DAGH: A Data-Management Infrastructure for Parallel Adaptive Mesh-Refinement Techniques

Preliminary Users Guide

Manish Parashar and James C. Browne
Department of Computer Sciences
University of Texas at Austin
{parashar, browne}@cs.utexas.edu

Contents

1 Introduction

The DAGH (Distributed Adaptive Grid Hierarchy) data-management infrastructure for parallel adaptive mesh refinement techniques consists of two components:
(1) A set of programming abstractions in which computations on dynamic hierarchical grid structures are directly implementable. The appropriate programming abstractions are a hierarchy of scalable distributed dynamic grids and a set of operations on this grid hierarchy. The operations include creation, partitioning, computations on the grid such as stencil operations, communication among grid partitions at a single level and communication among grids at different levels of the hierarchy.
(2) A set of distributed dynamic data-structures that support the implementation of the abstractions in parallel execution environments and preserve efficient execution while providing transparent distribution of the grid hierarchy across processing element execution environment.
The fundamental programming abstractions provided by the infrastructure are:

Geometry Abstractions: Geometry abstractions enable the user to identify and operate on regions of the computational domain. These abstractions include points (class *Coords*), bounding boxes (class *BBox*), lists of bounding boxes (class *BBoxList*). The abstractions can be used to define regions that have to be operated on, refined, masked out in an operation, input or output, etc.

Grid Hierarchy Abstraction: The grid hierarchy abstraction (class *GridHierarchy*) defines the template for the entire AMR grid hierarchy. The structure of the hierarchy is a DAG consisting of a dynamic set of levels and a dynamic number of component grids at each level. The grid hierarchy is automatically partitioned across the specified number of processing elements so as to balance load and to optimize inter-grid and intra-grid communications requirements.

Grid Function Abstraction: Grid function abstraction (class *GridFunction*) represents an application variable defined on the computational domain. A grid function is defined on and associated with grid hierarchy. Distributed storage for the grid function is automatically allocated and maintained as the grid hierarchy evolves. Operations on the grid function are performed in a data-parallel fashion at a specified level(s) of the hierarchy.

The DAGH data-management infrastructure is implemented as a `C++` class library on top of the MPI communication system. Currently supported platforms include the IBM SP-2 and a network of RS6000 and Sun workstations.

1.1 Programming Model & Interface

DAGH supports a coarse grained data-parallel programming model where parallelism is achieved by operating on regions of the grid hierarchy in parallel.

The DAGH programming interface is developed so as to enable computations to performed by traditional Fortran 77 & Fortran 90 kernels. Consequently, all storage is maintained in Fortran compatible fashion.

1.2 Parallel/Sequential Operation

DAGH automatically switches between parallel and sequential execution depending on whether the MPI communication system is initialized or not. If MPI has been initialized, DAGH distributed the computation across the number of processors specified at run-time. In the current implementations of MPI, the MPI communications system is initialized by adding the following to the top of main application routine:

```
MPI_Init( &argc, &argv );
```

2 The Grid Geometry Abstractions

Grid geometry abstractions are used to address and operate on regions in the computational domain and consist of three abstract data types:

1. *Coords*
2. *BBox*
3. *BBoxList*

The grid geometry abstractions provided by the DAGH data-management infrastructure borrow from the LPARX[1] system developed by S. Kohn and S. Baden

2.1 Coords

The *Coords* class is to used to specify a point in the computational domain and is declared as follows:

```
Coords c(1, int const i) /* 1-D */
Coords c(2, int const i, int const j) /* 2-D */
Coords c(3, int const i, int const j, int const k) /* 3-D */
```

The components of a Coords object c in the i^{th} dimension can be indexed as follows: `c(i)`

Operators: The following operators are defined on a Coords object:

- Arithmetic operators *+, -, *, /, +=, -=, *=, & /=* between 2 Coords objects and between a Coords object and a constant.

- Unary negation operator.
- Logical operators ==, <, <=, >, >=, and != between 2 Coords objects.

