DAGH: User's Guide

Shyamal Mitra, Manish Parashar, & J.C. Browne
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

-

Table of Contents

Chapter 1: Introduction

DAGH (which stands for Distributed Adaptive Grid Hierarchy) was developed as a computational toolkit for the Binary Black Hole NSF Grand Challenge Project. It provides the framework to solve systems of partial differential equations using adaptive finite difference methods. The computations can be executed sequentially or in parallel according to the specification of the user. DAGH also provides a programming interface so that these computations can be performed by traditional Fortran 77 and Fortran 90 or C and C++ kernels.

Motivation for the tutorial

DAGH was developed as a programming infrastructure to support the solutions of partial differential equations, that arise in general relativity, using an adaptive mesh refinement method. It became apparent very soon that it could be used in other disciplines as well. Thus there was a need to explain the conceptual frame work of DAGH and give guidelines on how to use it. A tutorial introduction to the use of DAGH annotated with examples follows.

Goals of the tutorial

The goals for the tutorial are:

- to acquaint you with the adaptive mesh refinement (AMR) technique for the solution of partial differential equations.
- to familiarize you with the conceptual frame work of DAGH and how it is implemented.
- to provide you with a working knowledge of DAGH interfaces through annotated examples.

After you have gone through this tutorial you will be able to write a driver for DAGH and run it in either sequential or parallel mode.

Layout of the tutorial

In Chapter 2 we explain the motivation behind adaptive mesh refinement (AMR) and briefly discuss the concept of AMR. We illustrate the use of adaptive mesh refinements in Chapter 3 through the Berger-Oliger algorithm. We give a sketch of the algorithm and the data structure they suggest using.

DAGH is introduced in Chapter 4. We give a conceptual overview of DAGH, the structure of DAGH based code, and the interface that DAGH provides to applications.

The transport equation, our working example, is defined in Chapter 5. We include a segment of Fortran 90 code that traces the evolution of this equation. We start with a single grid and a single processor. We analyse the structure of the driver and the makefile needed to run it. We discuss the input parameters and output data for this run. We then show what changes are required to run this example on several processors. Finally we analyse an adaptive mesh refinement version of this driver that is run on several processors.

Our simple example does not cover all the classes and functions that are available in DAGH. For more complex problems you may need a larger set of classes or functions. Though, not a part of this tutorial we provide an annotated glossary of all the classes, terms, and interfaces that are available.

Chapter 2: Adaptive Mesh Refinement

Motivation

In the numerical solution of partial differential equations (PDE) a discrete domain is chosen where algebraic analogues of the PDEs are solved. One standard method is to introduce a grid and estimate the values of the unknowns at the grid points through the solutions of these algebraic equations. The spacing of the grid points determines the local error and hence the accuracy of the solution. The spacing also determines the number of calculations to be made to cover the domain of the problem and thus the cost of the computation.

For well behaved problems a grid of uniform mesh spacing (in each of the coordinate directions) gives satisfactory results. However, there are classes of problems where the solution is more difficult to estimate in some regions (perhaps due to discontinuities, steep gradients, shocks, etc.) than in others. One could use a uniform grid having a spacing fine enough so that the local errors estimated in these difficult regions are acceptable. But this approach is computationally extremely costly. Besides for time dependent problems it is difficult to predict in advance a mesh spacing that will give acceptable results.

Principles of Adaptive Mesh Refinement

In the adaptive mesh refinement technique we start with a base coarse grid. As the solution proceeds we identify the regions requiring more resolution by some parameter characterizing the solution, say the local truncation error. We superimpose finer subgrids only on these regions. Finer and finer subgrids are added recursively until either a given maximum level of refinement is reached or the local truncation error has dropped below the desired level. Thus in an adaptive mesh refinement computation grid spacing is fixed for the base grid only and is determined locally for the subgrids according to the requirements of the problem.

Implementation Features

In our implementation, we maintain a *shadow* hierarchy to estimate the local truncation error. The shadow hierarchy is a 2:1 coarser copy of the main grid hierarchy. The grid functions on the main hierarchy are updated along with those on the shadow hierarchy. This is equivalent to taking one integration step in the shadow hierarchy and two integration steps in the main hierarchy. When it is time for regridding, the truncation error is estimated by subtracting the grid functions on the shadow hierarchy from the corresponding values on the main hierarchy. The advantage of this method is that we do not replicate fine grid storage at regridding times.

When a fine grid is created, the function values at the fine grid points are obtained through a linear interpolation of the function values at the grid points of the underlying coarser grid. This initialization of the fine grid point values is known as *prolongation*.

After a fine grid (nested within a coarse grid) has been integrated the coarse grid values are updated by injecting the fine grid solution values onto the coarse grid points. This updating process of the coarse grid values is called *restriction*.

When the program is run on parallel processors we maintain a *ghost* region for intra-grid communication. Suppose we distribute the computation on a grid over several processors we keep a buffer or ghost region along the inner boundaries of each component grid. The values in the ghost region are used to updated the inner boundary values of neighboring component grids.

Chapter 3: Berger-Oliger Method

Applicability of the Berger-Oliger Method

There are several algorithms that use the adaptive mesh refinement techniques. In our tutorial we give an example of one such algorithm - the Berger-Oliger Method [Berger, M.J. & Oliger, J. (1984) J.Comp.Phys., 53, 484-512] which is well suited for the solution of hyperbolic partial differential equations. In their original paper Berger and Oliger implemented their algorithm in one and two dimensions, but this method can be extended to higher dimensions in a straightforward way.

Sketch of the Berger-Oliger Algorithm

The algorithm starts with a rectangular coarse grid. As the solution progresses grid points with high local truncation errors are flagged. Fine grids are created such that all the flagged points are interior to some fine grid. The flagged points are clustered so that the area of the fine grids is minimized.

The fine grids are rectangles. In the original algorithm, these rectangles are rotated with respect to the orientation of the underlying coarse grid. In our implementation of the algorithm, we use rectangles (or cubes for the 3-dimensional cases) whose sides are parallel to the coordinate axes. The advantage of using rectangles (or cubes) is that one can use the same integrator for both the coarse and fine grid.

The fine grids have a separate identity from the underlying coarse grid. They have their own storage and solution vector. But the underlying coarse grid and the fine grid maintain a parent-child relation. Grid generation is applied at each grid level to the flagged points to build the next finer level of grids. For time dependent calculations as the regions that need refinement change so does the pattern of grids. Fine grid levels maybe deleted when not needed. Thus, new grids are created and old grids are destroyed in response to the computation.

If h_l is the mesh spacing for a grid, then for a subgrid with just one higher level of refinement, the mesh spacing is $h_{l+1} = h_l / r$. The refinement factor r is usually some multiple of 2. In practice the same refinement factor is used for all levels of refinement. A constant mesh ratio of the time step to space step is maintained on all the grids. This is achieved by using the same refinement factor for the time step as for the space step. The integration process is such that when one advances one grid step all its subgrids are integrated to the same time value.

Data Structures

The fundamental data structure that is used is a tree or a directed acylic graph of grids. Each grid in the grid hierarchy corresponds to a node in the tree. When a fine grid is nested within a coarse grid, the fine grid is called the child of the coarse grid. Subgrids of the same coarse grid and at the same level of refinement are called siblings. Subgrids at the same level of refinement but not having the same parents are called neighbors. There is a link between neighbors. This is shown in Figure 1.

