DAGH User's Manual

Shyamal Mitra

29 July 1999

Contents

1 Program Structure

DAGH provides the class libraries needed to implement adaptive mesh refinement methods. It also provides the framework to run an application on parallel processors. An application built using DAGH as the framework consists of a C++ main driver which invokes computational routines that maybe written in C, C++, or Fortran. A parallel adaptive application written using DAGH has the following structure:

1.1 Driver

The driver (written in C++) creates and initializes the DAGH hierarchy. It invokes the solution routines and provides the interface to visualization routines. The structure of the driver is discussed in detail in the next section.

1.2 Routines to perform grid level operations

Update functions for

- Interior points
- Boundary points

The user supplies the kernel to update the values of the function at the grid points - both inside the region under consideration as well as on the boundary. The kernels can be written in Fortran 77, Fortran 90, C, or C++. Note that DAGH uses storage maintained in Fortran compatible fashion.

1.3 Routines to perform inter-grid level operations

- Prolongation
- Restriction

For non-adaptive applications these routines are not needed. But when mesh refinement is involved, communication between grids at different levels of refinement has to be maintained. When a refined grid is created, its boundaries are usually interior to some coarser grid. The boundary values of the refined grid are initialized through prolongation from the coarser grid values. After updating the function values on a fine grid, the underlying coarse grid values are updated through restriction.

1.4 Several utility operations

- I/O routines
- Visualization interface
- Clustering routines

DAGH provides the I/O framework. The input parameters can be hard coded in the driver or read from an input file. The output is sent to an I/O server. The output can be written to a file or sent to a visualization package through an interface defined by the user. In adaptive methods, regions that need refinement are identified. The clustering routines (supplied by the user) determine the optimal sub grids to cover those regions.

1.5 Routines to perform error estimates

In the adaptive mesh refinement method, as the solution proceeds the local error due to the updating is estimated at each of the grid points. The error estimation can be obtained through Richardson's expansion and a shadow hierarchy (a strict 2:1 coarsened grid) or by some other formulation supplied by the user.

1.6 Input parameter file (optional)

If the DAGH I/O routines are involved, then the input parameters can be read from a file.

1.7 Makefile

A makefile is necessary to compile the various routines and link the object files and the libraries to produce the executable code.

2 Driver

In a DAGH application the foremost routine is the driver. It is usually written in C++. Thus all the data types, control flow statements, and operators of C++ can be used. In addition there are DAGH data types, control statements, and operators that can be incorporated in the driver. A template for the driver follows:

- Initialize the environment
  - MPI (Message Passing Interface)
  - I/O
  - Graphical Interface
  - Open files
  - Steering (change values of variables during the computational process)
  - Interaction (check point, roll back, restart)
- Read the input parameters for the problem
  - dimension of the grid
  - size of the grid
  - grid step
  - lower and upper bounds for each of the coordinate directions
  - time step
- Setup Grid Structure
- Setup Grid functions
- Setup initial conditions
- Start the Main Loop
  - User interaction or control
  - Update grid functions at

    * Interior
    * Boundary
  - Communicate result of updating with other grids
  - Output result to a file and/or output to a visualization interface

- End Main Loop

- Shutdown environment

  - MPI
  - I/O
  - Graphical Interface
  - Close files

3 Data Types and Data Declarations

3.1 C++ Data Declarations

Since the driver is written in C++ any legal C++ data declarations can be included in the driver. The fundamental data types being

- char

- int

- float

- double

The fundamental data types can be extended by applying qualifiers. The qualifiers *long* and *short* can be applied to the type *int*. The types *char* and *int* can be qualified by *signed* and *unsigned*.

3.2 DAGH Data Declarations

DAGH provides a set of programming abstractions - a hierarchy of scalable distributed dynamic grids and a set of operations on this hierarchy. The operations include creation, partitioning, computations on the grid, communication among grid partitions at a single level and communication among grids at different levels of the hierarchy. The programming abstractions and the corresponding data types are:

- Grid Geometry
- Grid Hierarchy
- Grid Function

DAGH data declarations are actually invocations of the constructors for the C++ classes which define and implement the DAGH data types.

3.2.1 Grid Geometry

The grid geometry abstractions provided by DAGH borrow from the LPARX[1] system developed by S. Kohn and S. Baden. These abstractions are used to address and operate on regions of the computational domain and consist of three abstract data types.

1. Coords
2. BBox
3. BBoxList

Coords and DCoords The Coords and DCoords classes are used to specify a point in the computational domain. The coordinates of a point can be either of type *int* for class Coords or type *double* for class DCoords and are declared as follows for dimensions of 1, 2, and 3:

Coords c(1, int const i);

DCoords c(1, double const i);

Coords c(2, int const i, int const j);

DCoords c(2, double const i, double const j);

Coords c(3, int const i, int const j, int const k);

DCoords c(3, double const i, double const j, double const k);

The grid has a finite number of discrete points. Coords is a class used to specify a point in the grid and it has integer values. If you have a 2-dimensional grid having 11 points (or 10 cells) along each axis, then the valid values for Coords can go from (0,0) to (10,10).

If you now associate a function with a lower bound of (0.0, 0.0) and a upper bound of (1.0, 1.0) with the above grid then the grid point having Coords (0,0) will have DCoords (0.0, 0.0). The grid point having Coords (1,1) will have DCoords (0.1, 0.1) and so on.

BBox The BBox class provides the abstraction for a bounding box that consists of a lower bound, an upper bound, and a step size. It is declared as follows for dimensions 1, 2, and 3:

BBox b(1, int const imin, int const imax, int const stepi);

BBox b(2, int const imin, int const jmin,
int const imax, int const jmax,
int const stepi, int const stepj);

BBox b(3, int const imin, int const jmin, int const kmin,
int const imax, int const jmax, int const kmax,
int const stepi, int const stepj, int const stepk);

Once you specify the maximum number of levels of refinement and the refinement factor, DAGH creates a grid having the finest mesh size. Since this grid consists of discrete points, the coordinates of the grid points are integral values, with the lower left corner of the grid taken as origin. Bounding boxes are defined using the coordinates of this finest grid as reference.

For example, consider a 1-dimensional base grid having 4 cells. If the maximum number of levels of refinement is 2 and the refinement factor is also 2, the finest grid would contain 8 cells. To specify a bounding box covering the whole base grid we would make the following declaration:

BBox bg (1, 0, 8, 2);

To specify a bounding box covering the whole fine grid the declaration would be:

BBox bf (1, 0, 8, 1);

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

BBoxList bblist;

3.2.2 Grid Hierarchy

Grid hierarchy, an abstract data type of DAGH, represents a family of distributed, dynamically overlayed grids $\{G_n^l\}$. The superscript l represents the level of refinement. l ranges from 0 the coarsest (base) grid to L, the finest grid. The subscript n indexes component grids at the same level of refinement.

Declaring a Grid Hierarchy A grid hierarchy GH is declared as follows:

GridHierarchy GH(const int dim, const int type, const int MaxLevels);

dim dimension or rank of each component grid in the hierarchy. Currently supported dimensions are 1, 2, and 3.

type type of each component grid in the hierarchy. The types can be DAGHCellCentered, DAGHVertexCentered, DAGHFaceCentered_X, DAGHFaceCentered_Y, and DAGHFaceCentered_Z.

MaxLevels maximum number of refinement levels that the hierarchy can attain.

Specifying a Grid Hierarchy The coarsest or base grid has to be defined. There are several functions that define the base grid. [In all of these functions that follow GH is the Grid Hierarchy that we have declared previously.]

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

bb a 2-dimensional array that contains the coordinates of the lower bound and upper bound pairs of the bounding box that define the base grid. For a 3-dimensional bounding box the array would contain - (xmin, xmax, ymin, ymax, zmin, zmax)

shape a 1-dimensional array that contains the number of grid cells in each coordinate direction

The time specifications are set in some user defined units (say seconds):

SetTimeSpecs (GridHierarchy const&GH, double const starttime, double const stoptime, const ntsteps);

starttime starting time for integration

stoptime ending time for integration

ntsteps number of time steps involved

The boundary of the grid can be specified through the following functions:

SetBoundaryType (GridHierarchy const&GH, const int type)

type specifies the nature of the boundary and how it is to be updated at the end of an integration. It can be one of:

- **DAGHNoBoundary** no external boundary; do nothing.
- **DAGHBoundaryConst** set boundary to a constant value (see below on how to set this value)
- **DAGHBoundaryRegular** same as above
- **DAGHBoundaryShift** shift value from the inner zone to the boundary
- **DAGHBoundaryPeriodic** periodic boundary - not defined at present
- **DAGHBoundaryUserDef** call user defined function to update the boundary

The boundary width or number of cells that make up the boundary is set as follows:

SetBoundaryWidth (GridHierarchy const& GH, const int width);

The boundary value, or the constant, that is used to update the boundary is set as follows:

SetBoundaryValue (GridHierarchy const& GH, const (Type) value);

Type int or double

The degree of refinement can be set by specifying the refinement factor. Currently only refinement factors that are powers of 2 can be used.

SetRefineFactor (GridHierarchy const& GH, const int rfactor);

rfactor refinement factor that must be an integer power of 2

In this example we would like to set up a 2-dimensional grid hierarchy. The values of the function we are computing are defined at the vertices. We want to set the maximum level of refinement at 10. The base grid and subsequent finer grids have a boundary of one cell width. At the end of an integration cycle the cells on the boundary are treated no differently from the cells in the interior.

There are two coordinate systems involved. One is associated with the grid, which has a finite number of discrete points. The coordinates of the grid are integers. The other coordinate system is associated with the function that we are integrating and the coordinates are real valued. The function **SetBaseGrid** ties the two coordinate systems together. The computational domain of the function has a lower left bound of (0.0, 0.0) and a upper right bound of (1.0, 1.0). The base grid, on the other hand, has 33 grid points or 32 grid cells along each coordinate axis.

```
const int dim = 2;           // number of dimensions
const int maxlev = 10;       // maximum number of refinement levels
const int width = 1;         // width of boundary
const int rfactor = 2;       // refinement factor

int nx = 33;                 // number of grid points along x
int ny = 33;                 // number of grid points along y
int shape[dim];              // size of grid
```

```
shape[0] = nx - 1;          // number of grid cells along x
shape[1] = ny - 1;          // number of grid cells along y

double xmin = 0.0;          // lower x bound of function
double xmax = 1.0;          // upper x bound of function
double ymin = 0.0;          // lower y bound of function
double ymax = 1.0;          // upper y bound of function
double bb[2*dim];           // bounding box of function
bb[0] = xmin;
bb[1] = xmax;
bb[2] = ymin;
bb[3] = ymax;

GridHierarchy GH (dim, DAGHVertexCentered, maxlev);
SetBaseGrid (GH, bb, shape);
SetRefineFactor (GH, rfactor);
SetBoundaryWidth (GH, width);
SetBoundaryType (GH, DAGHNoBoundary);
ComposeHierarchy (GH);      // distribute grid over n processors
```

The new function here **ComposeHierarchy** distributes the grid data structure over the number of processors specified by the user.

3.2.3 Grid Function

Grid Function abstraction represents an application variable defined on the computational domain. A Grid Function is defined on and associated with the Grid Hierarchy. Distributed storage for the Grid Function is automatically allocated and maintained as the Grid Hierarchy evolves. Operations on the Grid Functions are performed in a data-parallel fashion at specified levels of the hierarchy. A Grid Function is defined as follows:

GridFunction (Dimension) GF (const char *name, GridHierarchy &GH);

GridFunction (Dimension) GF (const char *name, const int t_sten, const int s_sten, GridHierarchy &GH, const int cflag, const int sflag, const int extghflag);

GridFunction (Dimension) GF (const char *name, const int t_sten, const int s_sten, GridHierarchy &GH, const int type, const int cflag, const int sflag, const int bflag, const int adptbflag, const int extghflag);

GridFunction (Dimension) GF (const char *name, const int t_sten, const int s_sten, GridHierarchy &GH, const int type, const int align, const int cflag, const int sflag, const int bflag, const int adptbflag, const int extghflag);

GridFunction (Dimension) GF (const char *name, const int t_sten, const int s_sten, const int time, const int level, GridHierarchy &GH, const int type, const int cflag, const int sflag, const int bflag, const int adptbflag, const int extghflag);

Dimension currently supported dimension include 1, 2, and, 3

name symbolic name associated with the Grid Function and used to identify it during I/O.

t_sten specifies the time stencil to be used. Storage is maintained for (2*t_sten+1) time levels numbered from -t_sten to t_sten.

s_sten specifies the space stencil to be used. The space stencil defines the ghost regions maintained around distributed grid blocks and also defines the inter-grid communications.

GH specifies the Grid Hierarchy that will serve as the template for the Grid Function.

type represents the type of the Grid Function. It can be one of

- DAGHCellCentered
- DAGHVertexCentered
- DAGHFaceCentered_X
- DAGHFaceCentered_Y
- DAGHFaceCentered_Z

cflag specifies the type of communication for the ghost region required by the Grid Function. Values include

- DAGHNoComm
- DAGHComm (default)
- DAGHCommFaceOnly
- DAGHCommNoFace
- DAGHCommNoCorner
- DAGHCommNoEdge
- DAGHCommCurrentTimeOnly

sflag specifies if the Grid Function has a shadow or not. It can be

- DAGHNoShadow
- DAGHHasShadow (default)

bflag specifies the type of external boundary. Valid values include

- DAGHNoBoundary
- DAGHBoundaryRegular (default)
- DAGHBoundaryShift
- DAGHBoundaryPeriodic (not fully implemented in 3D)
- DAGHBoundaryUserDef

align specifies the alignment of the Grid Function if the rank of the Grid Hierarchy is greater than the rank of the Grid Function. The values include

- DAGH_X
- DAGH_Y
- DAGH_Z
- DAGH_XY
- DAGH_YZ
- DAGH_XZ

adptbflag specifies the type of adaptive boundary update. The values are

- DAGHNoAdaptBoundary
- DAGHAdaptBoundaryInterp (default)
- DAGHAdaptBoundaryUserDef
- DAGHAdaptBoundaryBoth

extghflag specifies if there is an external ghost region around the entire domain in addition to the stencil ghost region.

- DAGHNoExternalGhost (default)
- DAGHHasExternalGhost

level specifies the level of refinement and it goes from 0 to MaxLevels-1.