2.2 BBox

The *BBox* class abstracts a bounding box consisting of a lower bound, an upper bound, and a step size. It is declared as follows:

```
BBox b(1, int const imin, int const imax, int const step)  /* 1-D */
BBox b(2, int const imin, int const jmin,
          int const imax, int const jmax,
          int const step) /* 2-D */
BBox b(3, int const imin, int const jmin, int const kmin,
          int const imax, int const jmax, int const kmax,
          int const step) /* 3-D */
```

Query Operators: Query operators defined on a BBox object *bb* are:

Coords upper(bb): Returns the upper bound of the bounding box

Coords upper(bb, i): Returns the upper bound of the bounding box in the i^{th} dimension.

Coords lower(bb): Returns the lower bound of the bounding box

Coords lower(bb, i): Returns the lower bound of the bounding box in the i^{th} dimension.

Coords extents(bb): Returns the extents of the bounding box

Coords extents(bb, i): Returns the extent bound of the bounding box in the i^{th} dimension.

int size(bb): Returns the number of point in the bounding box

int stepsize(bb): Returns the step size of the bounding box

int inside(bb, c): Returns true if Coords *c* contained by the bounding box

Unary Operators: Unary operators defined between on BBox object *bb* are:

```
grow(bb, int const amount)
/* Grow upper and lower bounds by (amount*step) */
growupper(bb, int const amount)
/* Grow upper bound by (amount*step) */
growlower(bb, int const amount)
/* Grow lower bound by (amount*step) */
shift(bb, int const amount)
/* Shift upper and lower bound by (amount*step) */
```

Binary Operators: Binary operator defined between BBox objects bb1 and bb2 are:

- Assignment: `bb1 = bb2`
- Union: `bb1 + bb2` & `bb1 += bb2`
- Intersection: `bb1 * bb2` & `bb1 *= bb2`

2.3 BBoxList

The *BBoxList* class implements a list of BBox objects as is declared as follows:

```
BBoxList bblist
```

Operators Operations on a BBoxList object *bblist* are:

BBox first(bblist): Get the first BBox in the list

BBox next(bblist): Get the next BBox in the list

BBox prev(bblist): Get the prev BBox in the list

BBox current(bblist): Get the current BBox in the list

BBox last(bblist): Get the last BBox in the list

add(bblist, bb): Add BBox *bb* to the end of the list

insert(bblist, bb): Insert BBox *bb* before the current BBox

remove(bblist): Remove the current BBox

int number(bblist): Returns the number of BBox objects in the list

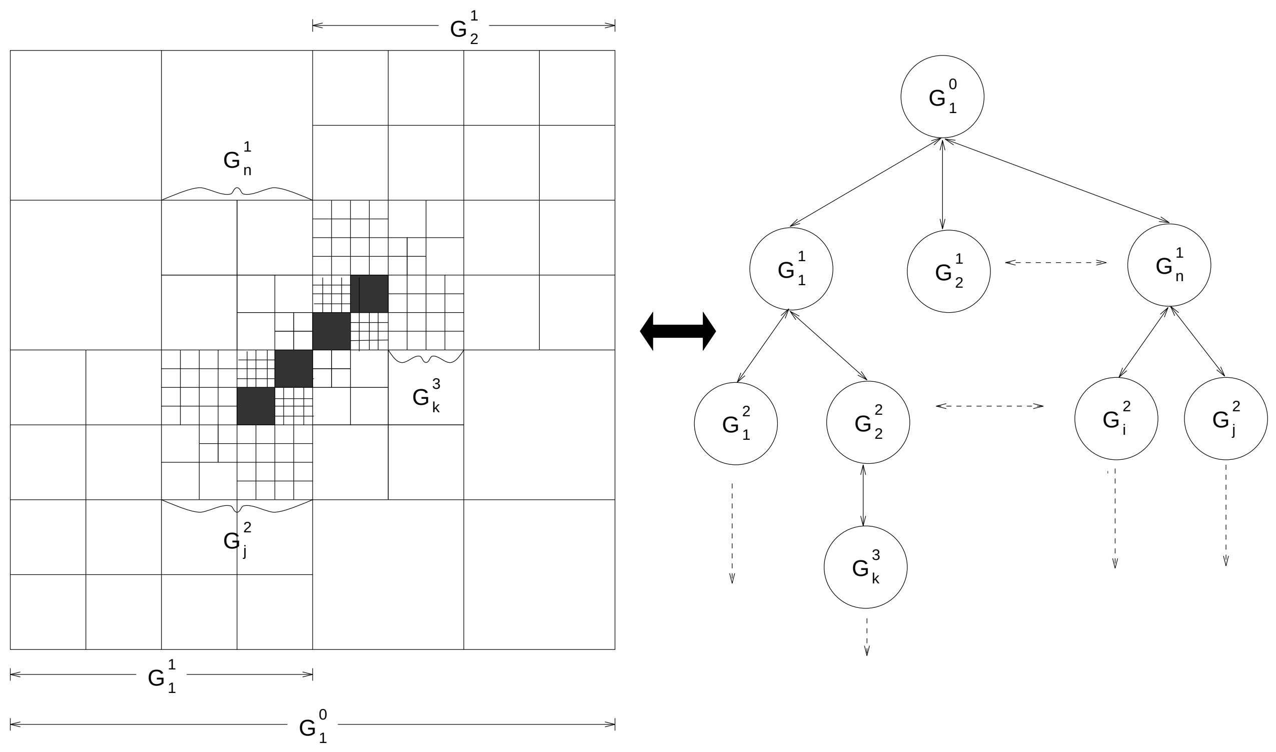

Figure 1: Adaptive Grid Hierarchy - (2D)

3 The Grid Hierarchy Abstraction

The DAGH infrastructure abstracts a distributed adaptive grid hierarchy as an abstract data type *Grid-Hierarchy*. GridHierarchy represents a family of distributed, dynamically overlayed grids, $\{G_n^l\}$, where the superscript l; $0 \leq l \leq L$ represents the level of refinement (0 being the coarsest (base) grid and L the finest grid) and the subscript n indexes *component grids* at the same level. A complementary view of the grid hierarchy is a directed acyclic graph (DAG) where each node of the graph represents a component grid. Levels of this DAG correspond to the levels of refinement in the adaptive grid hierarchy, and nodes at the same level of the DAG correspond to component grids at a particular level of refinement. The adaptive grid hierarchy is a dynamic structure, both in the number of levels of refinement and in the number of component grids at any particular level of refinement. The two views of the DAGH grid hierarchy are shown in Figure 1.

3.1 Defining a GridHierarchy

Definition of a GridHierarchy consists of 3 steps: (1) declaration, (2) parameter specification, (3) composition.

GridHierarchy Declaration: The syntax for declaring a GridHierarchy is as follows:

```
GridHierarchy GH(int const Dimension,int const Type,int const MaxLevels);
```

where:

- *Dimension* is the rank of the each component grid in the hierarchy. Supported dimensions are 1, 2, & 3.
- *Type* specifies the type of each component grid in the hierarchy. Currently supported values include **NON_CELL_CENTERED** and **CELL_CENTERED**.
- MaxLevels declares the maximum number of levels that the hierarchy can contain.

GridHierarchy Specification: Specification of the GridHierarchy consists of defining the structure of the base grid. The syntax is:

```
SetBaseGrid (GridHierarchy const &GH, double const *bb, int const *shape);
```

where:

- *GH* is the GridHierarchy whose base grid is being set.
- *bb* is an array of type *double*, dimension 2**Dimension*, and contains the bounding box of the base grid. The bounding box coordinates are stored in the *bb* array as lower bound/upper bound pairs. In 3-d this format is: $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$.
- *shape* is an array of type *integer*, dimension *Dimension* and contains the number of grid elements in each dimension.

Optional parameters that can be specified include a refinement factor and external boundary specifications. If these parameters are not specified a preset default value is used.

Refinement Factor: Currently, only refinement factors that are a power of 2 can be used. The syntax for specifying the refinement factor is:

```
SetRefineFactor(GridHierarchy const &GH, int const refine_factor);
```

where:

- *GH* is the GridHierarchy whose refinement factor is being set.
- *refine_factor* is the refinement factor.

The default value of the refinement factor is 2.