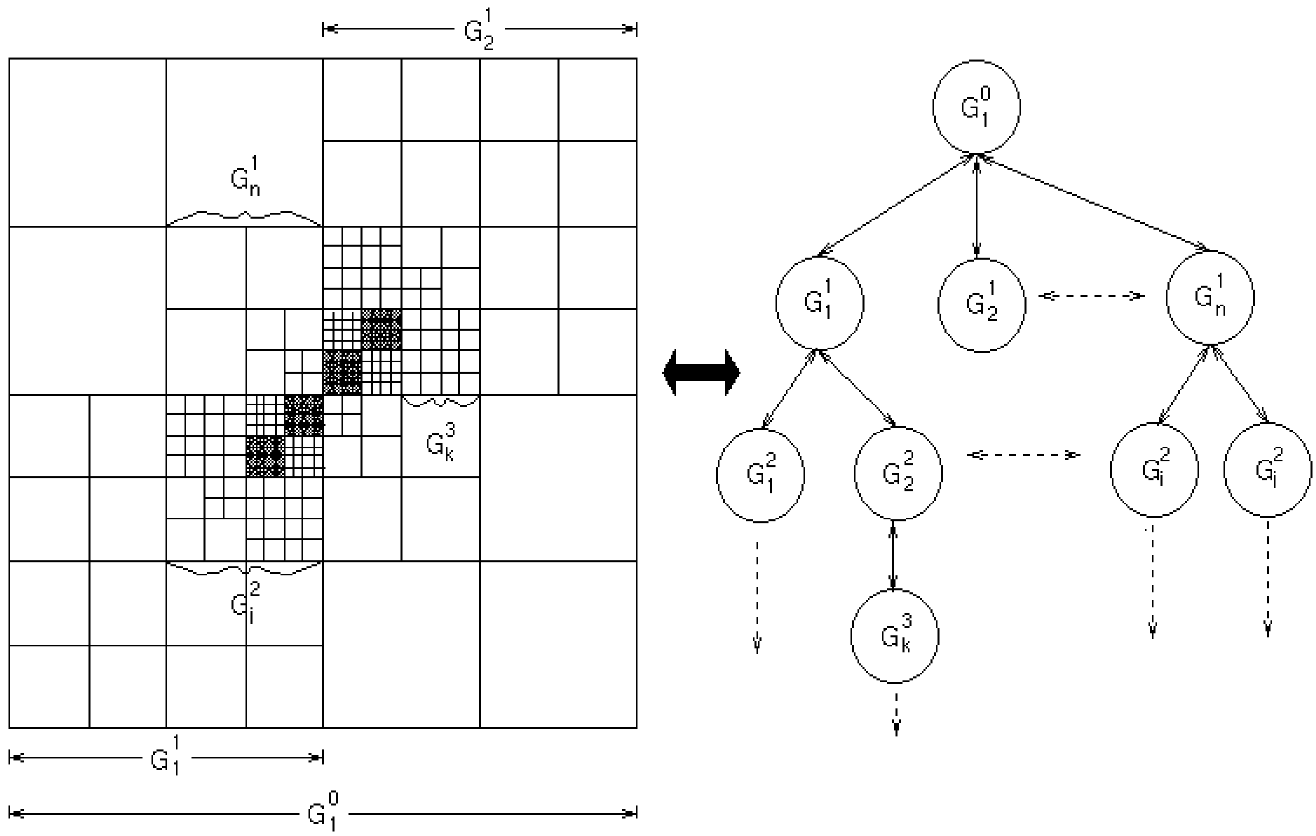

Figure 1: Grid Structure and associated Data Structure

The representations for two and three dimensional grid structures are more complex. Fine grids could be nested in more than one coarse grid. So a single parent node is replaced by a linked list of parent grids. Moreover, grids at the same level of refinement can overlap and hence a pointer to a list of intersecting grids is added to the information attached to each node.

Clustering

The basic problem of grid generation can be stated as follows: given a list of flagged grid points how should rectangular subgrids be placed so that all the flagged points in a coarser grid are interior to some finer subgrid and the area of the subgrids is minimized? There could be several regions of a coarse grid that need refinement. A clustering algorithm groups these regions into a few clusters so that a single refined grid can be generated for all the regions in a cluster. The usual approach is to use a simple clustering algorithm that works in most cases. When the simple algorithm does not work a more expensive algorithm is tried.

A simple clustering algorithm is the nearest neighbor algorithm. This algorithm starts with one flagged point forming a cluster. Successive flagged points are added if the distance from those points to the cluster is less than a specified distance (say two mesh widths).

A more sophisticated algorithm connects the flagged points into a minimal spanning tree. The minimal spanning tree is a connected acyclic graph such that the sum of the edges is a minimum. A cluster is started with just one of the flagged points. The algorithm adds neighboring flagged points and fits a rotated rectangle immediately. The solution is evaluated, i.e. the ratio of the number of flagged grid points to the total number of coarse grid points covered by the fine grid is calculated. The solution gets a favorable evaluation if this ratio is high enough (typically between 1/2 and 3/4). The rectangle grows until there are no more flagged points or it receives an unfavorable evaluation. In case of an unfavorable evaluation a new rectangle is started.

Chapter 4: Introduction to DAGH

Conceptual Overview of DAGH

DAGH provides a program development infrastructure for implementation of solutions of partial differential equations using adaptive mesh refinement algorithms. The principles of hierarchical abstraction and separation of concerns were incorporated into the development of DAGH.

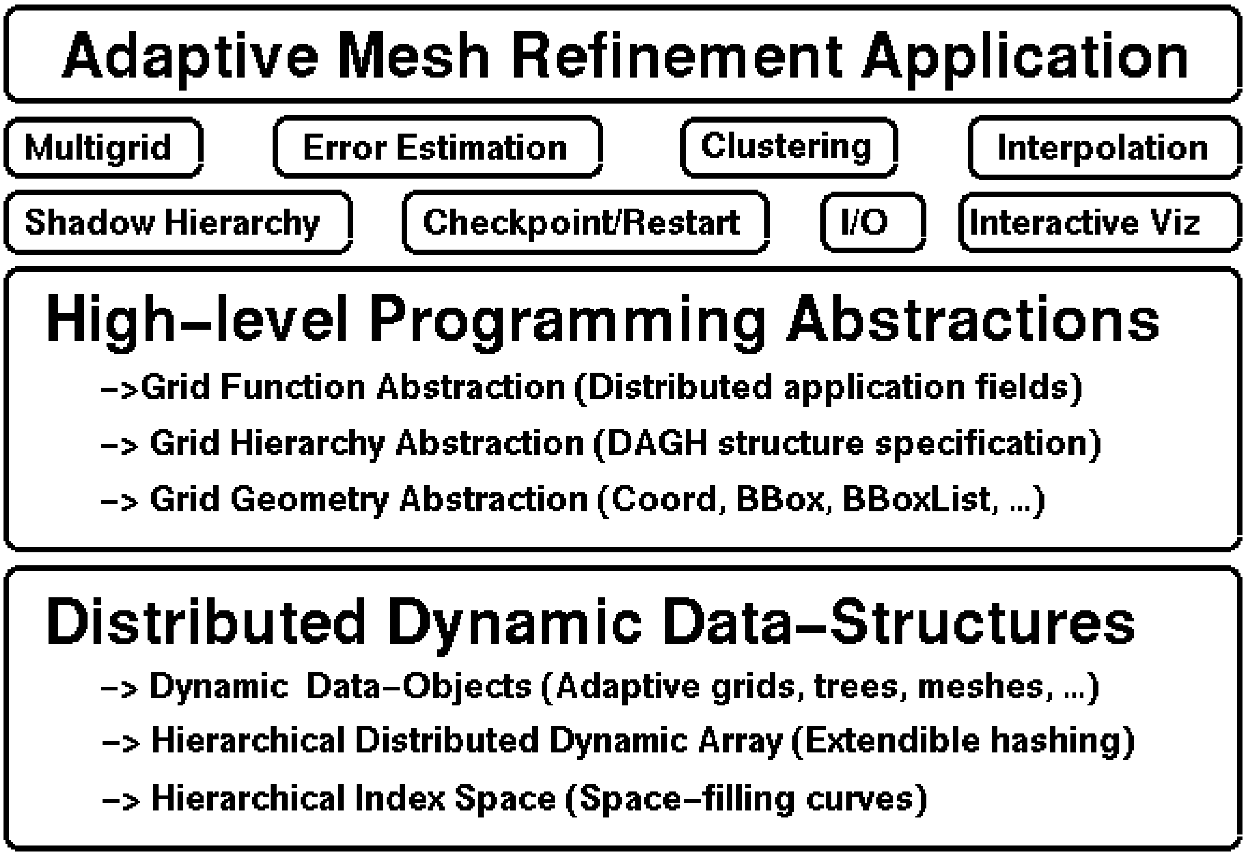

Figure 2: Design model for DAGH

Figure 2 is a schematic design model for DAGH. Each level can be thought of as a set of abstract data types. The lowest level defines a hierarchical dynamic distributed array (HDDA) that is a generalization of the familiar static array. It has the operations of creation, deletion, array expansion and contraction, and array element access defined on it.