We associate a Grid Function **u** of type **double** with a 2-dimensional Grid Hierarchy **GH**. We set the ghost regions or the space stencil to be one cell width. We set the time stencil to be one. The Grid Function is declared as:

```
int t_sten = 1, s_sten = 1;
GridFunction (2) <DOUBLE> u ("u", t_sten, s_sten, GH);
```

More than one Grid Function can be associated with the same Grid Hierarchy. We could have a 2-dimensional hydrodynamical code where some quantities like density and pressure are defined at the centers of cells. But there could be other quantities like velocity components that are defined on the faces. Even if the Grid Hierarchy had been declared as cell centered, the constructor for the Grid Function overrides that.

```
int t_sten = 1, s_sten = 1;
GridFunction (2) <DOUBLE> d ("d", t_sten, s_sten, GH, DAGHCellCentered);
GridFunction (2) <DOUBLE> p ("p", t_sten, s_sten, GH, DAGHCellCentered);
GridFunction (2) <DOUBLE> vx ("vx", t_sten, s_sten, GH, DAGHFaceCentered_X);
GridFunction (2) <DOUBLE> vy ("vy", t_sten, s_sten, GH, DAGHFaceCentered_Y);
```

4 Control Flow

4.1 C++ Control Flow Statements

The main driver may include any C++ control flow statements. These may include:
if (expression) statement
if (expression) statement **else** statement
switch (expression) statement

while (expression) statement
do statement **while** (expression);
for (init-statement; expression; expression) statement

case constant-expression : statement
default : statement
break;
continue;

return $expression_{opt}$;

goto identifier;
identifier : statement

4.2 DAGH Control Flow Statements

DAGH control flow statements are operators defined by C++ classes that constitute the DAGH runtime system. Data parallel operations on all component grids (c) at a particular time (time) and level (level) of a Grid Function (u) can be defined using the *forall* operator.
forall (u, time, level, c) statement-list **end_forall**

Inter-grid operations can be defined directly using the *forall_levels* data parallel operator. If the default composite partitioning scheme is used, this operation is local to each processor. For example, restricting from level *level* to *level-1*, on component grids *c*, at time *time*, for Grid Function *u* can be specified as follows:
forall_levels (u, level-1, level, c)
u (time, level-1, c) = u (time, level, c);

end_forall_levels

There is another operator foreachGF that is associated with Grid Functions. This allows the user to define operations on all declared Grid Functions. The syntax for this operator is
foreachGF (GridHierarchy, GridFunction, Dimension, Type).

We have two Fortran routines prolong_amr and restrict_amr that we would like to associate with the defined grid function. This can be done by the following way:

foreachGF (GH, GF, DIM, GFTYPE)
SetProlongFunc (GF, (void *) &f_prolong_amr);
SetRestrictFunc (GF, (void *) &f_restrict_amr);
end_foreachGF

The SetProlongFunc and SetRestrictFunc have been discussed later in the manual.

5 C++ Operators

All the legal C++ operators are supported and the same precedence rules apply.

::	scope resolution	some_class::member
::	global	::variable_name
.	member selection	some_object.member
->	member selection	pointer->member
[]	subscripting	pointer [expr]
()	function call	expr (expr_list)
()	value construction	type (expr_list)
sizeof	size of object	sizeof expr
sizeof	size of type	sizeof (type)
++	pre/post increment	++lvalue, lvalue++
--	pre/post decrement	--lvalue, lvalue--
~	complement	~expr
!	not	!expr

-	unary minus	-expr
+	unary plus	+expr
\>	address of	\>lvalue
*	dereference	*expr
new	create (allocate)	new type
delete	destroy (de-allocate)	delete pointer
delete[]	destroy array	delete[] pointer
()	cast (type conversion)	(type) expr
*	multiply	expr * expr
/	divide	expr /expr
%	modulo	expr % expr
+	add	expr + expr
-	subtract	expr - expr
<<	shift left	expr << expr
>>	shift right	expr >> expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equal	expr == expr
!=	not equal	expr != expr
&	bitwise AND	expr & expr
^	bitwise exclusive OR	expr ^ expr
|	bitwise inclusive OR	expr | expr
&&	logical AND	expr && expr
||	logical inclusive OR	expr || expr
?:	conditional expression	expr ? expr : expr
=	simple assignment	lvalue = expr
*=	multiply and assign	lvalue *= expr
/=	divide and assign	lvalue /= expr
%=	modulo and assign	lvalue %= expr
+=	add and assign	lvalue += expr
-=	subtract and assign	lvalue -= expr
<<=	shift left and assign	lvalue <<= expr
>>=	shift right and assign	lvalue >>= expr
&=	AND and assign	lvalue &= expr

`|=`	inclusive OR and assign	lvalue `|=` expr
`^=`	exclusive OR and assign	lvalue `^=` expr
,	comma (sequencing)	expr , expr

6 Grid Geometry Operators

6.1 Coords and DCoords Operators

The following operators are defined on a Coords (DCoords) object:

- Arithmetic operators +, -, *, /, %, `+=`, `-=`, `*=`, `/=`, and `%=` between two Coords (DCoords) objects having the same rank or dimension and between a Coords (DCoords) object and an integer. When the operation is between a Coords (DCoords) object and an integer each component of the Coords (DCoords) gets operated on by the same integer.

- Unary negation operator -. Each component of the Coords (DCoords) gets negated.

- Logical operators `==`, `<`, `>`, `!=` between two Coords (DCoords) objects having the same rank or dimension. The operator is applied to the corresponding components of the two Coords (DCoords) objects.

The following query operators are defined for Coords objects a and b:

Coords max(a,b) Returns a Coords object having the maximum of each of the coordinates of a and b.

Coords min(a,b) Returns a Coords object having the minimum of each of the coordinates of a and b.

The corresponding query operators for DCoords are: DCoords max(a,b), and DCoords min(a,b). The following piece of code and its output illustrates the effect of some of these operators.

```
Coords a = (3, 10, 15, 30);
Coords b = (3, 5, 20, 25);
Coords d = max (a, b);
cout << "a + b = " << a + b << endl;
cout << "d = " << d << endl;
```

The output would be

```
a + b = (15, 35, 55);
d = (10, 20, 30);
```

6.2 BBox Operators

6.2.1 Member Functions that modify a BBox object

These member functions modify a given Bounding Box *bb* by increasing or decreasing its dimensions, by shifting the bounding box, and by refining or coarsening the step size. To illustrate the member functions refer to the bounding boxes defined below:

```
Coords a (2, 1, 1);
short int b[4] = {1, 1, 1, 1};
BBox bb1 (2, 4, 4, 10, 10, 2, 2);
BBox bb2 (2, 5, 7, 8, 15, 1, 2);
```

To print out the defining limits of a bounding box do

```
cout << "bb1 = " << bb1 << endl;
```

The output would be:

```
bb1 = [(4,4),(10,10),(2,2)]
```

Change the dimension of a BBox Increase the lower and upper bounds by a constant amount for a BBox object *bb* whose step size is the same in all dimensions.
bb.accrete(int const amount)
bb.accrete(Coords const c)
bb.grow(int const amount)
bb.grow(Coords const c)

```
bb1.accrete(2): bb1 = [(0,0),(14,14),(2,2)]
bb1.accrete(a): bb1 = [(2,2),(12,12),(2,2)]
bb1.grow(2): bb1 = [(0,0),(14,14),(2,2)]
bb1.grow(a): bb1 = [(2,2),(12,12),(2,2)]
```

Increase or decrease the lower and upper bounds by different amounts for a BBox object *bb* whose step size is not the same in all dimensions.
bb.growbydim(const short* c)
bb.shrinkbydim(const short* c)

```
bb2.growbydim(b): bb2 = [(4,5),(9,17),(1,2)]
bb2.shrinkbydim(b): bb2 = [(6,9),(7,13),(1,2)]
```

Increase the upper bounds of a BBox object *bb* which has the same step size in all the dimensions by a constant amount.
bb.growupper(const int i)
bb.growupper(Coords const &c)
bb.growupper(const int d, const int i)

```
bb1.growupper(2): bb1 = [(4,4),(14,14),(2,2)]
bb1.growupper(a): bb1 = [(4,4),(12,12),(2,2)]
bb1.growupper(0, 3): bb1 = [(4,4),(16,10),(2,2)]
```

Increase the lower bounds of a BBox object *bb* which has the same step size in all dimensions by a constant amount.
bb.growlower(const int i)
bb.growlower(Coords const &c)
bb.growlower(const int d, const int i)

```
bb1.growlower(2): bb1 = [(8,8),(10,10),(2,2)]
bb1.growlower(a): bb1 = [(6,6),(10,10),(2,2)]
bb1.growlower(0, 3): bb1 = [(10,4),(10,10),(2,2)]
```

Increase the upper or lower bound by different amounts of a BBox object *bb* whose step size is not the same in all dimensions.
bb.growupperbydim(const short* c)
bb.growlowerbydim(const short* c)

```
bb2.growupperbydim(b): bb2 = [(5,7),(9,17),(1,2)]
bb2.growlowerbydim(b): bb2 = [(6,9),(8,15),(1,2)]
```

Decrease the upper or lower bound by different amounts of a BBox object *bb* whose step size is not the same in all dimensions.
bb.shrinkupperbydim(const short* c)
bb.shrinklowerbydim(const short* c)

```
bb2.shrinkupperbydim(b): bb2 = [(5,7),(7,13),(1,2)]
bb2.shrinklowerbydim(b): bb2 = [(4,5),(8,15),(1,2)]
```

Shift the position of a BBox Shift the BBox object *bb* by a constant amount in all dimensions.
bb.shift(const int i)
bb.shift(Coords const &c)
Shift the BBox object *bb* by a constant amount in dimension d by a constant amount.
bb.shift(const int d, const int i)

```
bb1.shift(2): bb1 = [(8,8),(14,14),(2,2)]
bb1.shift(a): bb1 = [(6,6),(12,12),(2,2)]
bb1.shift(0, 2): bb1 = [(8,4),(14,10),(2,2)]
```

Change the step size of a BBox Increase the step size in all dimensions of BBox object *bb* by a constant amount.
bb.coarsen(const int by)
bb.coarsen(Coords const &by)
Increase the step size along dimension d of the BBox object *bb* by a constant amount.
bb.coarsen(const int d, const int by)

```
bb1.coarsen(2): bb1 = [(4,4),(8,8),(4,4)]
bb1.coarsen(a): bb1 = [(4,4),(10,10),(2,2)]
bb1.coarsen(0,2): bb1 = [(4,4),(8,10),(4,2)]
```

In the refinement process the step size is decreased. Cell centered and vertex centered grids are treated differently. The overlap parameter, *olap*, is 1 for a cell centered bounding box and 0 for a vertex centered bounding box. *Note that olap is defined differently for the class Grid Hierarchy.*
This is valid only for a VertexCentered BBox *bb*. The step size is decreased by a constant amount in all dimensions.
bb.refine(const int by)
bb.refine(Coords const &by)
The step size is decreased by a constant amount in all dimensions, for either cell centered box (olap = 1) or for a vertex centered box (olap = 0).

bb.refine(const int by, const int olap)
The step size is decreased by a constant amount only along dimension d, for either cell centered box (olap = 1) or for a vertex centered box (olap = 0).
bb.refine(const int d, const int by, const int olap)

```
bb1.refine(2): bb1 = [(4,4),(10,10),(1,1)]
bb1.refine(a): bb1 = [(4,4),(10,10),(2,2)]
bb1.refine(2, 1): bb1 = [(4,4),(11,11),(1,1)]
bb1.refine(0, 2, 1): bb1 = [(4,4),(11,10),(1,2)]
```

6.2.2 Operators that create a new Bounding Box

These functions create new BBox object from the old BBox object *bb* by modifying its dimensions, shifting its position, or changing the grid step size.

Create new BBox by changing the size of the old Returns a new BBox object by increasing the lower and upper bounds of *bb* by a constant amount. *bb* has the same step size in all dimensions.
BBox accrete(bb, int const amount)
BBox accrete(bb, Coords const c)
BBox grow(bb, int const amount)
BBox grow(bb, Coords const c)

```
bb3 = accrete (bb1, 2): bb3 = [(0,0),(14,14),(2,2)]
bb3 = accrete (bb1, a): bb3 = [(2,2),(12,12),(2,2)]
bb3 = grow (bb1, 2): bb3 =  [(0,0),(14,14),(2,2)]
bb3 = grow (bb1, a): bb3 = [(2,2),(12,12),(2,2)]
```

Returns a new BBox object by increasing or decreasing the lower and upper bounds of *bb* by different amounts. *bb* may not have the same step size in all dimensions.
BBox growbydim(bb, const short* c)
BBox shrinkbydim(bb, const short* c)

```
bb4 = growbydim (bb2, b): bb4 = [(4,5),(9,17),(1,2)]
bb4 = shrinkbydim (bb2, b): bb4 = [(6,9),(7,13),(1,2)]
```

Returns a new BBox object by increasing the upper bounds of *bb* by a constant amount. *bb* has the same step size in all the dimensions.
BBox growupper(bb, const int i)
BBox growupper(bb, const int d, const int i)

```
bb3 = growupper (bb1, 2): bb3 = [(4,4),(14,14),(2,2)]
bb3 = growupper (bb1, 0, 3): bb3 = [(4,4),(16,10),(2,2)]
```

Returns a new BBox object by increasing the lower bounds of BBox object *bb* in all dimensions by a constant amount. *bb* has the same step size in all dimensions.
BBox growlower(bb, const int i)
BBox growlower(bb, const int d, const int i)

```
bb3 = growlower (bb1, 2): bb3 = [(8,8),(10,10),(2,2)]
bb3 = growlower (bb1, 0, 3): bb3 = [(10,4),(10,10),(2,2)]
```

Returns a new BBox object by increasing the upper or lower bounds of BBox object *bb* in different amounts in different dimensions. *bb* may not have the same step size in all dimensions.
BBox growupperbydim(bb, const short* c)
BBox growlowerbydim(bb, const short* c)

```
bb4 = growupperbydim (bb2, b): bb4 = [(5,7),(9,17),(1,2)]
bb4 = growlowerbydim (bb2, b): bb4 = [(6,9),(8,15),(1,2)]
```

Returns a new BBox object by decreasing the upper or lower bound of BBox object *bb* by different amounts in different dimensions. *bb* may not have the same step size in all dimensions.
BBox shrinkupperbydim(bb, const short* c)
BBox shrinklowerbydim(bb, const short* c)

```
bb4 = shrinkupperbydim (bb2, b): bb4 = [(5,7),(7,13),(1,2)]
bb4 = shrinklowerbydim (bb2, b): bb4 = [(4,5),(8,15),(1,2)]
```

Create new BBox by shifting the old Returns a new BBox object by shifting the BBox object *bb* by a constant amount in all dimensions.
BBox shift(bb, const int i)
BBox shift(bb, Coords const &c)
Returns a new BBox object by shifting the BBox object *bb* by a constant amount along dimension d.
BBox shift(bb, const int d, const int i)

```
bb3 = shift (bb1, 2): bb3 = [(8,8),(14,14),(2,2)]
bb3 = shift (bb1, a): bb3 = [(6,6),(12,12),(2,2)]
bb3 = shift (bb1, 0, 2): bb3 = [(8,4),(14,10),(2,2)]
```

Create a new BBox by changing the step size of the old Returns a new BBox object by increasing the step size of BBox object *bb* by a constant amount in all dimensions.
BBox coarsen(bb, const int by)
BBox coarsen(bb, Coords const &by)
Returns a new BBox object by increasing the step size of BBox object *bb* along dimension d by a constant amount.
BBox coarsen(bb, const int d, const int by)

```
bb3 = coarsen (bb1, 2): bb3 = [(4,4),(8,8),(4,4)]
bb3 = coarsen (bb1, a): bb3 = [(4,4),(10,10),(2,2)]
bb3 = coarsen (bb1, 0, 2): bb3 = [(4,4),(8,10),(4,2)]
```

The step size is decreased in the refinement process. Vertex centered and cell centered grids are treated differently. The overlap parameter, *olap*, is 0 for a vertex centered bounding box and 1 for a cell centered bounding box. *Note that olap is defined differently for the class Grid Hierarchy.* The function returns a new BBox object obtained by decreasing the step size of a VertexCentered BBox *bb* by a constant amount in all dimensions.
BBox refine(bb, const int by)
BBox refine(bb, Coords const &by)
This function returns a new BBox object obtained from *bb* by decreasing its step size by a constant amount in all dimensions for either a vertex centered box (olap = 0) or for a cell centered box (olap = 1).
BBox refine(bb, const int by, const int olap)

This function returns a new BBox object obtained from *bb* by decreasing its step size by constant amount only along dimension d for either a vertex centered box (olap = 0) or for a cell centered box (olap = 1).
BBox refine(bb, const int d, const int by, const int olap)

```
bb3 = refine (bb1, 2): bb3 = [(4,4),(10,10),(1,1)]
bb3 = refine (bb1, a): bb3 = [(4,4),(10,10),(2,2)]
bb3 = refine (bb1, 2, 1): bb3 = [(4,4),(11,11),(1,1)]
bb3 = refine (bb1, 0, 2, 1): bb3 = [(4,4),(11,10),(1,2)]
```

6.2.3 Binary operators

These are operators defined between BBox objects *bb1* and *bb2*. Let us consider the following two bounding boxes to illustrate these binary operators:

```
BBox bb1 = [(4,4),(10,10),(1,2)]
BBox bb2 = [(5,7),(8,15),(1,1)]
```

Assignment bb1 = bb2

Union bb1 + bb2 and bb1 += bb2

Intersection bb1 * bb2 and bb1 *= bb2

Logical operators `==`, `!=`, `<`, `<=`, `>`, and `>=`

```
bb5 = bb1 + bb3: bb5 = [(4,4),(10,10),(1,2)]
bb5 = bb2 + bb1; bb5 = [(4,4),(10,15),(1,1)]
bb5 = bb1 * bb2: bb5 = [(5,7),(8,10),(1,2)]
```

6.2.4 Query operators

There are several query operators defined on a BBox object *bb*.