External Boundaries: External boundaries can be specified as regions (bounding boxes) on the grid hierarchy and can be operated on in a special manner. Currently, the system can automatically handle a regular external boundary. Such a boundary is specified as follows:

```
SetRegExtBndry(GridHierarchy const &GH, int const ext_bndry_width);
```

where:

- *GH* is the GridHierarchy whose refinement factor is being set.
- *ext_bndry_width* is the width of the external boundary.

The default value of the external boundary is 0.

GridHierarchy Composition: Composing the the GridHierarchy sets up the distributed data-structures and initializes information required to maintain this structure. The syntax is as follows:

```
ComposeHierarchy(GridHierarchy const &GH);
```

where:

- *GH* is the GridHierarchy to be composed.

3.2 Regriding the GridHierarchy

The regriding process consists of two step: (1) Re-constructing the grid hierarchy by refining/unrefining at different levels; and (2) Recomposing the grid hierarchy.

Refinement: The GridHierarchy is refined at a particular level by overlaying a refined grid over specified regions of the existing component grids at that level. The syntax for refinement is as follows:

```
Refine(GridHierarchy const &GH, BBoxList *bblist, int const Level);
```

where

- *GH* is the GridHierarchy to be refined.
- *bblist* is a list of bounding boxes defining the regions of the component grids at level *Level* that have to be refined. The BBoxList class is described in Section 2.
- *Level* is the level over which a new refined component grid has to be constructed.

Unrefinement: The GridHierarchy is unrefined at a particular level by destroying all component grids at all levels finer than the specified level. The syntax for unrefining is as follows:

```
Unrefine(GridHierarchy const &GH, int const Level);
```

where

- *GH* is the GridHierarchy to be refined.
- *Level* is the level at which unrefinement is to be performed.

Recomposition: GridHierarchy recomposition efficiently performs the following tasks:

1. Reconstruct the new grid hierarchy.
2. Re-partition and re-distribute the new hierarchy to balance load, and optimize inter-grid and intra-grid communications (maintain locality).
3. Perform all data transfers to re-construct grid functions associated with the grid hierarchy.
4. Prolongations to initialize grid functions at the new component grids using the prolongation function associated with the particular grid functions (see Section 4 for grid functions specification details).

The syntax for recomposition is:

```
RecomposeHierarchy(GridHierarchy const &GH);
```

where

- *GH* is the GridHierarchy to be recompose.

4 The Grid Function Abstraction

The abstract data type *GridFunction* associates application data elements with the GridHierarchy and defines storage with the different components grid of the hierarchy.

4.1 Defining a GridFunction

The syntax for defining a grid functions is as follows:

```
GridFunction(Dimensions)<Type> GF(char const *print_name,
  int const time_stencil, int const space_stencil,
  GridHierarchy const &GH);
```

where:

- *Dimension* is the dimension of the GridFunction. Currently supported dimension include 1, 2, and 3.
- *Type* is the type of the GridFunction. Currently supported types include all standard `C` types as well as static user defined types.
- *print_name* defined the symbolic name associated with the GridFunction and used to identify it during I/O.
- *time_stencil* specifies the temporal stencil to used for the GridFunction. Storage is maintained for $(2 * time\_stencil + 1)$ time levels numbered from $-time\_stencil$ to $time\_stencil$.
- *space_stencil* specifies the space stencil to used for the GridFunction. The space stencil defined the ghost regions regions maintained around distributed grid blocks and the inter-grid communications required.
- *GH* specifies the GridHierarchy that will serve as the template for the GridFunction.

A 3-dimensional grid function u of type *double* with a time stencil of 1 (3 timesteps) and a space stencil of 1 is defined on GridHierarchy gh as follows:

```
GridFunction(2)<double> u("U", 1, 1, gh);
```

4.2 Operating on a GridFunction

Operations on a grid function are performed in a data-parallel fashion and can be defined at two granularities: (1) components grids at a particular time and level; and (2) entire level of the GridHierarchy.

Operating on Component Grids: The c^{th} component grid at time t and level l of GridFunction u can be indexed as follow:

```
u(t,l,c) /* int const t, l, c */
```

Querying component grid geometry: Query operators are on component grids are:

BBox bbox(u(t,l,c)): returns the bounding box of the component grid.