We illustrate the separation of concerns by defining a separate level (above HDDA) to implement grids and/or meshes. This abstraction level implements grids by instantiating arrays as components of grids. In the definition of grids are the operations of creation, deletion, expansion, and contraction. These operations are directly translated to operations on instances of HDDA. The definition of grids also includes computational operators, partitioning operators, refinement and coarsening operators, etc.

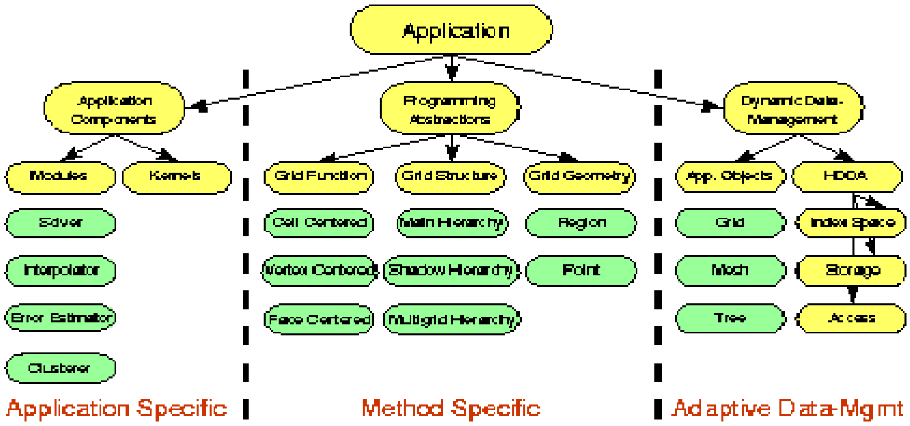

Figure 3: DAGH Abstraction Hierarchy

Figure 3 is another schematic diagram of the levels of abstraction in DAGH. But in this figure the hierarchy descends from left to right. Each level of the hierarchy is given in more detail from top to bottom.

An application consists of specific components. These components are mapped to operations on appropriate subtypes on their right in the programming abstraction layer. The grid subtypes are mapped to the types implemented by the dynamic data management layers to their right.

The Grid Structure abstractions define hierarchical grids and implement standard operations on these grids. These abstractions represent the structure of the geometry of the computational domain. The Grid Function abstractions define application fields on the Grid Structure. These abstractions define the data storage associated with a grid structure. This separation of Grid Function from Grid Hierarchy enables the structure of the computational domain to be defined and manipulated independent of the computational data.

The Grid Geometry abstractions represent regions in grid space that are independent of grid type. These abstractions provide the means for interacting with grids and addressing and directing computations to regions on the grids.

DAGH Data Structures

Hierarchical Dynamic Distributed Array (HDDA) is the data structure abstraction used in DAGH. HDDA is an array data type that has the operations of creation, deletion, array expansion and contraction, and array element access and storage defined on it. Separation of concerns applies vertically within the definition of HDDA. HDDA is composed of three abstractions: index spaces, storage, and access.

Index Space

The index space is a lattice of points in an n-dimensional discrete space. It is recursively defined so that each position in an index space could be an index space.

Storage

This is a mapping from the n-dimensional index space to a one dimensional physical storage. Space filling curves [Hans Sagan, Space-Filling Curves, Springer-Verlag, 1994] allow mappings from a multi-dimensional space to a 1-dimensional space. Thus given an index in the 1-dimensional space one can find its position in the multi-dimensional space.

Access

This is a set of operations for returning the values associated with positions in the index space from the associated storage.

Data storage is implemented using extendible hashing techniques to provide a dynamically extendible, globally indexed storage. Entries in the HDDA correspond to DAGH blocks. Each block in the list is assigned a cost corresponding to its computational load. The grid hierarchy can be partitioned by appropriately partitioning the linear representation of a DAGH instance across processors so as to balance the computational cost associated with each processor.

Structure of DAGH-based code

DAGH provides the class libraries needed to implement adaptive mesh refinement methods and to run a program on parallel processors. All programs that use DAGH have a common structure. This structure can be viewed as a template for application dependent routines which are supplied by the user. To create a DAGH-based application you have to provide -

- Driver

  The driver creates and initializes the DAGH hierarchy, sequences invocations of the solution operations, and interfaces to visualization routines. A detailed discussion of the structure of the driver is given next section.
- Routines to perform grid level operations

  Update functions for
    - Interior points
    - Boundary points

  To update the values of the function at the grid points both inside the region under consideration as well as on the boundary the user supplies the kernels. The kernels can be written in Fortran 77, Fortran 90, C, or C++. It is important to remember that DAGH uses storage maintained in Fortran compatible fashion.
- Routines to perform inter-grid level operations
    - Prolongation
    - Restriction
- Several utility operations
    - I/O routines
    - Visualizations interface
    - Clustering routines

  DAGH provides the I/O framework. The input parameters can be hard coded into the driver or if the DAGH I/O routine is initialized then the input parameters can be read from a file. The output is sent to an IO server. How the output is then written to a file or sent to a visualization interface is defined by the user.
- Routine to perform error estimates

- Input parameter file (optional)
- Makefile

Driver

The driver is a C or C++ program (although drivers can be written in Fortran this option is not discussed in the tutorial). The driver needs to be compiled. One usually uses a makefile to do the compilation. After compilation you can run the executable code on a single processor or use MPI to run it on several processors. The top level view of the structure of the driver is as 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
  - Files

The DAGH Interface

The DAGH interface includes:

- creation and deletion of instances of DAGH's
- definition of logical regions within the DAGH hierarchy (a logical region region that includes a span of the grid hierarchy)
- applications of user defined operations to the logical regions of the grid hierachy

An instance of DAGH typically includes:

- a main grid hierarchy to which the operations are applied
- a shadow hierarchy which is used to store past states of the DAGH for use in error estimations

---

Chapter 5: Tutorial

If DAGH is not already installed on your system refer to the appendix on how to install and compile DAGH. You should be aware of the path name to the directory that DAGH resides in. The source code and other associated files that will be used for the tutorials can be obtained via ftp from the site *ftp.cs.utexas.edu*. Enter *anonymous* for user name and your e-mail address as the password. Download the file *tutorials.tar.Z* from the directory */pub/dagh*. You can uncompress and untar the file by the following command:
% zcat tutorials.tar.Z | tar -xvf -
This will create a directory named *Tutorials*. There will be three sub-directories *UniGridSeq*, *UniGridPar*, and *AmrPar*. Each sub-directory is self contained. They have their own source files, header files, and makefiles that will be used in this tutorial.