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

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

int lower(bb,i) Returns the lower bound of the bounding box in the ith dimension

int upper(bb,i) Returns the upper bound of the bounding box in the ith dimension

Coords stepsize(bb) Returns the step sizes of the bounding box in all the respective dimensions

int stepsize(bb,i) Returns the stepsize of the bounding box in the ith dimension

Coords extents(bb) Returns the extent of the bounding box in each of the respective dimensions

int extents(bb,i) Returns the extent of the bounding box in the ith dimension

int bb.empty() Returns 1 if the bounding box is empty and 0 otherwise

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

int bb1.mergable(bb2,i,olap) Returns 1 if the bounding boxes bb1 and bb2 are mergable in the ith dimension, where olap, the overlap parameter, is 0 for vertex centered box and 1 for cell centered box

int inside(bb,c) Returns 1 if Coords c is contained by the bounding box, could be on an edge

int bb.interior(c) Returns 1 if Coords c is contained by the bounding box, but not on an edge

To illustrate some of the query operators let us introduce two more bounding boxes *bb3* and *bb4* and coords object *a* and *b*.

```
BBox bb3 = [(4,5),(6,8),(1,1)]
BBox bb4 = [(6,5),(10,8),(1,1)]
Coords a = (6,6)
Coords b = (5,7)

Coords c = lower (bb2): c = (5,7)
```

```
Coords c = upper (bb2): c = (8,15)
int x = lower (bb2, 0): x = 5
int y = lower (bb2, 1): y = 7
int x = upper (bb2, 0): x = 8
int y = upper (bb2, 1): y = 15

Coords c = stepsize (bb1): c = (1,2)
int x = stepsize (bb1): x = 1
int y = stepsize (bb1,1): y = 2

Coords c = extents (bb2): c = (4,9)
int x = extents (bb2, 0): x = 4
int y = extents (bb2, 1): y = 9

int x = size (bb1): x = 28

bb3.mergable (bb4, 0, 1) would return 1.

inside (bb3, a) would return 1.
bb3.interior (b) would return 1.
```

6.3 BBoxList Operators

6.3.1 Operators that modify a BBox List

These operators modify a BBox List by adding or removing Bounding Boxes from the List.

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

insert(bblist, bb) Insert BBox bb after the current BBox

remove(bblist) Remove the current BBox

After the *add* or *insert* operation the current BBox is the one just added or inserted. The functionalities of the two operators appear to be the same unless the pointer to the cuurent BBox is changed by one of the operators below. After the *remove* operation the current BBox is the one before in the list or the one after.

6.3.2 Operators that return a BBox from a List

These operators return a pointer to a bounding box from a list of bounding boxes depending on its position in the list. The pointer to the current BBox assumes this value.

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 previous BBox in the list

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

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

6.3.3 Query Operators

The following query functions are defined on a BBoxList object bblist:

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

int isempty(bblist) Returns a 1 if the list is empty or 0 otherwise

The following piece of code illustrates the use of some of these operators:

```
BBox bb1 (2, 1, 2, 3, 4, 1, 1);
BBox bb2 (2, 4, 3, 5, 7, 1, 1);
BBox bb3 (2, 5, 3, 8, 9, 2, 2);
BBox bb4 (2, 6, 2, 9, 7, 2, 1);
BBox bb5 (2, 3, 2, 5, 9, 1, 1);
BBoxList bblist;
BBox* bxptr;

add (bblist, bb1);
add (bblist, bb2);
add (bblist, bb3);
add (bblist, bb4);

cout << "bblist is: " << bblist << endl;
```

```
bxptr = current (bblist);
cout << "Current bbox is: " << *bxptr << endl;

bxptr = first (bblist);
cout << "First bbox is: " << *bxptr << endl;
bxptr = current (bblist);
cout << "Current bbox is: " << *bxptr << endl;

insert (bblist, bb5);
cout << "bblist is: " << bblist << endl;

bxptr = current (bblist);
cout << "Current bbox is: " << *bxptr << endl;

bxptr = next (bblist);
cout << "Next box is: " << *bxptr << endl;

bxptr = prev (bblist);
cout << "Previous box is: " << *bxptr << endl;

remove (bblist);
cout << "Number of bounding boxes in the list = ";
cout << number(bblist) << endl;

bxptr = current (bblist);
cout << "Current bbox is: " << *bxptr << endl;

bxptr = last (bblist);
cout << "Last box is: " << *bxptr << endl;
```

The output of the code is given below. Blank lines have been introduced for greater readability.

```
bblist is: Num:4
[(1,2),(3,4),(1,1)]
```

```
*********************************
[(4,3),(5,7),(1,1)]
*********************************
[(5,3),(8,9),(2,2)]
*********************************
[(6,2),(9,7),(2,1)]
*********************************

Current bbox is: [(6,2),(9,7),(2,1)]
First bbox is: [(1,2),(3,4),(1,1)]
Current bbox is: [(1,2),(3,4),(1,1)]

bblist is: Num:5
[(1,2),(3,4),(1,1)]
*********************************
[(3,2),(5,9),(1,1)]
*********************************
[(4,3),(5,7),(1,1)]
*********************************
[(5,3),(8,9),(2,2)]
*********************************
[(6,2),(9,7),(2,1)]
*********************************

Current bbox is: [(3,2),(5,9),(1,1)]
Next box is: [(4,3),(5,7),(1,1)]
Previous box is: [(3,2),(5,9),(1,1)]

Number of bounding boxes in the list = 4
Current bbox is: [(1,2),(3,4),(1,1)]

Last box is: [(6,2),(9,7),(2,1)]
```

7 Grid Hierarchy Operators

7.1 Additional Functions to Modify Base Grid Hierarchy

Given a Grid Hierarchy *GH*, there are additional functions to modify the base grid.

An external ghost region can be specified. This is an additional region, around each component grid, that is used for flux computations. It is storage given to the application to be used as desired. The default width of this region is 0.
SetExternalGhostWidth (GridHierarchy const &GH, const int width)

The adaptive boundary type of the grid hierarchy can be specified.
SetAdaptBoundaryType (GridHierarchy const &GH, const int type)
where *type* is one of the following:

- DAGHNoAdaptBoundary
- DAGHAdaptBoundaryInterp (default)
- DAGHAdaptBoundaryUserDef
- DAGHAdaptBoundaryBoth

You can specify how you want the grid to be distributed - along a particular axis or a set of axis or as per a list of bounding boxes. The distribution type you specify is enforced for uni-grid applications, but is only a suggestion for AMR applications.
SetDistributionType (GridHierarchy const &GH, const int type)
SetDistributionType (GridHierarchy const &GH, const int type, BBoxList &bbl)
The distribution *type* can be any one of the following:

- DAGHCompositeDistribution - using space filling curves
- DAGHBlockXDistribution
- DAGHBlockYDistribution

- DAGHBlockZDistribution
- DAGHBlockXYDistribution
- DAGHBlockYZDistribution
- DAGHBlockXZDistribution
- DAGHBlockAllDistribution
- DAGHUserDefDistribution

A work function computes the work associated with a grid component and is used during distribution. The default assumes that work is equal to the number of grid points (or cells, vertices, faces) in the component. A user defined work function *wf* can be specified.
SetWorkFunction (GridHierarchy const &GH, void* wf)

You can distribute the Grid Functions in the hierarchy over multiple processors. This is useful when you want to customize a Grid Function rather than use one of the predefined constructors that allow only certain combinations of settings.
ComposeGridFunctions (GridHierarchy const &GH)

7.2 Set up the Distributed Data Structures

Composing Grid Hierarchy sets up the distributed data structures and initializes the information required to maintain this structure.
ComposeHierarchy (GridHierarchy const &GH)

7.3 Refine the Grid Hierarchy

The Grid Hierarchy is refined at a particular level *Level* by overlaying a refined grid over specified regions of the existing component grids at that level. The specified regions are denoted by the list of Bounding Boxes *bblist*. The regions need not be contiguous. When no Bounding Box list is specified, the whole grid at that level is refined.
Refine (GridHierarchy const &GH, const int Level)
Refine (GridHierarchy const &GH, BBoxList& bblist, const int Level)

7.4 Recompose the Grid Hierarchy

The recomposition function does several tasks. It reconstructs a new Grid Hierarchy. It repartitions and redistributes the new hierarchy to balance the load, and optimize inter-grid and intra-grid communications to maintain locality. It performs all data transfers to reconstruct the Grid Functions associated with the Grid Hierarchy. It performs the prolongations to initialize the Grid Functions of the new component grids using the prolongation function associated with the particular Grid Function.
RecomposeHierarchy (GridHierarchy const &GH)

7.5 Redistribute the Grid Hierarchy

Redistribute is pertinent for unigrid applications. It allows you to change the distribution type on the fly. If you have a grid blocked in the X direction and you want to block it in the Y direction or in both X and Y directions you can use the redistribute call. Redistribute is applied to the list of Bounding Boxes *bbl* specified, otherwise it is applied to the whole grid.
RedistributeHierarchy (GridHierarchy const &GH, const int type)
RedistributeHierarchy (GridHierarchy const &GH, const int type, BBoxList& bbl)
The permitted distribution *type* includes:

- DAGHCompositeDistribution - using space filling curves
- DAGHBlockXDistribution
- DAGHBlockYDistribution
- DAGHBlockZDistribution
- DAGHBlockXYDistribution
- DAGHBlockYZDistribution
- DAGHBlockXZDistribution
- DAGHBlockAllDistribution
- DAGHUserDefDistribution

7.6 Checkpoint a Grid Hierarchy

DAGH allows you to checkpoint at any point in the application. The checkpoint file contains the current state of the Grid Hierarchy and associated Grid Function data. Each Grid Function has a checkpoint flag which determines whether it is checkpointed or not. By default all Grid Functions are checkpointed.

Set the checkpoint flag for a Grid Function *GF*
SetCheckpointFlag (GF, flag)
where flag = DAGHTrue or DAGHFalse. The default value of the flag is DAGHTrue.

Checkpoint the Grid Hierarchy *GH*
Checkpoint (GridHierarchy const &GH, const char* name)
where name (e.g. name = "/home/user/checkpoints/step10") is the leading part of the checkpoint file name. DAGH will add extensions to this filename. The filename can be a path. DAGH will write one checkpoint file per processor.

One can rollback to a previous state by calling *RecomposeHierarchy* with the checkpoint file as an argument. This can be done at any point in the application program. It resets the state of the Grid Hierarchy to that in the checkpoint file and initializes the data accordingly.
RecomposeHierarchy (GridHierarchy const &GH, const char* name)
where name is the checkpoint file that was used in the call to checkpoint.

One can restart computation from a stored state in a checkpoint file. This is done by calling *ComposeHierarchy* with the checkpoint file. DAGH sets the initial state of the Grid Hierarchy to that in the checkpoint file. For each Grid Function declared in the application, DAGH checks to see if any data information is available in the checkpoint file and if so reads it to initalize the Grid Function. If no information is present in the check point file (i.e. if the GF was not checkpointed or did not exist in the previous calculation) then it is applications responsibility to initalize it. Note the DAGH indexes into the checkpoint file using the Grid Function name. This restart can be used to restart the computation on a different number of processors and also

with more or fewer Grid Functions.
ComposeHierarchy (GridHierarchy const &GH, const char* name)
where *name* is the name of the checkpoint file.

7.7 Functions and Query operators on a Grid Hierarchy

7.7.1 DAGH Type

This query operator returns the type of each component grid in the hierarchy.
int **GH.dagh_type**()
The following types are returned. The integer values corresponding to these types are noted in parenthesis.