Coords upper(u(t,l,c)): returns the upper bound of the component grid.

Coords upper(u(t,l,c), i): returns the upper bound of the component grid in the i^{th} dimension.

Coords lower(u(t,l,c)): returns the lower bound of the component grid.

Coords lower(u(t,l,c), i): returns the lower bound of the component grid in the i^{th} dimension.

Coords extents(u(t,l,c)): returns the extents of the component grid.

Coords extents(u(t,l,c), i): returns the extent bound of the component grid in the i^{th} dimension.

int size(u(t,l,c)): returns the number of grid elements in the component grid.

Indexing elements of a Component Grids: Individual elements of the components grid can be indexed as follows:

```
u(t,l,c)(i)  /* 1-D */
u(t,l,c)(i,j)  /* 2-D */
u(t,l,c)(i,j,k)  /* 3-D */
```

Arithmetic operations: The following arithmetic operations are defined on a component grid:

```
u(t,l,c) op val; /* Type const val */
u(t,l,c) op u2(t2,l2,c2);
```

where *op* is one of =, +=, -=, *=, and /=; *u* and *u2* are GridFunctions of the same type and dimension; and *val* is a constant. The latter operation is performed in the region where component grids *u* and *u2* intersect.

Bounding-box based arithmetic operations: Arithmetic operation can be defined on regions of the component grid using bounding boxes. These operations have the following format:

```
Operation(lhs,rhs,to_bbox,from_bbox)
Operation(lhs,val,to_bbox)
```

where:

- *Operation* is one of *equals, plus, minus, multiply,* and *divide.*
- *lhs* is the component grid that is operated on.
- *rhs* is the component grid to be used in the right hand side of the expression.
- *val* is the constant value to be used in the right hand side of the expression.
- *to_bbox* is of type *BBox* and defines the region of *lhs* that is to be operated on.
- *from_bbox* is of type *BBox* and defines the region of *rhs* that that is to be used in the expression.

Note that the above operation, are performed only in the region where *lhs*, *rhs*, *to_bbox*, and *from_bbox* intersect. Either or both of *to_bbox*, and *from_bbox* can be omitted.

Logical Operations: Logical operations defined on component grid return a list of bounding boxes (class BBoxList) representing clustered regions where the relation is true. The following logical operations are defined on a component grid:

```
u(t,l,c) op val; /* Type const val */
u(t,l,c) op u2(t2,l2,c2);
```

where *op* is one of ==, >=, >, <=, <, and ! =; u and $u2$ are GridFunctions of the same type and dimension; and *val* is a constant. In the latter operation is performed in the region where component grids u and $u2$ intersect. Note

Bounding-box based logical operations: Arithmetic operation can be defined on regions of the component grid using bounding boxes. These operations have the following format:

```
Operation(lhs,rhs,to_bbox,from_bbox)
Operation(lhs,val,to_bbox)
```

where:

- *Operation* is one of *is_eq, is_gt, is_ge, is_lt, is_le,* and *is_ne.*
- *lhs* is the component grid to be used in the left hand side of the expression.
- *rhs* is the component grid to be used in the right hand side of the expression.
- *val* is the constant value to be used in the right hand side of the expression.
- *to_bbox* is of type *BBox* and defines the region of *lhs* that that is to be used in the expression.
- *from_bbox* is of type *BBox* and defines the region of *rhs* that that is to be used in the expression.

Note that the above operation, are performed only in the region where *lhs*, *rhs*, *to_bbox*, and *from_bbox* intersect. Either or both of *to_bbox*, and *from_bbox* can be omitted.