The Transport Equation

In our tutorial we will be studying the evolution of a simple 2-dimensional transport equation of the form
u,t + u,x + u,y = 0
where u represents some physical quantity and u,t represents the partial derivative of u with respect to t, and so on. The computational domain is a rectangle. The initial values of u are chosen from a gaussian distributed according to the initial parametrs chosen by the user. We will be using MacCormack scheme which is essentially a predictor-corrector method to study the evolution.

Code Structure in Pseudo-code

The MacCormack integration of the transport equation is presented in pseudo code. The driver replaces the main program and calls each of the subroutines at the appropriate places. The Fortran 90 code that implements the MacCormack integration scheme (sans the main program) is in the file transport.f. The boundaries are updated by calling a built-in DAGH function **BoundaryUpdate** rather than by using a Fortran subroutine to do the same. The reader should now examine the file transport.f to identify the corresponding subroutines in the pseudo code.

Main Program

```
Initialize the number of grid points nx, ny.
Initialize the initial time tzero.
Initialize the number of iterations.
For number of dimensions do
    Initialize lower bound.
    Initialize upper bound.
    Initialize shape information.
End for.
Calculate the mesh separation dx, dy and time step dt.
Initialize the function u.
Output the values of the function u on the computational domain.
Do for number of iterations specified
    Set previous value of function to present value.
    Evolve the function u by one time step.
    Update the boundaries.
    Output the values of u.
End do
```

End Main Program

Subroutine initial

```
Initialize the standard deviation sigma
Initialize the offset from the origin roffset
Do for j = 1 to ny
    Calculate y
    Do for i = 1 to nx
        Calculate x.
        Assign to the function u at coordinates (i,j) the value generated by the gaussian
        function.
    End do
End do
```

End subroutine initial

Subroutine evolve

```
Calculate the backward difference (predictor step).
Calculate the forward difference and average with old value (corrector step).
```

End subroutine evolve

Subroutine boundaries

Shift the values of the function lying within and next to the boundary to the boundary.

End subroutine boundaries

All the subroutines except the subroutine **boundaries** are in the file transport.f and will be used in the example that follows. We will be using the transport equation to illustrate how to set up the driver for DAGH and run the program in three different situations. In the simplest case we will use just one grid and run our program on a single processor. We will then run the single grid on several processors. Finally we will implement the adaptive mesh refinement technique and run our program on multiple processors.

Unigrid Sequential Code

We need a driver (tportseq.C) to run the Fortran code (transport.f). The reader should print out tportseq.C and examine the appropriate segments as he reads the chapter. The driver needs a header file (tportfortran.h) to use the Fortran subroutines. We have the Makefile to create the executable code.

Structure of the driver tportseq.C

We begin with the pre-processor directives to include the header files for DAGH and the Fortran code.

```
#include <DAGH.h>
#include "tportfortran.h"
```

If you have xgraph running on your system insert the path name to its directory replacing the path name defined below. Redefine H3eDISPLAY according to the example shown for the machine you are running your program on. If you do not have xgraph leave the comments in place.

```
#define H3eXGRAPH "/us5/parashar/bin/xgraph"
#define H3eDISPLAY "indra.ticam.utexas.edu:0.0"
```

We will define a grid structure that is a square having 32 grid points along any coordinate axis. We want the number of iterations to be 64. We define a shape array that gives the number of grid points along a certain coordinate axis. We also define an array bb that gives the lower and upper bounds of the square defining the computational domain. In this example instead of initializing nx, ny, etc. by reading the values through the DAGH I/O routines we have chosen to hard code our input parameters in the driver.

```
const INTEGER nx = 32;
const INTEGER ny = 32;

const INTEGER iterations = 2 * nx;

INTEGER shape[2];
DOUBLE bb[2*2];

shape[0] = nx;
bb[0] = 0.0;
bb[1] = 1.0;

shape[1] = ny;
bb[2] = 0.0;
bb[3] = 1.0;
```

There are several DAGH function calls to instantiate the grid structure. We want to establish a grid hierarchy GH. This is a 2-dimensional hierarchy, where the grid values are estimated at the vertices of the cells, and we have a single level, i.e. a uni-grid.

```
GridHierarchy GH (2, DAGHVertexCentered, 1);
```

We define the base grid or the coarsest grid in the grid hierarchy GH to be a grid defined by nx, ny, and bb. The bounding box has lower and upper bounds stored in the array bb and the shape array contains the number of grid points within the bounding box.

```
SetBaseGrid (GH, bb, shape);
```

The boundary points of this grid structure are spaced at intervals of 1 cell width. The values in the cells along the boundary are treated differently from the interior points.

```
SetBoundaryWidth (GH, 1);
```

ComposeHierarchy would, in case of a multi-processor execution, set up distributed data structures. We include a call to this function for the sake of completeness.

```
ComposeHierarchy (GH);
```

The time_stencil is set to 1 to be used in the GridFunction. Storage is maintained for 3 (= 2 * time_stencil + 1) time levels numbered from -1 to +1. The space_stencil is set to 1. It defines the "ghost" regions maintained around distributed grid blocks for inter-grid communication and updating.

```
INTEGER t_sten = 1, s_sten = 1;
```

The GridFunction associates the values from the function u with grid points in the GridHierarchy. It defines storage for these values. We have a 2-dimensional grid function, of type double, with a time_stencil of 1 and a space_stencil of 1 that returns values of for points in space and time.

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

The MacCormack method allows storage of the values of the function u to be shared between time step +1 from current time and time step -1 from current time.

```
SetTimeAlias (u, 1, -1);
```

We want the values on the inner cells next to the boundary to be shifted to the boundary.

```
SetBoundaryType (u, DAGHBoundaryShift);
```

MPI sets up a virtual machine having a network of processors. To obtain the number of processors and the local processor number the following function calls are made. NUM_PROC returns the number of processors in this network. MY_PROC returns the local processor number.

```
INTEGER me = MY_PROC(GH);
INTEGER num = NUM_PROC(GH);
```

In this example we have a single grid, so the level is set to 0.

```
INTEGER Level = 0;
```

CurrentTime returns the number of integration time steps that have accrued for that level. TimeStep returns the time interval used for integration at that level. For our single grid model the TimeStep will always be 1.

```
INTEGER Time = CurrentTime (GH, Level);
INTEGER TStep = TimeStep (GH, Level);
```

The space intervals are defined and intialized. tzero is the initial time needed for the initialization routine. The time step dt is calculated from the Courant condition.

```
DOUBLE dt, dx, dy, tzero = 0.0;
const DOUBLE dtfac = 0.2;
const DOUBLE sqrt3 = 1.7321;
dx = 1.0 / (nx - 1.0);
dy = 1.0 / (ny - 1.0);
dt = dtfac * dx / sqrt3;
```

The data storage associated with each component grid of the GridHierarchy is maintained so that it can be operated on by Fortran 77/90 kernels. The data associated with a component grid is passed to a Fortran subroutine as a combination of the data pointer and bounding box information. The macro **FA** has been defined to automatically generate these Fortran interfaces. The macro call **FA(u(Time,Level,c,DAGH_Main))** will provide the Fortran interface for the component grid c, at the current time and base level in the main grid hierarchy (DAGH_Main).

