-the-fly input/output control and interactive checkpoint, restart and rollback capabilities.

3 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 3.2. 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 4.

3.1 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.

3.2 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 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.

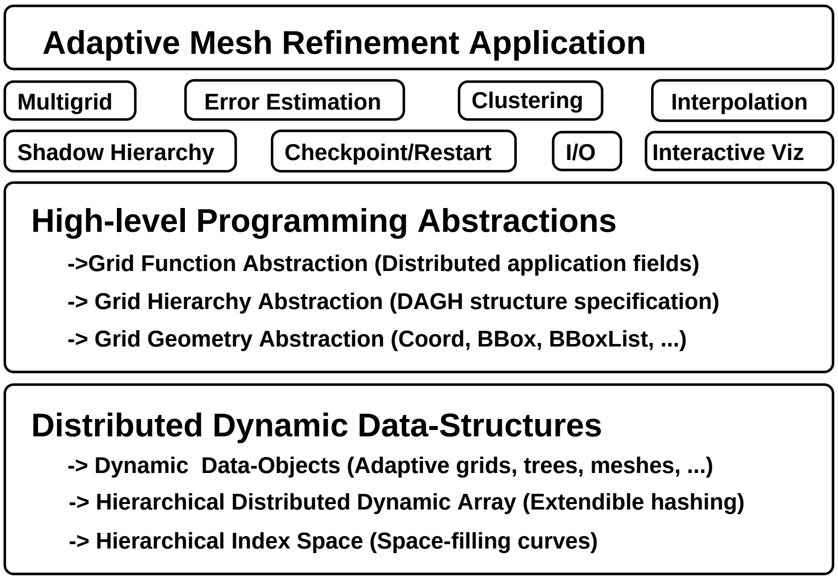

Figure 2: Design model for the HDDA/DAGH Program Development Infrastructure

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 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.

3.3 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.

3.4 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.

4 Design and Implementation of the HDDA/DAGH Infrastructure

4.1 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 [2] 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 5. 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.

4.2 Instantiation of the Design Model for HDDA/DAGH

Figure 3 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) 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

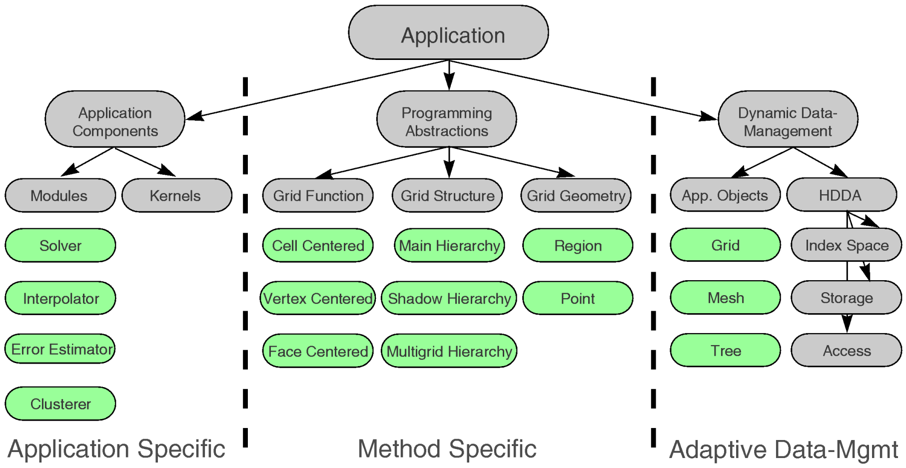

Figure 3: HDDA/DAGH Abstraction Hierarchy

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 AMR 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 4. 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

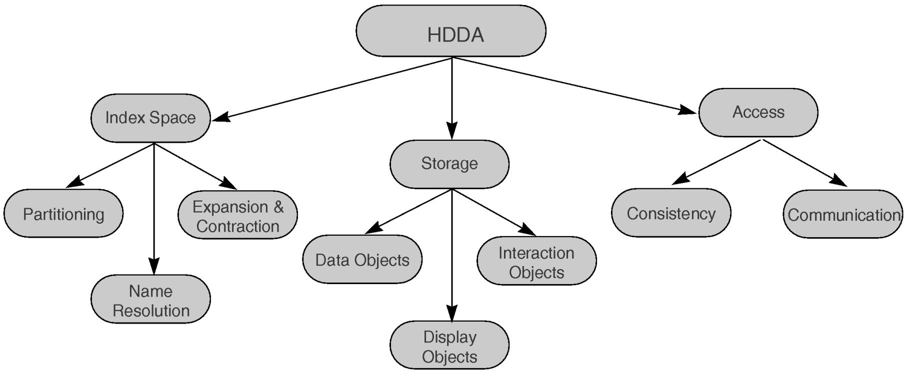

Figure 4: Hierarchical Abstractions of the HDDA

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.