- DAGHCellCentered (0)
- DAGHVertexCentered (1)
- DAGHFaceCentered_X (2)
- DAGHFaceCentered_Y (3)
- DAGHFaceCentered_Z (4)

7.7.2 Initialize the overlap flag array

This function initializes the overlap flag array *olap*. The three elements of the array are set to 0 for cell centered box and 1 for vertex centered box. *Note that the olap parameter is defined differently in the class Bounding Box.* For face centered box the corresponding dimension is set to 1 and the others to 0. For example, if type = DAGHFaceCentered_X then olap[0] = 1, and olap[1] = olap[2] = 0.
GH.DAGH_InitOverlaps (const int type, short* olap)

7.7.3 Boundary/Ghost region

The following query operators return the width or number of cells of the boundary and the ghost region.

int **GH.boundarywidth** ()
int **GH.externalghostwidth** ()

The following query operators return the boundary and adaptive boundary type.
int **GH.boundarytype** ()
int **GH.adaptboundarytype** ()
where the boundary type could be any of the following, the corresponding returned integer values are noted in parenthesis:

- DAGHNoBoundary (0)
- DAGHBoundaryConst (1)
- DAGHBoundaryRegular (1)
- DAGHBoundaryShift (2)
- DAGHBoundaryPeriodic (3)
- DAGHBoundaryUserDef (4)

and the adaptive boundary type could be any of the following, the corresponding integer values that are returned are noted in parenthesis:

- DAGHNoAdaptBoundary (0)
- DAGHAdaptBoundaryInterp (1)
- DAGHAdaptBoundaryUserDef (2)
- DAGHAdaptBoundaryBoth (3)

7.7.4 Information on levels

The coarsest level or base level has an index of 0. The next higher level of refinement is 1, and so on. If you specify the maximum number of levels to be 4, then the query *totallevels* will return 4, but *maxlevelindex* will return 3. These query operators return the level index number. Internally DAGH uses a different numbering scheme.

int GH.totallevels () returns the total levels in the Grid Hierarchy

int GH.coarselevelindex () returns the under lying coarse level index

int GH.finelevelindex () returns the over lying fine level index

int GH.maxlevelindex () returns the index of the maximum refinement level

7.7.5 Space and Time steps

The following query operators return information regarding the space and time steps on the main and shadow hierarchy. The parameter *ident* can take on values *DAGH_Main* or *DAGH_Shadow*. The default value is *DAGH_Main*. The parameter *level* refers to the level of refinement.

int GH.stepsize (const int level, const int ident) returns the step size for a given level in the hierarchy

int GH.step2level (const int s, const int ident) for a given a refinement factor this utility returns the level corrsponding to a step size *s*

int GH.timestep (const int level, const int ident) returns the time step taken on a given level in the hierarchy

int GH.numtsteps (const int level, const int ident) returns the number of space steps taken for a given level in the hierarchy

int GH.stepstaken (const int t, const int level, const int ident) returns the number time steps taken at time *t* for a given level in the hierarchy

double GH.delta_t (const int level, const int ident) returns the time interval at a given level in the hierarchy

double GH.delta_x (const int dim, const int level, const int ident) returns the space interval along the dimension *dim* for a given level in the hierarchy

GH.setupdatedvaluestep (const int ustep) In multi-time step schemes (e.g. leap frog with 3 time steps) the updated value can be at the advanced time step or at the current time step when storage is cycled (next to current, current to previous) at the end of a step. This function allows you to set the updated value step *ustep*. *ustep* can take on values *DAGHCurrentTime*(0), *DAGHNextTime*(1), or *DAGHPrevious-Time*(2), the corresponding integer values are noted in parenthesis.

int GH.updatedvaluestep ()this query operator returns the integer corresponding to the updated value step defined above

The shadow hierarchy is an exact 2:1 coarsening on the main hierarchy. Not only is the space step doubled in the shadow hierarchy, but the time step is also doubled. Let us assume that you specified a refinement factor of 2 and the maximum number of levels to be 4. The base level would have an index of 0 and the level of maximum refinement would have an index of 3. On the main hierarchy, the space step and time step of the base level would be $2^3(=8)$. The next level of refinement higher (having an index of 1) would have a space step and time step of $2^2(=4)$. The maximum level of refinement (having an index of 3) would have a space step and time step of $2^0(=1)$. On the shadow hierarchy, the space step and time step of the base level would be 16. The next level of refinement higher in the shadow hierarchy would have a space step and time step of 8. The corresponding number for the maximum level of refinement on the shadow hierarchy would be 2.

7.7.6 Refinement Factors

The following query operators return information on the refinement level and factor.

int GH.refinefactor () returns the refinement factor

int GH.refinedby (const int level) returns the total refinement factor (as compared to the coarse grid) corresponding to the given level

Taking a maximum level of refinement to be 4 and the refinement factor to be 2, *refinedby* would return 1 for level 0 and 8 for level 3.

7.7.7 Set or get time values

The following functions and query operators can be used to set or get information about time values. The parameter *ident* can take on values *DAGH_Main* or *DAGH_Shadow*.

int GH.getCurrentTime (const int level, const int ident) returns the current time for a given level on the hierarchy

GH.setCurrentTime (const int ctime, const int level, const int ident) sets the current time to *ctime* for a given level in the hierarchy

GH.incrCurrentTime (const int level, const int ident) increments the current time by one time step for that level in the hierarchy

GH.getPreviousTime (const int level, const int ident) returns the previous time which is the current time minus the time step for that level in the hierarchy

int GH.getNextTime (const int level, const int ident) returns the next time level which is the current time plus the time step for that level in the hierarchy

setCurrentTime and *incrCurrentTime* sets or increments the current time for the level specified. These operators do *not* change the time values for other levels.

7.8 DAGH structure and coordinate information

There are two systems of coordinates to denote points in a grid - one is the system that DAGH uses internally and the other the actual coordinates as specified in the computational domain. The internal system is called *Integer Coordinates* and the application specific system the *World Coordinates.* For example the lower bound of a 2 dimensional grid can be specified as (0,0) in Integer Coordinates, whereas its World Coordinates could be (2.35, -5.79). The World Coordinates are of type *double.*

7.8.1 Integer Coordinates

Coords GH.localCoords (double const *wc) converts the world coordinates to integer coordinates and returns a Coords object

Coords GH.localStep (double const *ws) converts the step size in world coordinates to integer coordinates and returns Coords object

7.8.2 World Coordinates

DCcords GH.worldCoords (const int* lc, const int* ls) converts integer coordinates *lc* and step size *ls* to world coordinates

DCoords GH.worldStep (const int* ls, double* ws) converts the integer coordinates step size into world coordinates step size

Consider a grid hierarchy defined by the following piece of code:

```
  INTEGER MaxLev = 4;         // Maximum level of refinement
  INTEGER Rfac = 2;           // Refinement factor

//  Grid Properties
  DOUBLE xmin = 0.0;          // Lower x bound
  DOUBLE xmax = 1.0;          // Upper x bound
  DOUBLE ymin = 0.0;          // Lower y bound
  DOUBLE ymax = 1.0;          // Upper y bound
  INTEGER nx = 33;            // Grid points along x
  INTEGER ny = 33;            // Grid points along y

//  Set up 2-dimensional grid hierarchy
  INTEGER shape[DIM];
  DOUBLE bb[2*DIM];

  shape[0] = nx - 1;
  shape[1] = ny - 1;

  bb[0] = xmin;
  bb[1] = xmax;
  bb[2] = ymin;
  bb[3] = ymax;

  GridHierarchy GH (DIM, DAGHVertexCentered, MaxLev);
  SetBaseGrid (GH, bb, shape);
  SetRefineFactor (GH, Rfac);
  SetBoundaryWidth (GH, 1);
  SetBoundaryType (GH, DAGHNoBoundary);
  ComposeHierarchy (GH);
```

There are 33 grid points or 32 space steps along each coordinate direction of the coarse grid. Since the grid is defined to have a maximum level of refinement of 4 and a refinement factor of 2, DAGH creates a grid having $32 * 2^3$ (=256) space steps along each axis. The lower and upper bounds in world coordinates would be (0.0, 0.0) and (1.0, 1.0) respectively. In integer coordinates the corresponding values would be (0,0) and (256, 256). World coordinates of (0.5, 0.5) would be (128, 128) in integer coordinates. An integer step size of 4 would be a step size of 0.015625 in world coordinates.

7.9 Bounding Box information

BBox GH.glbbbox () reurns a Bounding Box object containing the complete computational domain including the boundary

7.10 Multigrid Operators

int MultiGridLevels (GridHierarchy const &GH, const int level, const int ident) this query operator returns how many multigrid levels one can have for a given level. It is defined by the smallest component grid.

7.11 Ghost Communications

Ghost regions are regions maintained around distributed grid blocks for intergrid communications. At various times in the integration process values in the ghost regions are used to synchronize the boundary values of the distributed grid blocks. The *DAGH_SyncGhostRegions* function can be called with different parameters for this purpose.

GH.DAGH_SyncGhostRegions (const int time, const int level, const int ident)

GH.DAGH_SyncGhostRegions (const int time, const int level, const int mgl, const int ident)

GH.DAGH_SyncGhostRegions (const int time, const int level, const int axis, const int dir, const int ident)

GH.DAGH_SyncGhostRegions (const int time, const int level, const int mgl, const int axis, const int dir, const int ident) where

ident is the hierarchy DAGH_Main or DAGH_Shadow

level is the level of refinement

time is the current time on that level

axis is the axis along which the ghost regions are to be synchronized. It is one of

- DAGH_X
- DAGH_Y
- DAGH_Z
- DAGH_All

dir is the direction along which to synchronize the ghost regions and it can be one of

- DAGH_Forward
- DAGH_Backward
- DAGH_Both

mgl is the multigrid level (possibly within an AMR hierarchy)

8 Grid Function Operators

In the discussion of this sections and the ones to follow there are several references to Type, GFType, GFTYPE, or DAGH_GFType. These are place holders for the following types: INTEGER, REAL, DOUBLE, and CHARACTER.

8.1 Attributes of a Grid Function

8.1.1 Setting the attributes of a Grid Function

These member functions can be used to set the following predefined functions.

GF_SetInitFunc ((void*) &initf) is used to specify the function *initf* to initalize the Grid Function

GF_SetUpdateFunc ((void*) &uf) is used to specify the function *uf* to update the Grid Function

GF_SetBndryUpdateFunc ((void*) &bf) is used to specify the function *bf* to update the boundary

GF_SetAdaptiveBndryUpdateFunc ((void*) &abf) is used to specify the function *abf* to update the adaptive boundary

GF_SetProlongFunc ((void*) &pf) is used to specify the function *pf* to perform prolongations

GF_SetRestrictFunc ((void*) &rf) is used to specify the function *rf* to perform restrictions

GF_SetIOFunc ((void*) &iof) is used to specify the function *iof* to perform I/O operations for the Grid Function

Once the respective functions have been defined, the recommended way would be to use the following operators to apply the specified function at a particular time and level for the whole grid or for a part of the grid as specified by the Bounding Box *bb*. The parameter *args* is an array of dimension *cnt* that contains arguments to be passed to the respective function. *ident* can take on values DAGH_Main or DAGH_Shadow. In case of the *Update* operator, an array *time* of dimension *tcnt* is used to specify the number of timesteps of the Grid Function that are to be used in the update. The boundary update and adaptive boundary update functions are discussed in the section on **Boundary Functions**.

GF_Init (const int time, const int level, GFType* args, const int cnt, ident)

GF_Init (const int time, const int level, const BBox &bb, GFType* args, const int cnt, ident)

GF_Update (const int time_from, const int time_to, const int level, GFType* args, const int cnt, ident)

GF_Update (const int time_from, const int time_to, const int level, const BBox &bb, GFType* args, const int cnt, ident)

GF_Prolong (const int time_from, const int level_from, const int time_to, const int level_to, GFType* args, const int cnt, ident)

GF_Prolong (const int time_from, const int level_from, const int time_to, const int level_to, const BBox &bb, GFType* args, const int cnt, ident)

GF_Restrict (const int time_from, const int level_from, const int time_to, const int level_to, GFType* args, const int cnt, ident)

GF_Restrict (const int time_from, const int level_from, const int time_to, const int level_to, const BBox &bb, GFType* args, const int cnt, ident)

GF_IO (const int time, const int level, GFType* args, const int cnt, ident)

GF_IO (const int time, const int level, const BBox &bb, GFType* args, const int cnt, ident)

8.2 Boundary Functions

8.2.1 External Boundaries

The handling of the external boundary is controlled by the following parameters:

Update Flag This flag determines whether the external boundary is updated or not

SetBndryUpdateFlag (GridFunction &gf, const int flag)

where flag is either DAGHTrue or DAGHFalse. The default being DAGHTrue.

Boundary Type The boundary type defines how the boundary is updated

SetBoundaryType (GridFunction &gf, const int boundary)

where boundary is one of

DAGHNoBoundary no external boundary; do nothing

DAGHBoundaryConst set to a constant value

DAGHBoundaryRegular same as above

DAGHBoundaryShift shift values from the inner zones to the boundaries

DAGHBoundaryPeriodic periodic boundary (not defined at present)

DAGHBoundaryUserDef call user defined function

Boundary Width This defines the number of zones that make up the boundary. The default value is set for the Grid Hierarchy by using the call to *SetBoundaryWidth*.

SetBoundaryWidth (GridFunction &gf, const int width)

Boundary Value This defines the constant value used to update the boundary. DAGH_GFType is a place holder for a template class. In this call the boundary type is set to DAGHBoundaryConst or DAGHBoundaryRegular. The default value for *value* is (DAGH_GFType) 0.

GF_SetBoundaryValue (const DAGH_GFType value)

Boundary Update Function The user defined external boundary update function is set using the following function:

SetBndryUpdateFunction (GridFunction &gf, void* buf)

The external boundary update function is typedefed as follows:

typedef void (*BndryUpdateFunc) (
    Type*, int*, int*, int*,
    int*, int*, int*,
    Type*, const int*)

The function gets the following pieces of information

- Grid function at the current time and level
  Type* data, int* lb, int* ub, int* shape

- region to be updated
  int* lb, int* ub, int* shape

- other arguments
  Type* args, const int arg_cnt

Boundary Updates The external and adaptive boundaries are update in DAGH by making calls to

GF_BndryUpdate (const int time, const int level, const int ident)

GF_BndryUpdate (const int time, const int level, const int mgl, const int ident)

where

ident DAGH_Main or DAGH_Shadow

mgl multigrid level in an AMR hierarchy

8.2.2 Adaptive Boundaries

The handling of adaptive boundaries is controlled by 2 parameters - an adaptive boundary update flag and an adaptive boundary type.

Update Flag The adaptive boundary update flag determines whether the adaptive boundary will be updated or not. The *flag* is DAGHTrue or DAGHFalse. The default value is DAGHTrue.

SetAdaptBndryUpdateFlag (GridFunction &gf, const int flag)

Adaptive Boundary Type The adaptive boundary type determines how the adaptive boundary will be updated.

SetAdaptBoundaryType (GridFunction &gf, const int adaptbndry)

where adaptbndry is one of

DAGHNoAdaptBoundary No adaptive boundary; do nothing

DAGHAdaptBoundaryInterp Use space-time interpolation to update the adaptive boundaries. Space prolongation uses the user defined *prolong* function that has to be defined.

DAGHAdaptBoundaryUserDef Call a user defined function to update the adaptive boundary

DAGHAdaptBoundaryBoth This uses the space-time interpolation first and then calls the user defined function

Adaptive Boundary Update Function The user defined adaptive boundary update function is set using the following function:

SetAdaptiveBndryUpdateFunction (GridFunction &gf, void* abuf)

The adaptive boundary update function is typedefed as follows:

typedef void (*AdptBndryUpdateFunc) (
    Type*, int*, int*, int*,
    Type*, int*, int*, int*, double*,
    Type*, int*, int*, int*, double*,
    int*, int*, int*,
    Type*, const int*)

where the function gets the following information

- Grid Function at the current time and level
  Type* data, int* lb, int* ub, int* shape
- Grid Function on the parent grid at ta_parent
  Type* data, int* lb, int* ub, int* shape, double* weight
- Grid Function on the parent grid at tp_parent
  Type* data, int* lb, int* ub, int* shape, double* 1.0-weight
- region to be updated
  int* lb, int* ub, int* shape
- other arguments
  Type* args, const int arg_cnt

where weight = (time - tp_parent) / (ta_parent - tp_parent)

8.2.3 Query Boundary Information

short boundary_flag (GridFunction& gf, const int time, const int level, const int c, const int dir) for a given component grid *c* and direction *dir* it returns the type of boundary - i.e. external, adaptive, or none in that direction.

const short* boundary_flag (GridFunction& gf, const int time, const int level, const int c) this returns an array of size 2*dim containing the boundary flags for the component grid *c*. The element 0 of this array is the boundary flag in the negative x direction, element 1 is the boundary flag in the positive x direction, element 2 is the boundary flag in the negative y direction, and so on.