The initialization routine resides in the file transport.f provided by the user. The call **f_initial** invokes this Fortran routine. The **forall** operator performs the data parallel operations on all component grids (c) at a particular time (Time) and level (Level).

```
forall (u, Time, Level, c)
   f_initial (FA(u(Time,Level,c,DAGH_Main)),&tzero,&dx,&dy);
end_forall
```

We want to set up the main loop to iterate through the evolution of the physical conditions described by the partial differential equation. But before we begin the loop we will write out the maximum, minimum, and the norm. You can also see the result of the initialization visually if you have xgraph running. Leave the comments in place if you do not have xgraph.

```
INTEGER currentiter = 0;
cout << me  << ": Iteration: " << currentiter
             << " Max: " << MaxVal (u,Time,Level,DAGH_Main)
             << " Min: " << MinVal (u,Time,Level,DAGH_Main)
             << " Norm: " << Norm2 (u,Time,Level,DAGH_Main) << endl;

/*
  BBox viewbbX (2, 0, ny/2, nx-1, ny/2, 1);
  BBox viewbbY (2, nx/2, 0, nx/2, ny-1, 1);

  View (u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
        H3eXGRAPH, H3eDISPLAY, DAGH_Main);
  View (u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
        H3eXGRAPH, H3eDISPLAY, DAGH_Main);
*/

for (currentiter=1; currentiter <= iterations; currentiter++) {
```

The function values of u for a time step before the current time is initialized to the values of the function u at the current time. This is done for base level, and for all component grids c (in this case only the base level).

```
forall (u, Time, Level, c)
   u (Time-TStep, Level, c, DAGH_Main) = u (Time, Level, c, DAGH_Main);
end_forall
```

We will call the predictor subroutine from the Fortran routine provided by the user, across all the grid components for the base level.

```
forall (u, Time, Level, c)
   f_evolveP (FA (u (Time, Level, c, DAGH_Main)),
                    FA (u (Time-TStep, Level, c, DAGH_Main)),
                    &dt, &dx, &dy);
end_forall
```

Sync updates the boundary values of the various grids using the values in the ghost regions.

```
Sync (u, Time, Level, DAGH_Main)
```

We will call the corrector subroutine from the Fortran program across all the grid components for the base level l.

```
forall (u, Time, Level, c)
   f_evolveC (FA (u (Time, Level, c, DAGH_Main)),
                   FA (u (Time-TStep, Level, c, DAGH_Main)),
                   &dt, &dx, &dy);
end_forall
```

Once again the boundary values of the various grids need to be updated using the values in the ghost regions.

```
Sync (u, Time, Level, DAGH_Main)
```

Finally, the boundary values are updated by shifting the values just interior to the boundary to the boundary.

```
BoundaryUpdate (u, Time, Level, DAGH_Main);
```

We will print out the maximum, minimum, and the norm. We can also visualize the result on xgraph. Leave the comments in place if you are not running xgraph.

```
cout << me << ": Iteration: " << currentiter
            << " Max: " << MaxVal (u,Time,Level,DAGH_Main)
            << " Min: " << MinVal (u,Time,Level,DAGH_Main)
            << " Norm: " << Norm2 (u,Time,Level,DAGH_Main) << endl;

/*
if (currentiter % 10 == 0) {
   View (u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
         H3eXGRAPH, H3eDISPLAY, DAGH_Main);
   View (u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
         H3eXGRAPH, H3eDISPLAY, DAGH_Main);
 }
*/
```

Structure of Makefile

We assume that the reader is familiar with the Unix make utility. Issuing the command *man make* will bring up the on-line manual pages on the subject. This is a template of a makefile. You will have to modify this template so that it runs on your system. You must know where the compiled version of DAGH resides on your system.

The actual path name where DAGH resides in your system needs to go in here.

```
DAGH_HOME = /u8/mitra/NDAGH
```

make.defn is generated by Autoconf. The actual path name where it resides may be subsituted here or you can copy *make.defn* into the directory you are running your code. We recommend the latter.

```
include make.defn
```

All the source codes and object codes that are required have to be entered.

```
APP_F_SRC = transport.f
APP_F_OBJ = $(APP_F_SRC:.f=.o)

APP_CC_SRC = tportseq.C
APP_CC_OBJ = $(APP_CC_SRC:.C=.o)

APPOBJ = $(APP_F_OBJ) $(APP_CC_OBJ)
```

All the application specific flags go here.

```
C++APPFLAGS = -g
CAPPFLAGS = -g
F77APPFLAGS = -g
F99APPFLAGS = -g
```

In this example we are calling the executable code produced *tportseq.*

```
EXEC = tportseq
```

This tells make what object files and libraries to link to create the executable code.

```
$(EXEC) :        $(APPOBJ)
                 $(RM) $(EXEC)
                 $(C++LINK)  $(C++FLAGS) -o $@ \
                 $(APPOBJ) $(APPLIB) $ (LDLIBS)
```

This command allows the user to remove any core files, object files, and compiler created template repository files before link and compile are executed.

```
clean:
        $(RM) -f *.o core $(EXEC)
        $(RM) -r $(REPOSITORY)
```

Running your DAGH program

You first clean your directory by issuing the command
%make clean

Then you create the executable code with the command
%make tportseq

Finally to run your code type
%tportseq

Input data and Outputs for Sequential Unigrid Code

In our sequential unigrid code we did not utilize the DAGH I/O routine but inserted the input parameters in the driver. We initialized the grid structure by setting up nx, and ny the number of grid points along the coordinate axes. We set the number of iterations to be 64. We intialized the shape array as well as the lower and upper bounds for the grid structure. We also defined the ghost regions and set the boundary width. We obtained as output the maximum, minimum, and the norm at every iteration.

Unigrid Parallel Code

There are only a few modifications to be made to run our program on parallel processors. The driver is in the file tportpar.C. The reader should examine *tportpar.C* and compare it to *tportseq.C.* In the driver we initalize the MPI environment. The MPI (Message Passing Interface) sets up a virtual machine that can access one or more processors according to how it is configured.

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

At the end of the driver we close the MPI environment. The virtual machine set up by MPI is turned off.

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

The Makefile remains almost the same (a few name changes from *tportseq* to *tportpar*), and so does the process to create the executable code.
%make clean
%make tportpar

To run the code on parallel processors the following command works on our machines:
%mpirun -np4 tportpar

The np4 indicates that the number of processors to be used is 4. On your system, you may need a different command to run MPI. Your system administrator can help you in this regard.

AMR Parallel Code

We add the adaptive mesh refinement to the program in this chapter. The driver (tportamr.C) is more complex than before. The user must define clustering, mesh refinement, error estimation, etc. The prolongation and restriction routines in this example are found in grid.f. To cluster the flagged points we need the routines Cluster1.C and Cluster3.C. The file tportutil.f has subroutine *readinput* that reads the input parameters from a file input.par. The truncation error estimate code is in the file truncation.C. The same makefile with a few additions can be used to obtain the executable code and the same command to run on the program on several processors using MPI.

Analysis of the driver tportamr.C

We will define a flag to indicate that processor number 0 will write to stdout.

```
#define VizServer 0
```

If you have xgraph running include the following two lines. Substitute the path name where xgraph resides on your system and redefine H3eDISPLAY with the name of the machine you are working on. If you do not have xgraph comment out these lines.

```
#define H3eXGRAPH "/u5/parashar/xgraph/xgraph"
#define H3eDISPLAY "indra.ticam.utexas.edu:0.0"
```

We will give the pre-processor directive to include the DAGH header files.

```
#include "DAGH.h"
#include "DAGHCluster.h"
```

The application specific header files are also included.

```
#include "tportamr.h"
#include "tportfortran.h"
```

We will set the default AMR parameters. These parameters will be read in from the input file later on. The maximum level of refinement is set at 1, indicating a single grid. A buffer zone is added around the fine grid. The larger the buffer zone the longer the time interval over which the grids are stable and the time between regridding is lengthened. But greater the buffer width, the more work is needed to integrate the extra points in the buffer zone. The block width defines the minimum size of the cluster. The minimum efficiency of clustering is given by the ratio of the number of flagged points to the number of grid points.

```
INTEGER MaxLev = 1;           // Maximum level of refinement
INTEGER BufferWidth = 1;      // Buffer width used by clusterer
INTEGER BlockWidth = 1;       // Minimum granularity used by clusterer
DOUBLE MinEfficiency = 0.7;   // Minimum efficiency of clustering
GFTYPE Thresh = 0.0;          // Truncation error threshold for regriding
INTEGER RegridEvery = 4;      // How often do I regrid?
INTEGER NumIters = 0;         // Number of timesteps on the base grid
```