4.3 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.

4.3.1 Hierarchical, Extendible Index Space

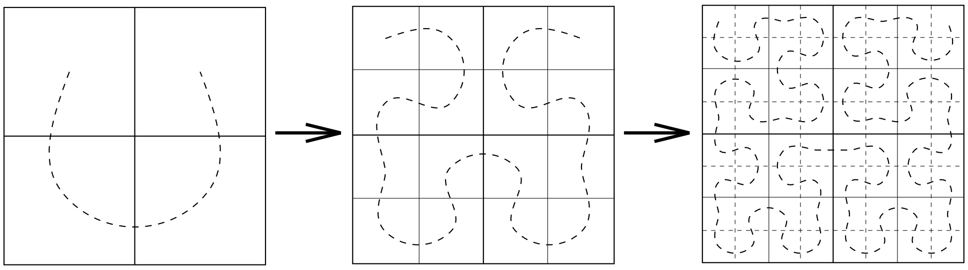

Figure 5: Hierarchical Space-Filling Mappings

The hierarchical, extendible index space component of the HDDA is derived directly from the application domain using space-filling mappings [6] which are computationally efficient, recursive mappings from N-dimensional space to 1-dimensional space. Figure 5 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.

4.3.2 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.

4.3.3 Storage and Access

Data storage is implemented using extendible hashing techniques [7] to provide a dynamically extendible, globally indexed storage (see Figure 6). 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 7.

4.3.4 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.

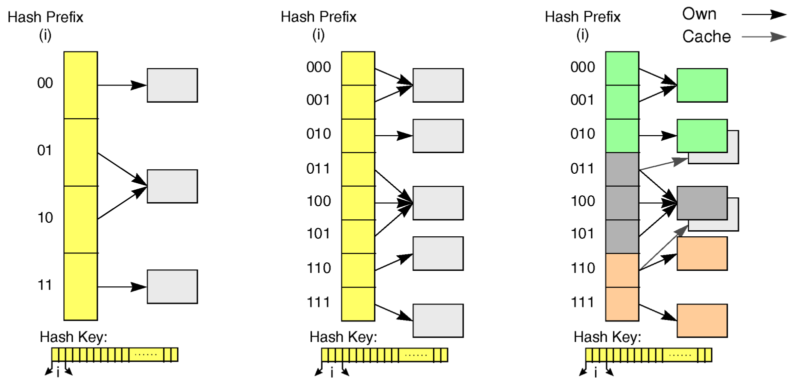

Figure 6: Extendible hashing for distributed dynamic storage and access

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.

4.3.5 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

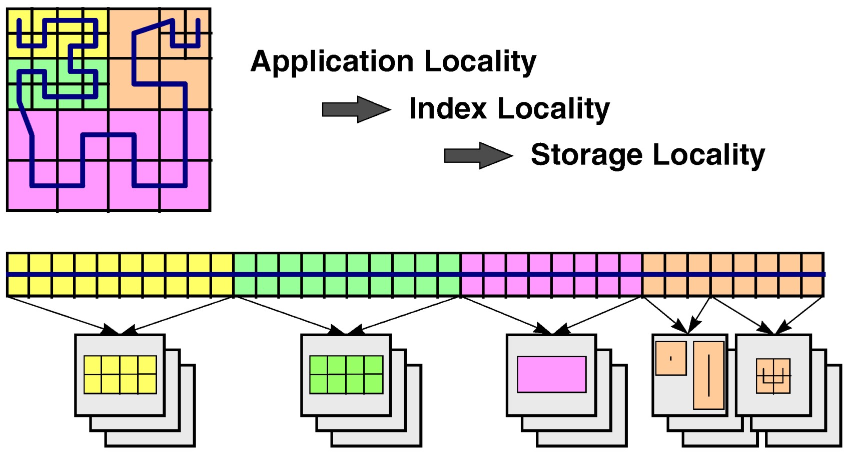

Figure 7: HDDA/DAGH distributed dynamic storage

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

4.4 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 8 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 9. 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 regions 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