8.3 Time Functions

In the following section we discuss the functions and query operators that are related to time. The parameter *ident* can take on values DAGH_Main or DAGH_Shadow.

8.3.1 Set or get current time information

GF_SetCurrentTime (const int ctime, const int level, const int ident) sets the current time to *ctime* of a grid function for a given level

GF_IncrCurrentTime (const int level, const int ident) increments the current time by one time step of the grid function for a given level

GF_CurrentTime (int const level, const int ident) gets the current time of the grid function at a given level

GF_PreviousTime (const int level, const int ident) gets the previous time which is the current time minus one time step of the grid function for a given level

GF_NextTime (const int level, const int ident) gets the next time which is the current time plus one time step of the grid function for a given level

8.3.2 Swap storage for time levels

GF_SwapTimeLevels (const int level, const int time1, const int time2) for a given level this swaps the storage associated with time1 with time2

GF_CycleTimeLevels (const int level) for a given level this cycles storage between all the time levels, e.g. $NextTime \rightarrow CurrentTime \rightarrow PreviousTime \rightarrow NextTime$.

8.4 Ghost Region Updates

In parallel processing the computational domain is subdivided into several sub-domains that are assigned to various processors. Each sub-domain maintains an overlapping zone at its boundaries with its neighboring sub-domains for synchronization of the data during the updating process. This overlapping zone is called the ghost region. Intra-grid communications consist of reading in ghost regions from neighboring processing elements and writing out to ghost regions. The following parameters are used in this section:

ident DAGH_Main or DAGH_Shadow

mgl is the multigrid level

dir is the direction along which to update the ghost region

- DAGH_Forward
- DAGH_Backward
- DAGH_Both

axis is the axis along which to update the ghost region

- DAGH_X
- DAGH_Y
- DAGH_Z
- DAGH_All

GF_WriteGhosts operator writes out ghost regions to neighboring processing elements.

GF_WriteGhosts (const int time, const int level)

GF_WriteGhosts (const int time, const int level, const int ident)

GF_WriteGhosts (const int time, const int level, const int mgl, const int ident)

GF_WriteGhosts (const int time, const int level, const int axis, const int dir, const int ident)

GF_WriteGhosts (const int time, const int level, const int mgl, const int axis, const int dir, const int ident)

GF_ReadGhosts operator reads in ghost regions from neighboring processors.

GF_ReadGhosts (const int time, const int level)

GF_ReadGhosts (const int time, const int level, const int ident)

GF_ReadGhosts (const int time, const int level, const int mgl, const int ident)

GF_ReadGhosts (const int time, const int level, const int axis, const int dir, const int ident)

GF_ReadGhosts (const int time, const int level, const int mgl, const int axis, const int dir, const int ident)

GF_Sync operator first writes out to ghost regions of neighboring processing elements and then reads in ghost regions from neighboring processors.

GF_Sync (const int time, const int level)

GF_Sync (const int time, const int level, const int ident)

GF_Sync (const int time, const int level, const int mgl, const int ident)

GF_Sync (const int time, const int level, const int axis, const int dir, const int ident)

GF_Sync (const int time, const int level, const int mgl, const int axis, const int dir, const int ident)

8.5 Operators on DAGH_Main, DAGH_Shadow, and Multigrids

8.5.1 Operators on DAGH_Main and DAGH_Shadow

These operators are defined for the Main and Shadow hierarchy. Hence the parameter *ident* takes on value *DAGH_Main* and *DAGH_Shadow*. GFType is a place holder for a template class.

The *GF_Fill* operator fills a Grid Function for a given time and level with a constant quantity *val*. *mgl* refers to the multigrid level.

GF_Fill (const GFType& val, const int time, const int level, const int ident)

GF_Fill (const GFType& val, const int time, const int level, const int mgl, const int ident)

The *GF_Copy* operator copies the contents of Grid Function *rhs* at time *time2* and level *level2* into Grid Function *gf* for time *time1* and *level1*. *where* is the bounding box in *gf* where the copy is to be done.

GF_Copy (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const BBox& where, const int ident)

GF_Copy (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const int ident)

The *GF_equals* operator assigns either a constant quantity *val* or the contents of the Grid Function *rhs* at time *time2* and *level2* to the Grid Function *gf* at its corresponding time and refinement level. *gf* would appear on the left hand side of the expression. *where* is the Bounding Box in *gf* where the assignment is made.

GF_equals (const int time, const int level, const BBox& where, const GFType& val, const int ident)

GF_equals (const int time, const int level, const GFType& val, const int ident)

GF_equals (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const BBox& where, const int ident)

GF_equals (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const int ident)

The *GF_plus* operator adds either a constant quantity *val* or the contents of the Grid Function *rhs* at time *time2* and *level2* to the Grid Function *gf* at its corresponding time and refinement level. *gf* would appear on the left hand side of the expression. *where* is the Bounding Box in *gf* where the addition is performed.

GF_plus (const int time, const int level, const BBox& where, const GFType& val, const int ident)

GF_plus (const int time, const int level, const GFType& val, const int ident)

GF_plus (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const BBox& where, const int ident)

GF_plus (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const int ident)

The *GF_minus* operator subtracts either a constant quantity *val* or the contents of the Grid Function *rhs* at time *time2* and *level2* from the Grid Function *gf* at its corresponding time and refinement level. *gf* would appear on the left hand side of the expression. *where* is the Bounding Box in *gf* where the subtraction is performed.

GF_minus (const int time, const int level, const BBox& where, const GFType& val, const int ident)

GF_minus (GridFunction& gf, const int time, const int level, const GFType& val, const int ident)

GF_minus (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const BBox& where, const int ident)

GF_minus (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const int ident)

The *GF_multiply* operator multiplies either by a constant quantity *val* or by the contents of the Grid Function *rhs* at time *time2* and *level2* the contents of Grid Function *gf* at its corresponding time and refinement level. *gf* would appear on the left hand side of the expression. *where* is the Bounding Box in *gf* where the multiplication is performed.

GF_multiply (const int time, const int level, const BBox& where, const GFType& val, const int ident)

GF_multiply (const int time, const int level, const GFType& val, const int ident)

GF_multiply (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const BBox& where, const int ident)

GF_multiply (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const int ident)

The *GF_divide* operator divides either by a constant quantity *val* or by the contents of the Grid Function *rhs* at time *time2* and *level2* the contents of Grid Function *gf* at its corresponding time and refinement level. *gf* would appear on the left hand side of the expression. *where* is the Bounding Box in *gf* where the division is performed.

GF_divide (const int time, const int level, const BBox& where, const GFType& val, const int ident)

GF_divide (const int time, const int level, const GFType& val, const int ident)

GF_divide (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const BBox& where, const int ident)

GF_divide (const int time1, const int level1, const GridFunction& rhs, const int time2, const int level2, const int ident)

8.5.2 Shadow Hierarchy Methods

The shadow hierarchy is a strict 2:1 coarsened copy of the main hierarchy, i.e. the both the space and time steps in the shadow hierarchy are twice that in the main hierarchy. The shadow hierarchy is mainly used to obtain the local truncation error. *GF_SetupShadow* creates a shadow grid for the Grid Function *gf*, whereas *GF_ReleaseShadow* returns the storage assigned for the shadow.

GF_SetupShadow (const int time, const int level)

GF_ReleaseShadow (const int time, const int level)

8.5.3 Multigrid Methods

In adaptive mesh refinement methods more than one grid hierarchy is utilized. The first two functions set up the storage for a multigrid system at given time and level. The parameters *mglc* and *mglf* refer to the coarse and fine multigrid levels respectively. The integer values coressponding to the x, y, and z axes are 0, 1, and 2 respectively. *GF_ReleaseMultiGrid* returns the storage assigned for the multigrid.

GF_SetupMultiGrid (const int time, const int level, const int mglc, const int mglf, const int ident)

GF_SetupMultiGrid (const int time, const int level, const int axis, const int mglc, const int mglf, const int ident)

GF_ReleaseMultiGrid (const int time, const int level, const int ident)

8.5.4 Reduction Operations on DAGH_Main, DAGH_Shadow, and Multigrid

The following reduction operators work the main, shadow, or multigrid hierarchies. The type returned is the type of the data associated with the Grid Function. *ident* takes on values DAGH_Main or DAGH_Shadow. *mgl* is the multigrid level. *where* is the Bounding Box in which the operator acts.
GF_maxval returns the maximum value of the data stored in the Grid Function.

GF_maxval (const int time, const int level, const int ident)

GF_maxval (const int time, const int level, const BBox& where, const int ident)

GF_maxval (const int time, const int level, const int mgl, const int ident)

GF_maxval (const int time, const int level, const int mgl, const BBox& where, const int ident)

GF_minval returns the minimum value of the data stored in the Grid Function.

GF_minval (const int time, const int level, const int ident)

GF_minval (const int time, const int level, const BBox& where, const int ident)

GF_minval (const int time, const int level, const int mgl, const int ident)

GF_minval (const int time, const int level, const int mgl, const BBox& where, const int ident)

GF_sum returns the sum of all the data values stored in the Grid Function.

GF_sum (const int time, const int level, const int ident)

GF_sum (const int time, const int level, const BBox& where, const int ident)

GF_sum (const int time, const int level, const int mgl, const int ident)

GF_sum (const int time, const int level, const int mgl, const BBox& where, const int ident)

GF_product returns the product of all the data values stored in the Grid Function.

GF_product (const int time, const int level, const int ident)

GF_product (const int time, const int level, const BBox& where, const int ident)

GF_product (const int time, const int level, const int mgl, const int ident)

GF_product (const int time, const int level, const int mgl, const BBox& where, const int ident)

GF_norm2 returns the euclidean norm of all the data values stored in the Grid Function.

GF_norm2 (const int time, const int level, const int ident)

GF_norm2 (const int time, const int level, const BBox& where, const int ident)

GF_norm2 (const int time, const int level, const int mgl, const int ident)

GF_norm2 (const int time, const int level, const int mgl, const BBox& where, const int ident)

8.6 General Queries

These general query operators return bounding boxes. *ident* can take on values DAGH_Main or DAGH_Shadow. *c* refers to the component grid. *mgl* is the multigrid level.

This query operator returns a bounding box but no data.

const BBox& databbox (const int time, const int level, const int c, const int ident)

BBox databbox (const int time, const int level, const int c, const int ident)

const BBox& databbox (const int time, const int level, const int c, const int mgl, const int ident)

BBox databbox (const int time, const int level, const int c, const int mgl, const int ident)

This query operator return a bounding box that is the interior region without the boundary.

const BBox& interiorbbox (const int time, const int level, const int c, const int ident)

BBox interiorbbox (const int time, const int level, const int c, const int ident)

const BBox& interiorbbox (const int time, const int level, const int c, const int mgl, const int ident)

BBox interiorbbox (const int time, const int level, const int c, const int mgl, const int ident)

This query operator returns a bounding box that is the boundary.

const BBox& bndrybbox (const int time, const int level, const int c, const int dir, const int ident)

BBox bndrybbox (const int time, const int level, const int c, const int dir, const int ident)

const BBox& bndrybbox (const int time, const int level, const int c, const int dir, const int mgl, const int ident)

BBox bndrybbox (const int time, const int level, const int c, const int dir, const int mgl, const int ident)

8.7 IO Support

There is an IO function for each GridFunction. This function is called when you do a GF_IO. It assumes that you will handle all aspects of IO including those introduced by parallelism. There is also DAGHIO support. This allocates IO nodes that will do collation and IO and is setup in multiple steps within the DAGH driver.

If you want to handle the IO routines yourself the following read and write functions can be used as templates. Note: *these functions will work only for a single grid*. The read function *rdata* will read data from file *input.data*. The format of the input data is x, y coordinates and function value on each line. The write function *wdata* will write to a file *output.data*. The format of the output data file is similar to the input data file.

```
void rdata (double* data, int* lb, int* ub, int*shape,
            double* bb) {
  double x, y, datain;
  int i, j, k;
```

```
  ifstream fin ("input.data");
  if (!fin) {
    cerr << "Cannot open input file input.data." << endl;
    exit (-1);
  }

  while (fin >> x >> y >> datain) {
    if ((x >= bb[0]) && (x <= bb[1]) &&
        (y >= bb[2]) && (y <= bb[3])) {
      i = ((x - bb[0]) * (ub[0] - lb[0]) / (bb[1] - bb[0]))
          + lb[0] + 1;
      j = ((y - bb[2]) * (ub[1] - lb[1]) / (bb[3] - bb[2]))
          + lb[1] + 1;
      k = (j - 1) * shape[0]  + i - 1;
      data[k] = datain;
      cout << x << "  " << y << "  " << datain << endl;
    }
  }
}

void wdata (double* data, int* lb, int* ub, int*shape,
            double* bb) {
  double x, y;
  int i, j, k, maxpts;

  ofstream fout ("output.data");
  if (!fout) {
    cerr << "Cannot open output file output.data." << endl;
    exit (-1);
  }

  maxpts = shape[0] * shape[1];

  for (k = 0; k < maxpts; k++) {
    i = (k % shape[0]) + 1;
    j = (k / shape[0]) + 1;
    x = ((i - 1 - lb[0]) * (bb[1] - bb[0]) / (ub[0] - lb[0]))
```

```
        - bb[0];
    y = ((j - 1 - lb[1]) * (bb[3] - bb[2]) / (ub[1] - lb[1]))
        - bb[2];
    fout << x << "  " << y << "  " << data[k] << endl;
    }
}
```

To call the read or write functions within the driver use the following syntax:

```
forall(u,Time,Level,c)
  rdata(FA(u(Time,Level,c,DAGH_Main)), bb);
end_forall

forall(u,Time,Level,c)
  wdata(FA(u(Time,Level,c,DAGH_Main)), bb);
end_forall
```

You can also read and write your data in *Hierarchical Data Format* or HDF. This is a multi-object file format for reading and writing scientific data in a distributed environment. HDF was developed at the National Center for Supercomputing Applications. You can get the HDF library and documentation from the web site *http:// hdf.ncsa.uiuc.edu.* DAGH is compatible with HDF version 4.x and not with version 5. Add *-DIO_NCSAIO* to your compile flags when you compile DAGH as well as your application. The following libraries will have to be loaded in this order: *-lmfhdf -ldf -ljpeg -lz.* You can add that list to your Makefile. Your application should also include the header file DAGHIO_hdf_ncsa.h.