Data-Parallel Operator: Data parallel operations on all component grids (c) at a particular time (t) and level (l) of a GridFunction u can be defined using the provided *forall* operator as follows:

```
forall(u,t,l,c)
  ......
  ......
end_forall
```

Operating on a level of the GridHierarchy: GridFunction operations defined an entire level of the GridHierarchy have the following format:

```
Operation(lhs,tlhs,llhs,val,to_bbox)
Operation(lhs,tlhs,llhs,rhs,trhs,lrhs,to_bbox,from_bbox)
```

where:

- *Operation* is one of *equals, plus, minus, multiply,* and *divide.*
- *lhs* is the GridFunction that is operated on.
- *tlhs* specifies the time of the *lhs* to be operated on.
- *llhs* specifies the level of the *lhs* to be operated on.
- *rhs* is the GridFunction to be used in the right hand side of the expression.
- *trhs* specifies the time of the *rhs* to be used on the right hand side of the expression.
- *lrhs* specifies the level of the *rhs* to be used on the right hand side of the expression.
- *val* is the constant value to be used in the right hand side of the expression.
- *to_bbox* is of type *BBox* and defines the region of *lhs* that is to be operated on.
- *from_bbox* is of type *BBox* and defines the region of *rhs* that that is to be used in the expression.

Note that the above operation, are performed only in the region where *lhs*, *rhs*, *to_bbox*, and *from_bbox* intersect. Either or both of *to_bbox*, and *from_bbox* can be omitted.

4.3 Intra-Grid Communications

Intra-Grid communications consist of reading in ghost regions from neighboring processing elements and writing out to the ghost regions of neighboring processing elements. The following operators are defined to perform intra-grid communications:

Ghost Region Update: The *UpdateGhostRegions* operator first writes out ghost regions of its neighboring processing elements and reads in its ghost regions. Its syntax is:

```
UpdateGhostRegions(GF, t, l, c)
UpdateGhostRegions(GF, t, l)
```

The former operator updates the ghost regions of the c^{th} component grid at level l and time t of GridFunction GF; while the latter updates the ghost regions of all the components grids at level l and time t of GridFunction *GF*.

Ghost Region Read: The *ReadGhostRegions* operator reads in ghost regions from neighboring processors. Its syntax is:

```
ReadGhostRegions(GF, t, l, c)
ReadGhostRegions(GF, t, l)
```

The former operator reads the ghost regions for the c^{th} component grid at level l and time t of GridFunction GF; while the latter reads the ghost regions for all the components grids at level l and time t of GridFunction *GF*.

Ghost Region Write: The *WriteGhostRegions* operator writes out to ghost regions to neighboring processing elements. Its syntax is:

```
WriteGhostRegions(GF, t, l, c)
WriteGhostRegions(GF, t, l)
```

The former operator writes out the ghost regions for the c^{th} component grid at level l and time t of GridFunction GF; while the latter write out the ghost regions for all the components grids at level l and time t of GridFunction *GF*.

4.4 Inter-Grid Communications

Inter-grid operations can be defined directly using the *forall_levels* data-parallel operator. If the default composite partitioning scheme is used, this operations is local to each processor. For example restricting from level l to level *l-1* for GridFunction u at time t can be specified as follows:

```
forall_levels(u,l-1,l,c)
  u(t,l-1,c) = u(t,l,c);
end_forall_levels
```

4.5 GridFunction Parameters

In addition to the space/time stencil that is specified during GridFunction definition, the following optional parameters can also be specified for each GridFunction.

InitFunction The InitFunction function pointer is used to specify the function to be used to initialize the GridFunction. Its default is NULL.

UpdateFunc The UpdateFunc function pointer is used to specify the function to be used to update the GridFunction. Its default is NULL.

ProlongFunc The ProlongFunc function pointer is used to specify the function to be used to perform prolongations. By default, it points to a linear interpolator.

RestrictFunc The RestrictFunc function pointer is used to specify the function to be used to perform restrictions. Its default is NULL and injection is used to perform restrictions.

IOFunc The IOFunc function pointer is used to specify the function used to handle I/O for the GridFunction. Its default is NULL.