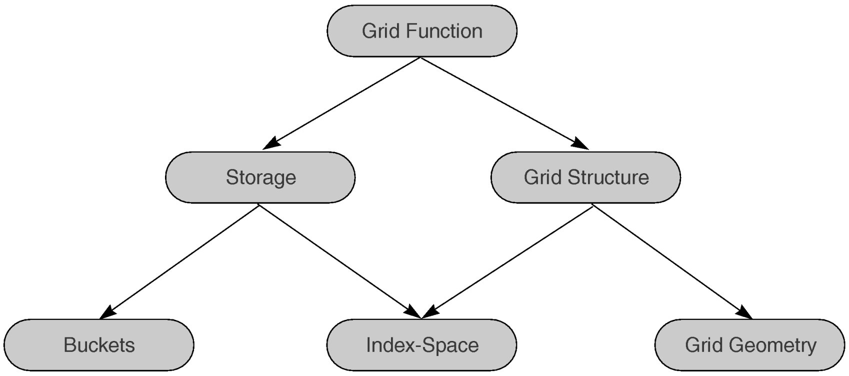

Figure 8: Preserving Design in Implementation: Hierarchical Abstraction

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 AMR 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 8 and 9 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.

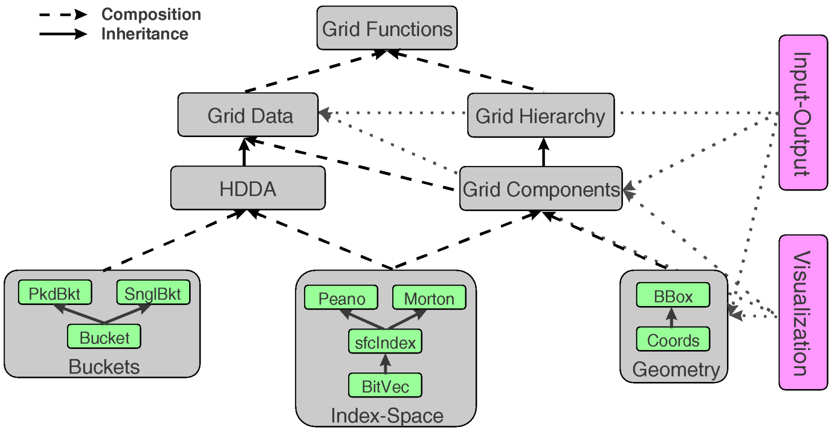

Figure 9: Preserving Design in Implementation: Object Oriented Implementation Hierarchy

4.5 Interaction and Steering in HDDA/DAGH

Application of separation of concerns to the HDDA/DAGH design allows HDDA base objects to be directly extended with visualization, analysis and interaction capabilities. This is achieved by allowing the base objects to inherit from visualization and input/output classes as shown in Figure 9. These objects encapsulate visualization, interaction and input/output operations and define base methods and operators. When application objects are defined by specializing HDDA base objects, these methods and operators are also specialized. For example, grid implemented by specialization of HDDA objects will also specialize