The function *rhdf* will read data from a hdf formatted file *input.hdf* and initialize a grid function. The function *whdf* will write the contents of a grid function to a file *output.hdf.* The write function also has code to show how to write attributes to describe the data that is being written out. You can tailor that section according to the level of detail that is necessary to be stored. Both these functions are applicable for a single grid. The hdf format allows multi-grid data to be stored. Refer to the documentation on HDF to see how it is done.

```
void rhdf (double* data, int* lb, int* ub, int* shape){
```

```
  const int maxlen = 50;

// Open an existing HDF file "input.hdf" to read the
// attributes and initialize the Scientific Data (SD)
// interface.
  long int sd_id = SDstart ("input.hdf", DFACC_RDONLY);
  if (sd_id == FAIL) {
    cout << "SDstart failed." << endl;
    exit (-1);
  }

// Locate a scientific data set (SDS) by name "tport".
  long int retn = 0;
  long int sds_id = 0;      // ID of array
  char name[maxlen];        // name of array
  long int rank = 0;        // dimension of array
  long int dim[2];          // number of points along each axis
  long int num_type = 0;    // type of data
  long int attrs = 0;       // number of attributes

  int sds_index = SDnametoindex (sd_id, "tport");
  if (sds_index == FAIL) {
    cout << "SDnametoindex failed." << endl;
    exit (-1);
  }

  sds_id = SDselect (sd_id, sds_index);
  if (sds_id == FAIL) {
    cout << "SDselect failed." << endl;
    exit (-1);
  }

  retn = SDgetinfo (sds_id, name, &rank, dim, &num_type,
                    &attrs);
  if (retn == FAIL) {
    cout << "SDgetinfo failed." << endl;
    exit (-1);
```

```
  }

  if ((shape[0] != dim[0]) || (shape[1] != dim[1])) {
    cout << "Mismatch in grid and input dimensions." << endl;
    exit (-1);
  }

// Define the location, pattern, and size of the data to be read
  long int start[2] = {0, 0};  // location in the SDS where
                               // reading will begin
  long int edge[2] = {dim[0], dim[1]};  // number of data values to
                                        // be read in each dimension
  long int stride[2] = {1, 1};  // interval between read values
                                // in each dimension

// Read the array
  retn = SDreaddata (sds_id, start, stride, edge, (VOIDP) data);
  if (retn == FAIL) {
    cout << "SDreaddata failed." << endl;
    exit (-1);
  }

//  Terminate access to the array
  retn = SDendaccess (sds_id);
  if (retn == FAIL) {
    cout << "SDendaccess failed." << endl;
    exit (-1);
  }

// Terminate access to the SD interface and close input file
  retn = SDend (sd_id);
  if (retn == FAIL) {
    cout << "SDend failed." << endl;
    exit (-1);
  }
}
```

```
void whdf (double* data, int* lb, int* ub, int* shape){

  const int rank = 2;    // number of dimensions
  long int dim[2] = {shape[1], shape[0]};

// Create and open a new HDF file "output.hdf".
// Initialize and open the Scientific Data (SD) interface.
  char output_file[] = "output.hdf";
  long int sd_id = SDstart (output_file, DFACC_CREATE);
  if (sd_id == FAIL) {
    cout << "SDstart failed." << endl;
    exit (-1);
  }

// Define the characteristics of the new HDF data set
// The name of the array is "tport" and sds_id is the array id.
  char array_name[] = "tport";
  long int sds_id = SDcreate (sd_id, array_name, DFNT_DOUBLE,
                              rank, dim);
  if (sds_id == FAIL) {
    cout << "SDcreate failed." << endl;
    exit (-1);
  }

// Write existing data set to new data array in HDF file
  long int retn = 0;
  long int start[2] = {0, 0};  // location in the SDS where
                               // writing begins
  long int edge[2] = {shape[1], shape[0]};
                      // number of data values to be written
  long int stride[2] = {1, 1};  // interval between written
                                // values in each dimension

  retn = SDwritedata (sds_id, start, stride, edge, (VOIDP) data);
  if (retn == FAIL) {
    cout << "SDwritedata failed." << endl;
    exit (-1);
```

```
  }

// Set an attribute that describes the file contents
  char contents[] = "Gaussian Data";
  retn = SDsetattr (sd_id, "file_contents", DFNT_CHAR8,
                    strlen (contents), (VOIDP) contents);
  if (retn == FAIL) {
    cout << "SDsetattr failed for file ID." << endl;
    exit (-1);
  }

// Set an attribute that describes the format of the array
  char fmt[] = "f14.7";
  retn = SDsetattr (sds_id, "format", DFNT_DOUBLE,
                    strlen(fmt), (VOIDP) fmt);
  if (retn == FAIL) {
    cout << "SDsetattr failed for array ID" << endl;
    exit (-1);
  }

// Set an attribute that specifies the dimension metric
  char* dimet[] = {"meters", "meters"};
  long int dim_id = 0;     // identifier for any dimension
  for (int i = 0; i < 2; i++) {
//  Get dimension id
    dim_id = SDgetdimid (sds_id, i);
    if (dim_id == FAIL) {
      cout << "SDgetdimid failed for dimension " << i << endl;
      exit (-1);
    }
//  Set the attribute that specifies the dimension metric
    retn = SDsetattr (dim_id, "dim_metric", DFNT_CHAR8,
                      strlen (dimet[i]), (VOIDP) dimet[i]);
    if (retn == FAIL) {
      cout << "SDsetattr failed for dimension " << i << endl;
      exit (-1);
    }
```

```
  }
// Terminate access to the array
  retn = SDendaccess (sds_id);
  if (retn == FAIL) {
    cout << "SDendaccess failed." << endl;
    exit (-1);
  }

// Terminate access to the SD interface and close the file
  retn = SDend (sd_id);
  if (retn == FAIL) {
    cout << "SDend failed." << endl;
    exit (-1);
  }
}
```

To call the hdf read or write functions within the driver use the following syntax:

```
forall(u,Time,Level,c)
  rhdf(FA(u(Time,Level,c,DAGH_Main)));
end_forall

forall(u,Time,Level,c)
  whdf(FA(u(Time,Level,c,DAGH_Main)));
end_forall
```

Once the output is written to a hdf formatted file sometimes it is necessary to examine the attributes of the data that has been stored without actually examining the data values themselves. The following routine shows how you can extract that information in several ways. For a more complete discussion on this, refer to the HDF documentation.

```
//  This program opens an HDF file and reads the various attributes

#include <iostream.h>
```

```
#include "hdf.h"
#include "mfhdf.h"

int main (int argc, char* argv[]) {

  const int maxlen = 50;
  int i = 0;
  int j = 0;

//  Open an existing HDF file to read the attributes and initialize
//  the SD (Scientific Data) interface.
  char hdf_file[] = "output.hdf";
  long int sd_id = SDstart (hdf_file, DFACC_READ);   // File id
  if (sd_id == FAIL) {
    cout << "SDstart failed" << endl;
    exit (-1);
  }

//  Determine the contents of the HDF file
  long int retn = 0;
  long int n_datasets = 0;     // number of datasets within the file
  long int n_file_attr = 0;    // number of file attributes
  retn = SDfileinfo (sd_id, &n_datasets, &n_file_attr);
  if (retn == FAIL) {
    cout << "SDfileinfo failed" << endl;
    exit (-1);
  }
  else {
    cout << "Num of data sets = " << n_datasets << endl;
    cout << "Num of file attributes = " << n_file_attr << endl;
  }

//  Access and print the name of every data set in the file
  long int sds_id = 0;     // ID of array
  char name[maxlen];       // name of array
  long int rank = 0;       // dimension of array
  long int dim[3];         // number of points along each axis
```

```
  long int num_type = 0;   // type of data
  long int attrs = 0;      // number of attributes

  for (i = 0; i < n_datasets; i++) {
    sds_id = SDselect (sd_id, i);
    if (sds_id == FAIL) {
      cout << "SDselect failed" << endl;
      exit (-1);
    }
    else {
      retn = SDgetinfo (sds_id, name, &rank, dim,
                        &num_type, &attrs);
      if (retn == FAIL) {
        cout << "SDgetinfo failed" << endl;
        exit (-1);
      }
      else {
        cout << "name = " << name << endl;
        cout << "rank = " << rank << endl;
        cout << "dim: ";
        for (j = 0; j < rank; j++) {
          cout << dim[j] << "  ";
        }
        cout << endl;
        cout << "type = " << num_type << endl;
        cout << "attr = " << attrs << endl;
      }
    }

//  Terminate access to the array
    retn = SDendaccess (sds_id);
    if (retn == FAIL) {
      cout << "SDendaccess failed" << endl;
      exit (-1);
    }
  }
```

```
//  You can also locate a scientific data set (SDS)
//  if you know its name.
  char arrn[] = "tport";
  int sds_index = SDnametoindex (sd_id, arrn);
  if (sds_index == FAIL) {
    cout << "SDnametoindex failed" << endl;
    exit (-1);
  }
  sds_id = SDselect (sd_id, sds_index);
  if (sds_id == FAIL) {
    cout << "SDselect failed" << endl ;
    exit (-1);
  }

  retn = SDgetinfo (sds_id, name, &rank, dim, &num_type, &attrs);
  if (retn == FAIL) {
    cout << "SDgetinfo failed" << endl;
    exit (-1);
  }

  cout << "name = " << name << endl;
  cout << "rank = " << rank << endl;
  cout << "dim: ";
  for (j = 0; j < rank; j++) {
    cout << dim[j] << "  ";
  }
  cout << endl;
  cout << "type = " << num_type << endl;
  cout << "attr = " << attrs << endl;

//  Terminate access to the array
    retn = SDendaccess (sds_id);
    if (retn == FAIL) {
      cout << "SDendaccess failed" << endl;
      exit (-1);
    }
```

```
//  Get information about the file attribute
  long int attr_index = 0;
  long int count = 0;
  char attr_name[maxlen] = "\0";
  char buffer[maxlen] = "\0";

  retn = SDattrinfo (sd_id, attr_index, attr_name,
                     &num_type, &count);
  if (retn == FAIL) {
    cout << "SDattrinfo failed" << endl;
    exit (-1);
  }

//  Read the attribute data
  retn = SDreadattr (sd_id, attr_index, buffer);
  if (retn == FAIL) {
    cout << "SDreadattr failed" << endl;
    exit (-1);
  }
  else {
    cout << attr_name << ": ";
    for (i = 0; i < count; i++)
      cout << buffer[i];
    cout << endl;
  }

//  Get the identifier for the first data set
  sds_index = SDnametoindex (sd_id, arrn);
  if (sds_index == FAIL) {
    cout << "SDnametoindex failed" << endl;
    exit (-1);
   }
  sds_id = SDselect (sd_id, sds_index);
  if (sds_id == FAIL) {
     cout << "SDselect failed" << endl;
     exit (-1);
```

```
  }

//  Find the data set attribute named "format"
  attr_index = SDfindattr (sds_id, "format");
  if (attr_index == FAIL) {
    printf ("SDfindattr failed.  \n\n");
    exit (-1);
  }

//  Read the attribute data
  for (j = 0; j < maxlen; j++) buffer[j] = '\0';
  retn = SDreadattr(sds_id, attr_index, buffer);
  if (retn == FAIL) {
    cout << "SDreadattr failed" << endl;
    exit (-1);
  }
  else
    cout << "format: " << buffer << endl;

// Find the data set attribute named "dim_metric"
  long int dim_id = 0;
  for (i = 0; i < rank; i++) {
    dim_id = SDgetdimid (sds_id, i);
    if (dim_id == FAIL) {
      cout << "SDgetdimid failed for dimension "  << i << endl;
      exit (-1);
    }

    attr_index = SDfindattr (dim_id, "dim_metric");
    if (attr_index == FAIL) {
      cout << "SDreadattr failed" << endl;
      exit (-1);
    }

    for (j = 0; j < maxlen; j++) buffer[j] = '\0';
    retn = SDreadattr (dim_id, attr_index, buffer);
    if (retn == FAIL) {
```

```
      cout << "SDreadattr failed" << endl;
      exit (-1);
    }
    else
      cout << "dim_metric: " << buffer << endl;
  }

//  Terminate access to the array
  retn = SDendaccess (sds_id);
  if (retn == FAIL) {
    cout << "SDendaccess failed" << endl;
    exit (-1);
  }

//  Terminate access to the SD interface and close the file
  retn = SDend (sd_id);
  if (retn == FAIL) {
    cout << "SDend failed" << endl;
    exit (-1);
  }

  return 0;
}
```

A simpler and more flexible format for storing mulitdimensional scientific data is FlexIO. It borrows its storage mechanism from HDF and NetCDF. There is a higher level API that utilizes FlexIO to provide simplified access to Adaptive Mesh Refinement data structures. For documentation and information on how to download the FlexIO libraries refer to the web site *http://bach.ncsa.uiuc.edu/FlexIO*.

You need to compile DAGH and your application with the flag *-DIO_IEEEIO*. The following libraries need to be loaded in addition to the hdf libraries already mentioned: *-lhdfio -lieeeio*. Include the header file *DAGHIO_ieeeio_ncsa.h* in your application. FlexIO files have the suffix ".raw". To exemplify the simplicity of the FlexIO API I have included two functions *flexread* and *flexwrite* that read and write FlexIO files.

```
void flexread (double* data, int* lb, int* ub, int* shape) {
```

```
  int rank;
  int dims[2];
  IObase::DataType numbertype;

// Open file containing formatted data
  char input_file[] = "input.raw";
  IObase *reader = new IEEEIO (input_file, IObase::Read);
  if (!reader->isValid()) {
    cout << "Cannot open " << input_file << endl;
    exit (-1);
  }

// Get info about the data set
  reader->readInfo (numbertype, rank, dims);
  if ((rank != 2) || (dims[0] != shape[0])
     || (dims[1] != shape[1])) {
     cout << "Mismatch in coordinate dimensions" << endl;
     exit (-1);
  }

// Read data into the array
  reader->read(data);

// Close input file
  delete reader;
}

void flexwrite (double* data, int* lb, int* ub, int* shape){
  int rank = 2;
  int dim[2] = {shape[0], shape[1]};

//  Create the output file
  char output_file[] = "output.raw";
  IObase *writer = new IEEEIO (output_file, IObase::Create);

//  Write the dataset
  writer->write (IObase::Float64, rank, dim, data);
```

```
//  Close the output file
  delete writer;
}
```

These routines can be called in the usual way within the DAGH driver.

```
forall(u,Time,Level,c)
  flexread(FA(u(Time,Level,c,DAGH_Main)));
end_forall

forall(u,Time,Level,c)
  flexwrite(FA(u(Time,Level,c,DAGH_Main)));
end_forall
```

Another file format is available - RNPL. Rapid Numerical Prototyping Language (RNPL) was developed by Matthew Choptuick and Robert Marsa. You can obtain the source code and documentation from their ftp site *ftp://dirac.ph.utexas.edu/pub/rnpl*. If you are going to use the RNPL format compile DAGH and your application with the flag *-DIO_RNPLIO*. Add *-lrnpl* to the list of libraries to be loaded.

We have three DAGH IO types: DAGHIO_HDF_NCSA, DAGHIO_IEEEIO_NCSA, and DAGHIO_HDF_RNPL. The built-in IO routines of DAGH will take care of single and multiple grids. If you choose to use the built-in IO routines, these are the steps you have to take:

- Initialize the DAGHIO (usually just after MPI_Init)
  DAGHIOInit();
- Set DAGHIO type with any *one* of the three IO types defined above before the ComposeHierarchy statement. For example:
  DAGHIOType (GH, DAGHIO_HDF_NCSA); or,
  DAGHIOType (GH, DAGHIO_IEEEIO_NCSA); or,
  DAGHIOType (GH, DAGHIO_HDF_RNPL)
- Use BEGIN_COMPUTE and END_COMPUTE to mark off regions executed by the compute nodes. This includes declaration of the Grid Functions.