The interface for each of the above functions is defined in Section 5. The following operators can by used to set the above parameters:

```
SetInitFunction(InitFunc *initfunc)
SetUpdateFunction(UpdateFunc *updatefunc)
SetProlongFunction(ProlongFunc *prolongfunc)
SetRestrictFunction(RestrictFunc *restrictfunc)
SetIOFunction(IOFunc *iofunc)
```

If the respective parameters have been defined, the following operators to be used to apply the specified function at a particular time and level:

```
Initialize(GridFunction(d)<Type> GF, int const t, int const l,
   float *args, int const cnt)
Update(GridFunction(d)<Type> GF, int const *t, int const tcnt, int const l,
       float *args, int const cnt)
Prolong(GridFunction(d)<Type> GF, int const t,  int const from_l, int const to_l,
float *args, int const cnt)
Restrict(GridFunction(d)<Type> GF, int const t, int const from_l, int const to_l,
 float *args, int const cnt)
IO(GridFunction(d)<Type> GF, int const t, int const l,
   float *args, int const cnt)
```

The parameter *args* is array of dimension *cnt* that contains arguments to be passed to the respective function. In case of the *Update* operator, an arrays t of dimension *tcnt* is used to specify the number of timesteps of the GridFunction that are to be used in the update.

5 Fortran Interfaces

5.1 Passing Arguments to Fortran Kernels

Data storage associated with each component grid of the GridHierarchy is maintained so that it can be operated on by Fortran 77/90 kernels. The data associated with a component grid is passed to a Fortran subroutine as a combination of the data pointer and bounding box information. In case of a 3-D GridFunction of type “float”, this interface would be:

`real data, integer lowerbound(3), integer upperbound(3), integer shape(3),`

Similarly the interface for a bounding box is as follows:

`integer lowerbound(3), integer upperbound(3)`

The macro *GC_FORTRAN_ARGS* and *BB_FORTRAN_ARGS* have been defined to automatically generate these interfaces. For example for the c^{th} component grid at time t and level l of GridFunction u the following statement will generate the required interface:

`GC_FORTRAN_ARGS(u(t,l,c))`

while that for a BBox bb can be generated as follows:

`BB_FORTRAN_ARGS(bb)`

5.2 Predefined Fortran Interfaces

Interfaces for the initialization, interior update, prolongation, restriction, and i/o have been predefined. Function having these interfaces can be associated with the a GridFunction as described in Section4 and be used to specify operations on it. The currently predefined interfaces for a 3 dimensional GridFunction of type "float" (correspondingly "real" in Fortran) are as follows:

Initialization Function:

```
subroutine initialize(data, lb, ub, shape, args, cnt)
integer lb(3), ub(3), shape(3)
real data(shape(1),shape(2),shape(3))
integer cnt
real args(cnt)
```

Update Function:

```
        subroutine update(data1, lb1, ub1, shape1,
                          data2, lb2, ub2, shape2,
                          args, cnt)
        integer lb1(3), ub1(3), shape1(3)
        real data1(shape1(1),shape1(2),shape1(3))
        integer lb2(3), ub2(3), shape2(3)
        real data2(shape2(1),shape2(2),shape2(3))
        integer cnt
        real args(cnt)
```

Prolongation Function:

```
subroutine prolong(data_from, lb_from, ub_from, shape_from,
                          data_to, lb_to, ub_to, shape_to,
                          args, cnt)
        integer lb_from(3), ub_from(3), shape_from(3)
        real data_from(shape_from(1),shape_from(2),shape_from(3))
        integer lb_to(3), ub_to(3), shape_to(3)
        real data_to(shape_to(1),shape_to(2),shape_to(3))
        integer cnt
      real args(cnt)
```

Restriction Function:

```
subroutine restrict(data_from, lb_from, ub_from, shape_from,
                  data_to, lb_to, ub_to, shape_to,
                  args, cnt)
        integer lb_from(3), ub_from(3), shape_from(3)
        real data_from(shape_from(1),shape_from(2),shape_from(3))
        integer lb_to(3), ub_to(3), shape_to(3)
        real data_to(shape_to(1),shape_to(2),shape_to(3))
        integer cnt
      real args(cnt)
```

I/O Function:

```
subroutine io(data, lb, ub, shape, args, cnt)
integer lb(3), ub(3), shape(3)
real data(shape(1),shape(2),shape(3))
integer cnt
real args(cnt)
```

In the interfaces above the real array *args* of dimension *cnt* is used to pass miscellaneous arguments to the function.

References

[1] 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.