We start the main function and initialize MPI.

```
void main (INTEGER argc, CHARACTER *argv[])
{
   cout << "Initializing MPI \n";
   MPI_Init (&argc, &argv);  // Initialize MPI
```

We will declare or define the local variables used.

```
INTEGER nx, ny;                     // Number of grid points
DOUBLE dx, dy, dt;                  // Grid spacings
DOUBLE xmin, xmax, ymin, ymax;      // Grid extent
DOUBLE cfl;                         // Courant factor
INTEGER niters = 0;                 // Number of iterations
INTEGER ml;                         // Maximum number of levels
INTEGER regride;                    // Regridding time interval
INTEGER buffw;                      // Buffer width
INTEGER blkw;                       // Minimum block width
GFTYPE thresh = 0;                  // Error threshold
DOUBLE mineff = 0;                  // Minimum efficiency
```

We will call the fortran subroutine *readinput* to read the input parameters from the file *input.par*. After reading the input file the AMR parameters get assigned.

```
f_readinput (
      &nx, &ny,          // Number of grid points
      &xmin, &xmax,      // X Coords range
      &ymin, &ymax,      // Y Coords range
      &cfl,              // Courant factor
      &niters,           // Number of iterations
      &ml,               // Max levels
      &thresh,           // Threshold
      &regride,          // Regrid every
      &mineff,           // Min efficiency
      &buffw,            // Buffer width
      &blkw,             // Min block width
      &outevery);        // Output every time step

MaxLev = ml;
RegridEvery = regride;
BufferWidth = buffw;
BlockWidth = blkw;
MinEfficiency = mineff;
Thresh = thresh;
NumIters = niters;
OutEvery = outevery;
```

We will calculate the coarse grid spacing and the time step.

```
dx = (xmax - xmin) / (DOUBLE) (nx - 1);
dy = (ymax - ymin) / (DOUBLE) (ny - 1);
dt = dx * cfl;
```

We will set up a 2-dimensional grid hierarchy GH. The constant DIM is set to 2 in the header files. The arrays giving the shape of the grid and the bounding boxes are declared and initialized. The refinement factor is set at 2, which implies that the mesh spacings are halved at each finer level. The DAGHNoBoundary is a directive not to do anything on the external boundary.

```
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, 2);
SetBoundaryWidth (GH, 1);
SetBoundaryType (GH, DAGHNoBoundary);
```

The data structures need to be distributed across processors.

```
ComposeHierarchy (GH);
```

MPI sets up a virtual machine having a network of processors. NUM_PROC returns the number of processors in this network. MY_PROC returns the local processor number.

```
INTEGER me = MY_PROC (GH);
INTEGER num = NUM_PROC (GH);
```

A ghost region is maintained around the distributed grid blocks for inter-grid communication and updating. The space stencil defines the width of the ghost region. The space stencil is set to 1. The time stencil is also set to 1. Storage is maintained for 3 (= 2 * t_sten + 1) time levels numbered from -1 to +1.

```
INTEGER t_sten = 1;
INTEGER s_sten = 1;
```

The GridFunction associates the values from the function u with grid points in the GridHierarchy. It defines storage for these values and establishes communication. In our example we want communication on the face of the cells only. We also want a shadow hierarchy to be used for error estimation.

```
GridFunction (DIM)<GFTYPE> u("u", t_sten, s_sten, GH, DAGHCommFaceOnly,
DAGHHasShadow);
```

To reduce the amount of storage we will share storage of the values of the function u. Storage is shared between time step +1 from the current time and time step -1 from current time.

```
SetTimeAlias (u, 1, -1);
```

The boundary values are treated differently. We want to shift the values on the inner cells of the boundary to the boundary.

```
SetBoundaryType (u, DAGHBoundaryShift);
```

A similar GridFunction is defined for the function *temp* .

```
GridFunction (DIM)<GFTYPE> temp("temp", t_sten, s_sten, GH, DAGHCommFaceOnly,
DAGHHasShadow);
SetTimeAlias (temp, 1, -1);
```

The prolongation function maps the values at the coarse grid points to the next finer level of grid points. The restriction function maps the values at the fine grid points to the next level of coarser grid points. We do this for all the distributed grid structures in the hierarchy.

```
foreachGF (GH, GF, DIM, GFTYPE)
   SetProlongFunction (GF, (void *) &f_prolong_amr);
   SetRestrictFunction (GF, (void *) &f_restrict_amr);
end_foreachGF
```

We will declare and initialize the local variables for the base or coarse grid.

```
INTEGER Level = 0;
INTEGER Time = CurrentTime (GH, Level, DAGH_Main);
INTEGER TStep = TimeStep (GH, Level, DAGH_Main);
```

We will initialize the main grid structure and the shadow using the Fortran initialization routine.

```
GFTYPE tzero = 0.0;

forall (u, Time, Level, c)
   f_initial (FA(u(Time,Level,c,DAGH_Main)),&tzero,&dx,&dy);
end_forall

forall (u, Time, Level, c)
   f_initial (FA(u(Time,Level,c,DAGH_Shadow)),&tzero,&dx,&dy);
end_forall
```

If you have xgraph you can see the result of the initialization. If you do not have xgraph leave this section commented out.

```
const int s = StepSize (GH, Level, DAGH_Main);
const BBox gbb = GH.glbbbox();
const int snx = gbb.upper(0);
const int sny = gbb.upper(1);
BBox viewbbX (2, 0, sny/2, snx, sny/2, s);
BBox viewbbY (2, snx/2, 0, snx/2, sny, s);

View (u, Time, Level, DAGH_X, viewbbX, 0, DAGHViz_XGRAPH, H3eXGRAPH,
H3eDISPLAY, DAGH_Main);
View (u, Time, Level, DAGH_Y, viewbbY, 0, DAGHViz_XGRAPH, H3eXGRAPH,
H3eDISPLAY, DAGH_Main);
```

We will start the main iterative loop. This loop carries out the evolution of the transport equation over all the grids at all levels. A recursive function *bno_RecursiveIntegrate* is used to integrate at time levels n and n-1.

```
INTEGER currentiter = 0;

for (currentiter=1; currentiter<=NumIters; currentiter++) {
   bno_RecursiveIntegrate (GH, Level, u, temp, dt, h, cfl);
```

The built-in functions MaxVal and MinVal return the maximum and minimum values. To obtain the result we need to define two local variables for the current time and the time step.

```
Time = CurrentTime (GH, Level, DAGH_Main);
TStep = TimeStep (GH, Level, DAGH_Main);

cout << me << " -> Iteration: " << currentiter
     << " MaxVal: " << MaxVal (u, Time, Level, DAGH_Main)
     << " MinVal: " << MinVal (u, Time, Level, DAGH_Main) << endl;
```

If you have xgraph running you can display the results of the evolution, otherwise leave this section commented out.

```
View (u, Time, Level, DAGH_X, viewbbX, 0, DAGHViz_XGRAPH, H3eXGRAPH,
H3eDISPLAY, DAGH_Main);
View (u, Time, Level, DAGH_Y, viewbbY, 0, DAGHViz_XGRAPH, H3eXGRAPH,
H3eDISPLAY, DAGH_Main);
```

This is the end of the loop. We will close the MPI environment, and that will be the end of the main function.

```
DAGHMPI_Finalize();
```