- Read from or write to a Grid Function using the *Read* and *Write* functions whose syntax has been defined below.
- Shut down IO server just before END_COMPUTE
  DAGHIOEnd (GH);
- Close files for hdf and flexio formats only after END_COMPUTE
  DAGHIO_HDF_NCSA_Finalize(); or,
  DAGHIO_IEEEIO_NCSA_Finalize;

This code sample shows how to use the built-in DAGHIO. The grid function that we are writing out is "u". The output that will be produced is *u.hdf*.

```
// Set IO Flags
#define VizServer 0     // Which processor will write out
#define DAGH_IO 1       // We are using the DAGHIO server


... ... ...

int main (int argc, char* argv[]) {

#ifndef DAGH_NO_MPI
  MPI_Init (&argc, &argv);
#endif

#ifdef DAGH_IO
  DAGHIOInit();
#endif

// Set up Grid Hierarchy GH

  ... ... ...

#ifdef DAGH_IO
  DAGHIOType (GH, DAGHIO_HDF_NCSA);
#endif

  ComposeHierarchy (GH);
```

```
// Processor information
  int me = MY_PROC (GH);  // processor number
  ... ... ...

BEGIN_COMPUTE

//  Set up Grid Function u and initialize

  ... ... ...

#ifdef DAGH_IO
  for (int curlev = 0; curlev <= GH.maxlevelindex(); curlev++) {
    if (me == VizServer)
      cout << "DAGHIO: Dumping Level: " << curlev << endl;
    const int saveTime = GH.getCurrentTime (curlev, DAGH_Main);
    GH.setCurrentTime(GH.getCurrentTime(curlev,DAGH_Main)/
      GH.timestep(0,DAGH_Main), curlev, DAGH_Main);
    u.GF_Write(GH.getCurrentTime(curlev,DAGH_Main),curlev,DAGH_Double);
    GH.setCurrentTime (saveTime, curlev, DAGH_Main);
  }
#endif

#ifdef DAGH_IO
  DAGHIOEnd (GH);
#endif

END_COMPUTE

#ifdef DAGH_IO
  DAGHIO_HDF_NCSA_Finalize();
#endif

#ifndef DAGH_NO_MPI
  DAGHMPI_Finalize();
#endif
```

```
  return (0);
}  // End main
```

If the grid function that you are writing out is ”u” the output file will be *u.hdf*, *u.raw*, or *u.sdf*, depending on whether you have chosen hdf, flexio, or rnpl format. At present the GF_Read functions have been declared but not defined.

GF_Write (const int time, const int level, const int mgl, const int ident, const int gfdtype)

GF_Write (const int time, const int level, const int gfdtype)

GF_Write (const int time, const int level, const int mgl, const int gfdtype)

GF_Write (const int time, const int level, const int mgl, const int ident, const BBox& where, const int gfdtype)

GF_Write (const int time, const int level, const BBox& where, const int gfdtype)

GF_Write (const int time, const int level, const int mgl, const BBox& where, const int gfdtype)

GF_Read (const int time, const int level, const int mgl, const int ident, const int gfdtype)

GF_Read (const int time, const int level, const int gfdtype)

GF_Read (const int time, const int level, const int mgl, const int gfdtype)

GF_Read (const int time, const int level, const int mgl, const int ident, const BBox& where, const int gfdtype)

GF_Read (const int time, const int level, const BBox& where, const int gfdtype)

GF_Read (const int time, const int level, const int mgl, const BBox& where, const int gfdtype)

9 Miscellaneous Topics

9.1 Unstructured Cell Data

9.1.1 Generate UCD Information

DAGH can generate Unstructured Cell Data (UCD) information for the current Grid Hierarchy. To create a DAGHUCD do

DAGHUCD(DIM) ucd (const int minlevel, const int maxlevel, GridHierarchy& gh, const int ident)

where DIM = 1, 2, or 3, and minlevel / maxlevel define the range of levels for which the UCD information is required for the the Grid Hierarchy *gh*. *ident* takes on values DAGH_Main or DAGH_Shadow.

9.1.2 Extract UCD Geometry Information

To extract the UCD geometry information from the DAGHUCD object do

ucd.GetUCDInfo (int& numnodes, double**& nodes, int& numcells, int**& cells, const int coordtype)

This will return:

numnodes number of unique nodes in the composite grid structure from minlevel to maxlevel

nodes [DIM][numnodes] array of coordinates for each node

numcells number of unique cells in the composite grid structure from minlevel to maxlevel

cells [numcells][nodes_per_cell] array of unique node numbers per cell. The number of nodes per cell depends on the type of cell and is fixed as 2 for 1-D, 4 for 2-D, and 8 for 3-D

The input parameter *coordtype* can be DAGHLocalCoords or DAGHWorldCoords depending on whether DAGH internal coordinates or application defined world coordinates are required.

9.1.3 Associated Data for each Node

To get the associated data for each node froma Grid Function *gf* do

GetUCDData (GridFunction& gf, const DAGHUCD& ucd, const int time, $GFType*$& ucddata, const int ucdserver)

where

ucd the ucd object instantiated above

time the time at which ucd data is required

ucddata output array of size numnodes that contains the Grid Function value at each node

ucdserver number of the processor on which this information is to be collected

9.2 User Defined Flags

Each DAGH Grid Function has a set of flags (currently 10) of type short that can be set or queried by the user. To set the n^{th} flag to *flag* do

SetUserFlag (GridFunction& gf, const int n, const short& flag)

To return the value of the n^{th} flag do

short GetUserFlag (GridFunction& gf, const int n)

9.3 Masked Interpolation

A new feature has been added to DAGH's interpolation operators. DAGH can now handle masked interpolation. The mask is a grid function. It provides a flag for each grid point to determine whether that grid location contains data or not. Two new types have been defined *MaskedProlongFunc* and *MaskedRestrictFunc* to handle masked interpolation.

```
/*  Masked Prolongation Function Interface  */
typedef void (*MaskedProlongFunc) (
    FORTRAN_INTERFACE (GF_TYPE), FORTRAN_INTERFACE (GF_TYPE),
    FORTRAN_INTERFACE (GF_TYPE), FORTRAN_INTERFACE (GF_TYPE),
    BBOX_INTERFACE,
    GF_TYPE *, const int *)

/*  Masked Restriction Function Interface  */
typedef void (*MaskedRestrictFunc) (
    FORTRAN_INTERFACE (GF_TYPE), FORTRAN_INTERFACE (GF_TYPE),
    FORTRAN_INTERFACE (GF_TYPE), FORTRAN_INTERFACE (GF_TYPE),
    BBOX_INTERFACE,
    GF_TYPE *, const int *)
```

The second and fourth FORTRAN_INTERFACEs refer to the *source* and *destination* masks. To set the masked prolongation and restriction functions use:

GF_SetMaskedProlongFunc ((void *) &mpf, (void *) &pm) where *mpf* is the masked prolongation interpolation routine and *pm* is the grid function designated as the prolongation mask.

GF_SetMaskedRestrictFunc ((void *) &mrf, (void *) &rm) where *mrf* is the masked restriction routine and *rm* is the grid function designated as the restriction mask.

To check if a grid function has both prolongation routine and prolongation mask assigned or it has both restriction routine and restriction mask assigned use the following operators:
int **has_maskedprolongfunc**() or,
int **has_maskedrestrictfunc**().

The default value for the masked prolong flag and masked restrict flag is DAGHFalse. If you have both the prolong flag and the masked prolong flag set to DAGHTrue only the normal prolongation will work and **not** the masked prolongation. If you want only the masked prolongation routine to apply make sure that the prolong flag is set to DAGHFalse and the masked prolong flag is set to DAGHTrue. Use the following functions to set the masked flags:

GF_SetMaskedProlongFlag (const int flag)

GF_SetMaskedRestrictFlag (const int flag)

To check whether the flags have been set or not do:
int **maskedprolong**() or,
int **maskedrestrict**().

Once the masked prolongation and masked restriction functions have been assigned, you could use the following operators to apply the specified functions at a particular time and level for the whole grid or for a part of the grid as specified by the bounding box *bb*. The parameter *args* is an array of dimension *cnt*. It contains arguments to be passed to the masked prolongation or masked restriction functions. The parameter *ident* can take on values DAGH_Main or DAGH_Shadow.

GF_MaskedProlong (const int time_from, const int level_from, const int time_to, const int level_to, GFType *args, const int cnt, ident)

GF_MaskedProlong (const int time_from, const int level_from, const int time_to, const level_to, const BBox &bb, GFType *args, const int cnt, ident)

GF_MaskedRestrict (const int time_from, const int level_from, const int time_to, const int level_to, GFType *args, const int cnt, ident)

GF_MaskedRestrict (const int time_from, const int level_from, const int time_to, const level_to, const BBox &bb, GFType *args, const int cnt, ident)

If the grid function has a shadow, then the masked prolongation will also apply to the shadow. You must make sure then that the mask also has a shadow. If a grid function has a masked prolongation function assigned to it, then the masked prolongation will be applied to the boundary (except in the case for adaptive boundaries). For distributed computing, the masked prolongation function will apply to the ghost regions. The order in which the ordinary prolongation and masked prolongation is to be performed is left to the user. If you want to prolong those grid functions that do not have masks before those that do, use the following construct:

```
foreachGF (GH, GF, DIM, GFTYPE)
  if (GF.has_prolongfunc() == DAGHTrue)
    GF.GF_Prolong (time_from, level_from, time_to, level_to,
                   args, cnt, DAGH_Main);
end_foreachGF

foreachGF (GH, GF, DIM, GFTYPE)
  if (GF.has_maskedprolongfunc() == DAGHTrue)
    GF.GF_MaskedProlong (time_from, level_from, time_to, level_to,
                         args, cnt, DAGH_Main);
end_foreachGF
```

9.4 Visualization Support

9.4.1 DAGH Interface

There are several ways that one can visualize the results of a computation. DAGH provides a function that allows the user to interface with a visualization program like *xgraph*.

View (GridFunction& gf, const int time, const int level, const int axis, const BBox& where, const int pserver, const int type, const char* xg, const char* dserver, const int ident)

axis is the axis along which the grid is viewed. It could be

- DAGH_X
- DAGH_Y
- DAGH_Z

where is the bounding box that is to be displayed

pserver from which processor we are getting the data to do the visualization

type is one of DAGHViz_XGRAPH or DAGHViz_FILE

xg pointer to the executable file that does the visualization

dserver processor where the visualization is displayed

ident DAGH_Main or DAGH_Shadow

9.4.2 Using Files and AVS

One of the easiest ways to visualize your output is to write the results to a file and then write a module in AVS/Express to read the data from the file for visualization. To find out how you can output the results of your computation in DAGH to a file refer to the section on Basic IO in this manual.

If you output your results in HDF format there are public domain AVS modules that can read HDF formatted files. These modules can be downloaded from the web site (www.iavsc.org) of the International AVS Centre. If you have AVS5, go to /repository/avs5 and download read_hdf_image. If you have AVS/Express, go to /repository/express and download ReadHDFImage.

Suppose you want to visualize the values of a Grid Function **u** bound to a 2-D grid hierarchy. Let us say that the output file (**output.data**) that DAGH produces has the following format:

```
1089
0.6875000  0.0312500  0.0067459
0.7187500  0.0312500  0.0745211
...  ...  ...
```

The first number (1089) is the number of data lines in the file. All subsequent numbers are actual data points in the grid. Each data line has three components - the **x** and **y** coordinates of each grid point and the value of the grid function at that point.

To create a module in AVS/Express that will read the output.data file refer to your AVS/Express User's Guide. An outline of the steps that you will perform is given below:

- Create the module **DataReader** to read the datafile.
- Generate the C code **datareader.c** that actually opens the file and reads the data.
- Compile the DataReader's process in the Library Workspace.
- Build a user interface **DataUI** to read files.
- Build a module **DataField** that will create a renderable AVS field.

- Assemble the DataUI, DataReader, DataField, and a renderer like Uviewer2D.

This module can then be used to read the output.data file and view the data. Given below is an example of a process (datareader.c) that will read the data file and the v code (**datareader.v**) for the modules DataUI, DataReader, and DataField. You can use this as a template to create your own module. The following steps are to be taken to get this module running under AVS.

- Create a project directory **daghviz**.
- Create a subdirectory **datareader** and move datareader.c in it.
- Start AVS and select ScratchPad.
- From the **Object** menu bar load object datareader.v in the work area. Note the connections between the modules.
- Place the modules in one of the **Library Workspaces**.
- From the **Project** menu bar save the project by giving the full path name of the directory daghviz as the project directory.
- Highlight the module DataReader in the Library Workspace and from the Project menu bar compile the the process datareader.c and save the project after compilation.
- From the **File** menu bar select **New Application**.
- Select **SingleWindowApp** with **Uviewer2D**.
- From the Library Workspace pull down the modules DataUI, DataReader, and DataField.
- Connect the modules as in the original configuration. Connect the red output port of DataField to the red input port of Uviewer2D.
- The module DataUI will create a small window interface with a single pull down menu bar File. To view the data use the File menu bar to enter the full path name of the output.data file.