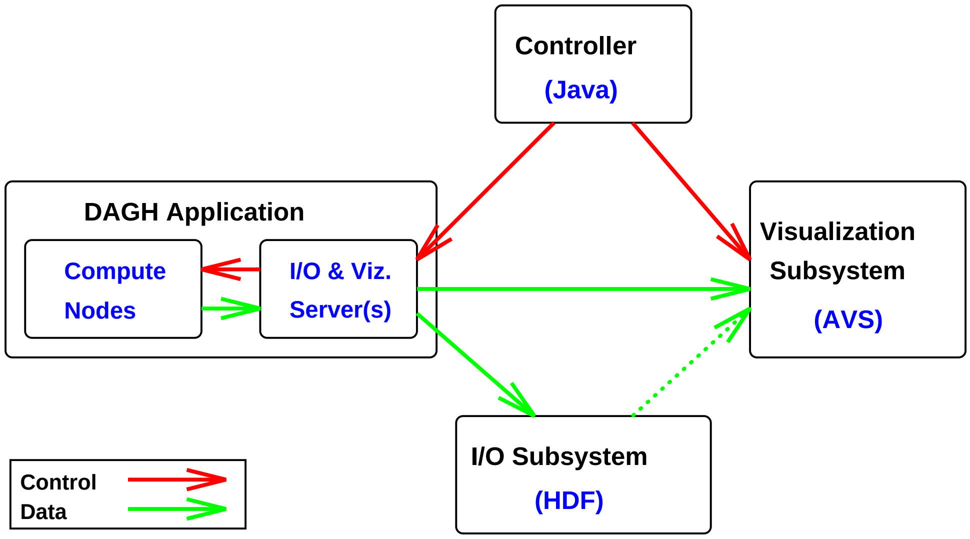

Figure 10: DAGH Interactive Steering Framework

visualization, interaction and input/output operators to grids and grid functions. These objects will then directly generate, for example, connectivity information for displaying the grid structure, contouring and iso-surface extraction operations for visualizing grid function data; grid interaction sensors to define and interpret possible interactions with the grid structure such as refinement and coarsening, and grid functions such as error estimation and interpolation; and input and output methods. An overview current HDDA/DAGH interactive framework is illustrated is Figure 10. Note that in this figure the application and the visualization and interactions subsystems operate off the same HDDA/DAGH data-structures.

5 Applications of HDDA/DAGH

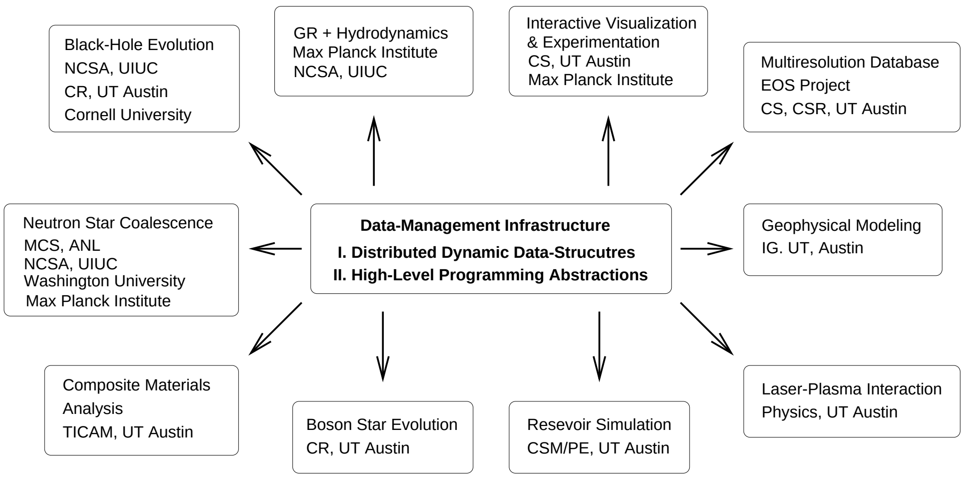

Figure 11: HDDA/DAGH Applications

Figure 11 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.

6 Conclusions & Current Status

Interactive experimentation and computational steering has a natural application to parallel adaptive algorithms that underlie most large-scale scientific and engineering simulations. However, these applications can fully benefit these capabilities only is the they can be driven from the same data-management infrastructure as the computations. In this paper we presented a systems engineering methodology based on the design principles of separation of concerns and hierarchical abstraction that can enable such an integration of computation, interaction, visualization, and steering in a common infrastructure. We then described the application of this methodology to the design of the HDDA/DAGH framework for parallel AMR that incorporates these capabilities. HDDA/DAGH has be successfully used to support a number of large-scale adaptive scientific and engineering simulations. We are currently evaluating the interactive steering capabilities in DAGH for these simulations. The final paper will include a description of this evaluation.

References

[1] M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski, "A Common Computational Infrastructure for Adaptive Algorithms for PDE Solutions", *Submitted for publications*, May 1997.

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

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

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