Structure of the recursive function

```
void bno_RecursiveIntegrate (
        GridHierarchy &GH,
        INTEGER const Level,
        GridFunction (DIM) <GFTYPE> &u,
        GridFunction (DIM) <GFTYPE> &temp,
        DOUBLE &dt, DOUBLE const &cfl)
```

We need to get the processor information. *me* is the id of the current processor and *num* is the number of processors that the data structures are distributed over.

```
INTEGER me = MY_PROC (GH);
INTEGER num = NUM_PROC (GH);
```

We select the number of iterations to run based on the current grid level.

```
INTEGER NoIterations = 0;
if (Level == 0)
   NoIterations = 1;
else
   NoIterations = RefineFactor (GH);
```

We take the recursive steps. First we declare and initialize the local variables.

```
for (INTEGER i = 0; i < NoIterations; i++) {
   INTEGER Time = CurrentTime (GH, Level, DAGH_Main);
   INTEGER TStep = TimeStep (GH, Level, DAGH_Main);
```

We will check if it is time for re-gridding. We get the truncation error from the user supplied error estimation routine *wave_trunc_est* and use the error threshold read in to determine if it is re-gridding time. This check and regridding is done in the function *bno_RegridSystemAbove*.

```
if (Level < MaxLevel (GH) &&
   StepsTaken (GH, Level, DAGH_Main) % RegridEvery == 0 &&
   StepsTaken (GH, Level, DAGH_Main) != 0){
      wave_trunc_est (GH, t, l, u, temp);
      bno_RegridSystemAbove (GH, temp, Level, Thresh);
  }
```

We will take one step on DAGH_Main hierarchy. The sequence of steps are similar to the ones we saw for the uni-grid sequential code. We begin by defining the grid spacing and the time step. We call the predictor subroutine. We update the boundary values of the various grids using the values in the ghost regions. We call the corrector subroutine. We do another update of the boundary using the values in the ghost region. Finally we shift the values just interior to the boundary to the boundary.

```
DOUBLE dagh_dx = DeltaX (GH, DAGH_X, Level, DAGH_Main);
DOUBLE dagh_dy = DeltaY (GH, DAGH_Y, Level, DAGH_Main);
DOUBLE dagh_dt = dagh_dx cfl;

forall (u, Time, Level, c)
   f_evolveP (FA (u (Time+TStep, Level, c, DAGH_Main)),
              FA (u (Time, Level, c, DAGH_Main)),
              &dagh_dt, &dagh_dx, &dagh_dy);
```

```
end_forall

Sync (u, Time, Level, DAGH_Main);

forall (u, Time, Level, c)
   f_evolveC (FA (u (Time+TStep, Level, c, DAGH_Main)),
              FA (u (Time, Level, c, DAGH_Main)),
              &dagh_dt, &dagh_dx, &dagh_dy);
end_forall

Sync (u, Time, Level, DAGH_Main);

BoundaryUpdate (u, Time, Level, DAGH_Main);

Sync (u, Time, Level, DAGH_Main);
```

We take a step on the shadow hierarchy. The shadow hierarchy is used to estimate the local truncation error. We define the grid spacings and time step. Then we update the shadow hierarchy from the main. The next sequence of steps are similar to the steps we took on the main hierarchy as discussed above.

```
if (StepsTaken(GH,Level,DAGH_Main) % RefineFactor(GH) == 0) {
   DOUBLE dagh_dx = DeltaX (GH, DAGH_X, Level, DAGH_Shadow);
   DOUBLE dagh_dy = DeltaY (GH, DAGH_Y, Level, DAGH_Shadow);
   DOUBLE dagh_dt = dagh_dx * cfl;

   forall (u, Time, Level, c)
      u (Time, Level, c, DAGH_Shadow) = u (Time, Level, c, DAGH_Main);
   end_forall

   Sync (u, Time, Level, DAGH_Shadow)

   forall (u, Time, Level, c)
      f_evolveP (FA (u (Time+TStep, Level, c, DAGH_Shadow)),
                 FA (u (Time, Level, c, DAGH_Shadow)),
                 &dagh_dt, &dagh_dx, &dagh_dy);
   end_forall

   Sync (u, Time, Level, DAGH_Shadow);

   forall (u, Time, Level, c)
      f_evolveC (FA (u (Time+TStep, Level, c, DAGH_Shadow)),
                 FA (u (Time, Level, c, DAGH_Shadow)),
                 &dagh_dt, &dagh_dx, &dagh_dy);
   end_forall

   Sync (u, Time, Level, DAGH_Shadow);

   BoundaryUpdate (u, Time, Level, DAGH_Shadow);

   Sync (u, Time, Level, DAGH_Shadow);
```

If we are not at the finest level go to the next higher level and do a recursive integration at that level.

```
if (Level < FineLevel (GH)) {
   bno_RecursiveIntegrate (GH, Level+1, u, temp, dt, cfl);
   }
```

We will move from the current time to the previous time.

```
CycleTimeLevels (u, Level);
```

The time step at the current level needs to be incremented.

```
IncrCurrentTime (GH, Level, DAGH_Main)
```

Finally we will map the function values at the fine grid points to the next level of coarser grid points.

```
if (Level < FineLevel (GH)) {
   Time = CurrentTime (GH, Level, DAGH_Main);
   INTEGER myargc = 0; GFTYPE myargs[1];
   Restrict (u, Time, Level+1, t, Level, myargs, myargc, DAGH_Main);
   Sync (u, Time, Level, DAGH_Main);
    }
```

Structure of the function bno_RegridSystemAbove

This function clusters the flagged points, regrids, and redistributes the coarse and fine grid blocks over the various processors.

```
void bno_RegridSystemAbove (
        GridHierarchy &GH,
        GridFunction (DIM) <GFTYPE> &u,
        INTEGER const Level, GFTYPE const thresh) {

        INTEGER me = MY_PROC (GH);
        INTEGER num = NUM_PROC (GH);
```

We will compute the levels to regrid from the finest level down to the current level.

```
INTEGER flev = FineLevel (GH);
INTEGER lev = 0;
if (flev == 0) lev = 0;
else if (flev == MaxLev - 1) lev = flev - 1;
else lev = flev;
```

We declare a local bounding box list. Then we loop from the the finest level to the current level. The flagged points are clustered using a 3-dimensional clustering algorithm. A refined grid is overlaid over the flagged points that have been clustered.

```
BBoxList bblist;

for (; lev >= Level; lev--) {
   INTEGER Time = CurrentTime (GH, lev, DAGH_Main);
   INTEGER TStep = TimeStep (GH, lev, DAGH_Main);

   DAGHCluster3d (u, Time, lev,
                  BlockWidth, BufferWidth,
                  MinEfficiency, thresh,
                  bblist, bblist, DAGH_Main);

   Refine (GH, bblist, lev);
}
```

Finally the coarse and fine grid blocks are redistributed over the various processors.

```
RecomposeHierarchy (GH);
```