This is the code for **datareader.c**.

```
#include "user.h"
#include <stdio.h>

int
data_reader(OMobj_id DataReader_id, OMevent_mask event_mask,
    int seq_num)
{
   /***********************/
   /*  Declare variables  */
   /***********************/
   char  *filename = NULL;
   int  num_values;
   int  xy_coordinates_size = 0;
   float *xy_coordinates = NULL;
   int  function_values_size = 0;
   float *function_values = NULL;

   int i;
   FILE *fp;

   /***********************/
   /*  Get input values   */
   /***********************/
   /* Get filename's value */
   if (OMget_name_str_val(DataReader_id,
       OMstr_to_name("filename"), &filename, 0) != 1)
      filename = NULL;


   /***********************/
   /* Function's Body     */
   /***********************/

   /* Open the data file  */
   fp = fopen (filename, "r");
```

```
if (fp == NULL) {
  fprintf (stderr, "Cannot open file %s. \n", filename);
  return (0);
}

 /* Get number of data points  */
 fscanf (fp, "%d", &num_values);


/**********************/
/*  Set output values  */
/**********************/

/* Set num_values's value */
OMset_name_int_val(DataReader_id,
   OMstr_to_name("num_values"), num_values);

/* Set a pointer to the xy_coordinates array  */
xy_coordinates = (float *)OMret_name_array_ptr(DataReader_id,
                  OMstr_to_name("xy_coordinates"),
  OM_GET_ARRAY_RW, &xy_coordinates_size, NULL);


/* Set a pointer to the function values array  */
function_values = (float *)OMret_name_array_ptr(DataReader_id,
                   OMstr_to_name("function_values"),
   OM_GET_ARRAY_RW, &function_values_size, NULL);


/* Read the data into arrays  */
for (i = 0; i < num_values; i++)
  fscanf (fp, "%f %f %f", xy_coordinates+2*i,
  xy_coordinates+2*i+1, function_values+i);


/***********************/
/*  Free input variables */
```

```
   /************************/
   if (filename) free(filename);
   if (xy_coordinates != NULL) ARRfree ((char *) xy_coordinates);
   if (function_values != NULL) ARRfree ((char *) function_values);

   return(1);
}
```

This is the v code **datareader.v** for the modules DataUI, DataReader, and DataField.

```
macro DataUI<NEx=242.,NEy=77.,NEscalingFactor=1.,
             NExOffset=308.,NEyOffset=58.> {
   UIapp UIapp<NEx=198.,NEy=187.> {
      title => "DataReader";
      menu => <-.UIcmdList;
      x = 23;
      y = 529;
   };
   UIcmdList UIcmdList<NEx=187.,NEy=110.> {
      cmdList => {<-.UIcmdListFile};
   };
   UIcmdList UIcmdListFile<NEx=187.,NEy=22.> {
      cmdList => {<-.UIcmdOpen,
         <-.UImenuSeparator,<-.UIcmdExit};
      label = "File";
   };
   UIcmd UIcmdOpen<NEx=-11.,NEy=22.> {
      label => "Open...";
   };
   UImenuSeparator UImenuSeparator<NEx=-11.,NEy=110.>;
   UIcmd UIcmdExit<NEx=-11.,NEy=187.> {
      label => "Exit...";
   };
   UIfileDialog UIfileDialog<NEx=-187.,NEy=22.> {
      visible => <-.UIcmdOpen.do;
      title => "Select a Data File";
      filename<NEportLevels={0,3}>;
```

```
      x = 490;
      y = 267;
      width = 300;
      height = 390;
      cancel = 1;
   };
   UImessageDialog UImessageDialog<NEx=-11.,NEy=253.> {
      visible => <-.UIcmdExit.do;
      title => "Exit?";
      message = "Are you sure you want to exit?";
      x = 534;
      y = 409;
      width = 212;
      height = 106;
      ok<NEportLevels={1,2}> = 0;
      cancel = 1;
   };
   GMOD.exit_process exit_process<NEx=-11.,NEy=330.> {
      do_exit => <-.UImessageDialog.ok;
   };
};


module DataReader<NEx=198.,NEy=165.,NEwidth=264.,NEheight=198.,
                  process="user",src_file="datareader.c",
  build_dir="datareader"> {
   omethod+notify_inst+req update = "data_reader";
   string+read+notify+req filename<NEportLevels={2,1},
   export=2> => <-.DataUI.UIfileDialog.filename;
   int+write num_values<NEportLevels=1,export=2>;
   float+write xy_coordinates<NEportLevels={1,2},export=2>
   [.num_values][2];
   float+write function_values<NEportLevels={1,2},export=2>
   [.num_values];
};
```

```
macro DataField<NEx=209.,NEy=275.,NEscalingFactor=1.,
                NExOffset=250.,NEyOffset=72.> {
   FLD_MAP.point_mesh point_mesh<NEx=22.,NEy=11.> {
      coord<NEportLevels={3,0}> => <-.<-.DataReader.xy_coordinates;
   };
   FLD_MAP.node_scalar node_scalar<NEx=-66.,NEy=77.> {
      in_data<NEportLevels={3,0}> => <-.<-.DataReader.function_values;
   };
   FLD_MAP.combine_mesh_data combine_mesh_data<NEx=110.,NEy=132.> {
      in_mesh => <-.point_mesh.out;
      in_nd => <-.node_scalar.out;
   };
   MODS.scat_to_tri scat_to_tri<NEx=44.,NEy=220.> {
      in_field => <-.combine_mesh_data.out;
      Scat2TriUI {
         UIpanel {
            option {
               set = 1;
            };
         };
      };
      out_obj<NEportLevels={1,3}>;
   };
};
```

9.4.3 Using Sockets and AVS

Another way to visualize your output using AVS/Express is to establish a socket connection. You will have to write a server module for AVS that will receive your data and display it and add a client routine within your DAGH application to send the data to the server. To find out how you can write a client routine within your DAGH application refer to the section on Basic IO in this manual.

Let us assume that you want to visualize the values of a Grid Function **u** that is bound to a 2-dimensional grid hierarchy. The client routine within the DAGH application streams data in the following manner. The first line of data it sends contains an integer specifying the number of lines of data being sent. Each subsequent line of data contains three floats or three doubles. The first two real numbers represent the **x** and **y** coordinates of the grid point and the last real number is the function value at that point.

The AVS/Express User's Guide has an outline of how to create a general purpose module. That outline has been modified to create a module that will act as our server.

- Create the module **DataSoc** to act as our server.
- Generate the C code **datasoc.c** that actually establishes a "listening" socket.
- Compile the DataSoc's process in the Library Workspace.
- Build a module **DataField** that will create a renderable AVS field.
- Assemble the DataSoc, DataField, and a renderer like Uviewer2D.

This module can then be used as our server module. Given below is an example of a process (datasoc.c) that will establish the socket and the v code (datasoc.v and datafield.v) for the modules DataSoc and DataField. You can use this as a template to create your own module. Neither the module (v code) nor the process (c code) is very elegant. But it works! If you have a more elegant solution, please send it to me and I will put it in our archives. The following steps are to be taken to get this module running under AVS.

- Create a project directory **daghsoc**.
- Create a subdirectory **socser** and move socser.c in it.
- Start AVS and select ScratchPad.
- From the **Object** menu bar load the objects datasoc.v and datafield.v in the work area.
- Place the modules in one of the Library Workspaces.
- From the **Project** menu bar save the project by giving the full path name of the directory daghsoc as the project directory.

- Highlight the module DataSoc in the Library Workspace and from the Project menu bar compile the process datasoc.c and save the project after compilation.
- From the **File** menu bar select **New Application**.
- Select **SingleWindowApp** with **Uviewer2D**.
- From the Library Workspace instance (pull down) modules DataSoc and DataField.
- Connect the output port of the **xy_coords** in DataSoc to the input port **point_mesh** in the module DataField. Connect the **function_values** output port in DataSoc to the **node_scalar** input port in the module DataField. Connect the red output port of DataField to the red input port of Uviewer2D.
- Open the module DataSoc and note the port number. Close the module.
- In another window go to the directory of the DAGH application. Edit the header file **soc.h**. Change the **SERV_TCP_PORT** number to reflect the current port number that you noted. Change SERV_HOST_ADDR to reflect the IP address of the host machine that you are running AVS on. This could be different from the machine you are running the DAGH application. Recompile the DAGH application. You may have to do a forced rebuild because only the header file was changed.

This is the code for **datasoc.c**.

```
#include "user.h"
#include "soc.h"
#include <avs/event.h>


typedef struct
{
  OMobj_id module_id;
  int lsd;
```

```
  int sd;
} datasoc_internal_t;

void
datasoc_read_cb (datasoc_internal_t *internal)
{
  char recvbuf[MAXSIZE], sendbuf[MAXSIZE];
  int i, msglength, num_values;
  float x, y, z;
  int xy_coords_size = 0;
  float *xy_coords = NULL;
  int function_values_size = 0;
  float *function_values = NULL;

  if (recv (internal->sd, recvbuf, sizeof(recvbuf), 0) < 0)
    return;

  sscanf (recvbuf, "%d", &num_values);
  OMset_name_int_val (internal->module_id,
      OMstr_to_name ("num_values"),
      num_values);

  msglength = strlen (recvbuf);
  sprintf (sendbuf,"%d", msglength);
  msglength = strlen (sendbuf);
  sendbuf[msglength] = '\0';
  if (send (internal->sd, sendbuf, sizeof(sendbuf), 0) < 0)
    return;

  /*  Set a pointer to the xy_coords array  */
  xy_coords = (float *)OMret_name_array_ptr(internal->module_id,
       OMstr_to_name("xy_coords"), OM_GET_ARRAY_RW,
       &xy_coords_size, NULL);

  /*  Set a pointer to the function values array  */
  function_values = (float *)OMret_name_array_ptr(internal->module_id,
            OMstr_to_name("function_values"), OM_GET_ARRAY_RW,
```

```
            &function_values_size, NULL);

  /*  Read the data into arrays  */
  for (i = 0; i < num_values; i++) {
    if (recv (internal->sd, recvbuf, sizeof(recvbuf), 0) < 0)
      return;

    sscanf (recvbuf, "%f %f %f", &x, &y, &z);
    xy_coords[2*i] = x;
    xy_coords[2*i+1] = y;
    function_values[i] = z;

    msglength = strlen (recvbuf);
    sprintf (sendbuf, "%d", msglength);
    msglength = strlen (sendbuf);
    sendbuf[msglength] = '\0';
    if (send (internal->sd, sendbuf, sizeof(sendbuf), 0) < 0)
      return;
  }

  /* remove our hook in the AVS/Express event mechanism */
  EVdel_select (EV_SELECT0, internal->sd, (void (*)(char*))
                datasoc_read_cb, NULL, (char*) internal,
                EV_SEL_IN);
  (void) close (internal->sd);
  internal->sd = -1;

  /* Free input variables  */
  if (xy_coords != NULL) ARRfree ((char *) xy_coords);
  if (function_values != NULL) ARRfree ((char *) function_values);

}


void
datasoc_connection_cb (datasoc_internal_t *internal)
{
```

```
  /* accept the pending connection */
  internal->sd = accept (internal->lsd, NULL, 0);
  if (internal->sd == -1)
    return;

  /* add our connection handler to the Express event loop */
  EVadd_select (EV_SELECT0, internal->sd,
                (void (*)(char*)) datasoc_read_cb,
                 NULL, (char*) internal, EV_SEL_IN);
}


int
datasoc_instantiation (OMobj_id datasoc_id,
       OMevent_mask event_mask,
       int seq_num)
{
  int  port;
  struct sockaddr_in addr;
  int addr_size;
  datasoc_internal_t *internal;

  /* Allocate and clear our internal data structure */
  internal = malloc (sizeof *internal);
  (void) memset (internal, 0, sizeof *internal);
  internal->module_id = datasoc_id;

  /* create a listening socket */
  internal->lsd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (internal->lsd == -1) {
    free (internal);
    return -1;
  }

  /* bind an address to our listening socket */
  memset (&addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
```

```
addr.sin_port = htons (0);
addr.sin_addr.s_addr = htonl (INADDR_ANY);
if (bind (internal->lsd, (struct sockaddr*) &addr,
    sizeof addr) == -1) {
  (void) close (internal->lsd);
  free (internal);
  return -1;
}
addr_size = sizeof addr;
if (getsockname (internal->lsd, (struct sockaddr*) &addr,
    &addr_size) == -1) {
  (void) close (internal->lsd);
  free (internal);
  return -1;
}

/* listen for connections - this allows clients to connect
   to our socket */
if (listen (internal->lsd, 5) == -1) {
  (void) close (internal->lsd);
  free (internal);
  return -1;
}

/* add our connection handler to the Express event loop */
EVadd_select (EV_SELECT0, internal->lsd,
             (void (*)(char*)) datasoc_connection_cb, NULL,
             (char*) internal, EV_SEL_IN);

 /* Get port's value */
OMset_name_int_val(datasoc_id, OMstr_to_name("port"),
      ntohs (addr.sin_port));

/* set our internal pointer for the module */
OMset_name_ptr_val (datasoc_id, OMstr_to_name ("internal"),
    internal, 0);
```

```
  return(1);
}


int
datasoc_deinstantiation (OMobj_id datasoc_id,
         OMevent_mask event_mask,
         int seq_num)
{
  datasoc_internal_t *internal;

  OMget_name_ptr_val (datasoc_id, OMstr_to_name ("internal"),
        (void**) &internal, 0);
  if (!internal)
    return 1;

  if (internal->lsd != -1) {
    EVdel_select (EV_SELECT0, internal->lsd,
                 (void (*)(char*)) datasoc_connection_cb, NULL,
                 (char*) internal, EV_SEL_IN);
    (void) close (internal->lsd);
  }
  if (internal->sd != -1) {
    EVdel_select (EV_SELECT0, internal->sd,
                 (void (*)(char*)) datasoc_read_cb,
                  NULL, (char*) internal, EV_SEL_IN);
    (void) close (internal->sd);
  }

  free (internal);
  OMset_name_ptr_val (datasoc_id, OMstr_to_name ("internal"),
      0, 0);

  return(1);
}
```

This is the header file **soc.h**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <fcntl.h>
#include <netdb.h>
#include <signal.h>
#include <fstream.h>
#include <strstream.h>
#include <strings.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/uio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/un.h>

#ifndef HAVE_BZERO
#define bzero(ptr, n)  memset(ptr, 0, n)
#endif

#define SERV_TCP_PORT 4005   /*  TCP port */
#define SERV_HOST_ADDR "128.83.121.77"
#define MAXSIZE 512
```

This is the v code datasoc.v for the module DataSoc.

```
module DataSoc<process="user",src_file="datasoc.c",build_dir="socser",
   NEx=22.,NEy=77.,NEwidth=319.,NEheight=264.> {
   omethod+notify_inst instantiation(port+write,
      internal+read+write) = "datasoc_instantiation";
```

```
   omethod+notify_deinst deinstantiation<export=1>(
      internal+read+write) = "datasoc_deinstantiation";
   int port<NEportLevels={0,0},export=1>;
   ptr internal<NEportLevels={0,0},NEvisible=0>;
   int num_values<NEportLevels={0,0},export=1>;
   float xy_coords<NEportLevels={0,2},export=1>[.num_values][2];
   float function_values<NEportLevels={0,2},export=1,NEx=121.,NEy=297.>
   [.num_values];
};
```

This is the v code datafield.v for the module DataField.

```
macro DataField<NEx=517.,NEy=352.,NEwidth=231.,NEheight=275.,
NExOffset=300.,NEyOffset=137.> {
   FLD_MAP.point_mesh point_mesh<NEx=-209.,NEy=0.> {
      coord<NEportLevels={3,0}>;
   };
   FLD_MAP.node_scalar node_scalar<NEx=110.,NEy=-55.> {
      in_data<NEportLevels={3,0}>;
   };
   FLD_MAP.combine_mesh_data combine_mesh_data<NEx=-33.,NEy=99.> {
      in_mesh => <-.point_mesh.out;
      in_nd => <-.node_scalar.out;
   };
   MODS.scat_to_tri scat_to_tri<NEx=44.,NEy=198.> {
      in_field => <-.combine_mesh_data.out;
      out_obj<NEportLevels={1,3}>;
   };
};
```

10 Fortran Interfaces

Data storage associated with each component grid of the Grid Hierarchy 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 a data pointer and bounding box information. In the case of a 3-D Grid Function of type float, this interface would be:

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

Similarly the interface for a bounding box is as follows:
integer lowerbound(3), upperbound(3), shape(3);

The macro FDA has been defined to generate the interface for Fortran data arguments. It passes a pointer to the data.
FDA (arg)

The macro FBA has been defined to generate the interface for Fortran bounding box arguments. It passes the bounding box information for lower bound, upper bound, and shape as arrays.
FBA (bb)

The macro FA combines the two macros. It passes both a pointer to the data and the arrays containing the bounding box information. For example, for the component grid c, at time t, level l, of Grid Function u, the following statement will generate the required interface:
FA (u(t,l,c));