---

transport.f

```
c***********************************************************************
c
c      (c) Joan Masso: 28 April 1994: quick and dirty 2d transport equation
c      Extended on 19 Apr 1995 to use fortran 90 instead of f77
c
c      It evolves the equation:
c                                u,t + u,x + u,y = 0
c      Using a maccormack scheme.
c      The initial data is a cruddy gaussian.
c      Boundaries are flat: copying the value of the neighbour
c
c***********************************************************************

c      Initializes the field with a cruddy gaussian
       subroutine initial(
     $     u,ulb,uub,ushape,
     $      time,dx,dy)

       implicit none
       integer ulb(2), uub(2), ushape(2)
       real*8 u(ushape(1),ushape(2))

       real*8 dx,dy

       real*8 time
       real*8 roffset
       real*8 sigma
       parameter  (roffset=0.8)
       parameter (sigma=.1)

       integer nx,ny
       integer sx,sy
       integer i,j
       real*8 x,y

       nx = ushape(1)
       ny = ushape(2)

       sx = (uub(1)-ulb(1))/(nx-1)
       sy = (uub(2)-ulb(2))/(ny-1)

       do j = 1, ny
          y = (ulb(2)+(j-1)*sy)*dy
          do i=1,nx
             x = (ulb(1)+(i-1)*sx)*dx
             u(i,j) = exp(-((sqrt(x**2+y**2)-roffset-time)
     $                /sigma)**2)
           enddo
        enddo
```

```
      return
      end
c
c
c     ****************************************************************
c
c     evolve the field.

      subroutine evolveP(
     *     u,ulb,uub,ushape,
     *     upp,uplb,upub,upshape,
     *     dt,dx,dy);
      implicit none

      integer ulb(2), uub(2), ushape(2)
      real*8 u(ushape(1),ushape(2))
      integer uplb(2), upub(2), upshape(2)
      real*8 upp(upshape(1),upshape(2))

      real*8 dt,dx,dy

      integer i,j
      integer nx,ny
      nx = ushape(1)
      ny = ushape(2)

c     Predictor step: backward derivatives. Note that we get a predicted
c     value at the exterior.
      u(2:,2:) = upp(2:,2:)
     $     - dt/dx* ( upp(2:,2:) - upp(1:nx-1,2:) )
     $     - dt/dy* ( upp(2:,2:) - upp(2:,1:ny-1) )

      return
      end

c     ************************************************************

      subroutine evolveC(
     $     u,ulb,uub,ushape,
     $     upp,uplb,upub,upshape,
     $     dt,dx,dy);
      implicit none

      integer ulb(2), uub(2), ushape(2)
      real*8 u(ushape(1),ushape(2))
      integer uplb(2), upub(2), upshape(2)
      real*8 upp(upshape(1),upshape(2))

      real*8 dt,dx,dy

      integer i,j
      integer nx,ny
      nx = ushape(1)
      ny = ushape(2)

c     Corrector step: forward derivatives. Average corrector with old value.
      u(2:nx-1,2:ny-1) =  (
     $     u(2:nx-1,2:ny-1) + upp(2:nx-1,2:ny-1)
     $     - dt/dx*(u(3:nx,2:ny-1)-u(2:nx-1,2:ny-1))
     $     - dt/dy*(u(2:nx-1,3:ny)-u(2:nx-1,2:ny-1)))/2.0
```

```
      return
      end
```

tportseq.C

```
//------------------------------------------------------------------------
//
// DAGH Driver for the Transport Application
// Unigrid Sequential Operation
//
//------------------------------------------------------------------------
#include <DAGH.h>
#include "tportfortran.h"

// If you have xgraph running on your system insert the path name
// to its directory. Redefine H3eDISPLAY for the machine you are
// running your program. Otherwise leave the comments in place.
/*
#define H3eXGRAPH "/u5/parashar/bin/xgraph"
#define H3eDISPLAY "indra.ticam.utexas.edu:0.0"
*/

void main(INTEGER argc, CHARACTER *argv[])
{

  const INTEGER nx = 32;
  const INTEGER ny = 32;

  const INTEGER iterations = 2*nx;

  // Set up the Grid Structure
  INTEGER shape[2];
  DOUBLE bb[2*2];

  shape[0] = nx;
  bb[0] = 0.0;
  bb[1] = 1.0;

  shape[1] = ny;
  bb[2] = 0.0;
  bb[3] = 1.0;

  GridHierarchy GH(2,DAGHVertexCentered,1);
  SetBaseGrid (GH,bb,shape);
  SetBoundaryWidth(GH,1);
  ComposeHierarchy(GH);

  // Setup the GridFunctions....
  INTEGER t_sten = 1, s_sten = 1;
  GridFunction(2)<DOUBLE> u("u",t_sten,s_sten,GH);
  SetTimeAlias(u,1,-1);
  SetBoundaryType(u,DAGHBoundaryShift);

  INTEGER me = MY_PROC(GH);
  INTEGER num = NUM_PROC(GH);
```

```
  INTEGER Level = 0;
  INTEGER Time = CurrentTime(GH,Level);
  INTEGER TStep = TimeStep(GH,Level);

  DOUBLE dt, dx, dy, tzero = 0.0;
  const DOUBLE dtfac = 0.2;
  const DOUBLE sqrt3 = 1.7321;

  dx = 1.0/(nx - 1.0);
  dy = 1.0/(ny - 1.0);
  dt = dtfac *dx / sqrt3;

  forall(u,Time,Level,c)
    f_initial(FA(u(Time,Level,c,DAGH_Main)),&tzero,&dx,&dy);
  end_forall

  INTEGER currentiter = 0;

  cout << me << ": Iteration: " << currentiter
           << " Max: " << MaxVal (u,Time,Level,DAGH_Main)
           << " Min: " << MinVal (u,Time,Level,DAGH_Main)
           << " Norm: " << Norm2 (u,Time,Level,DAGH_Main)
           << endl;

// Remove the comments, if you have xgraph running.
/*
  BBox viewbbX (2, 0, ny/2, nx-1, ny/2, 1);
  BBox viewbbY (2, nx/2, 0, nx/2, ny-1, 1);

  View (u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
        H3eXGRAPH, H3eDISPLAY, DAGH_Main);
  View (u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
        H3eXGRAPH, H3eDISPLAY, DAGH_Main);
*/

  for (currentiter=1; currentiter < iterations; currentiter++) {

      cout << me << ": Iteration " << currentiter
      << endl;

      // The evolved level will be at the new time
      forall(u,Time,Level,c)
        u(Time-TStep,Level,c,DAGH_Main) = u(Time,Level,c,DAGH_Main);
      end_forall

      forall(u,Time,Level,c)
        f_evolveP(FA(u(Time,Level,c,DAGH_Main)),
                  FA(u(Time-TStep,Level,c,DAGH_Main)),
                  &dt,&dx,&dy);
      end_forall

      Sync(u,Time,Level,DAGH_Main);

      forall(u,Time,Level,c)
        f_evolveC(FA(u(Time,Level,c,DAGH_Main)),
                  FA(u(Time-TStep,Level,c,DAGH_Main)),
                  &dt,&dx,&dy);
      end_forall

      Sync(u,Time,Level,DAGH_Main);
```

```
      BoundaryUpdate(u, Time, Level, DAGH_Main);
      cout << me << ": Iteration: " << currentiter
                 << " Max: " << MaxVal (u,Time,Level,DAGH_Main)
                 << " Min: " << MinVal (u,Time,Level,DAGH_Main)
                 << " Norm: " << Norm2 (u,Time,Level,DAGH_Main)
                 << endl;

// Remove the comments if you have xgraph running.
/*
      if (currentiter % 10 == 0) {
        View (u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
              H3eXGRAPH, H3eDISPLAY, DAGH_Main);
        View (u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
              H3eXGRAPH, H3eDISPLAY, DAGH_Main);
        }
*/

  }


  cout << "Bye !" << endl;
}
```

---

tportfortran.h

```
// Prototyping for FORTRAN routine readdata
#define f_readinput FORTRAN_NAME(readinput_, READINPUT, readinput)
extern "C" {
  void f_readinput(INTEGER *,INTEGER *,
                   DOUBLE *, DOUBLE *,
                   DOUBLE *, DOUBLE *,
                   DOUBLE *,
                   INTEGER *, INTEGER *,
                   DOUBLE *, INTEGER *,
                   DOUBLE *,
                   INTEGER *, INTEGER *,
                   INTEGER *);
}

#define f_initial FORTRAN_NAME(initial_, INITIAL, initial)
extern "C" {
  void f_initial(FI(DOUBLE), DOUBLE *, DOUBLE *, DOUBLE *);
}

#define f_evolveC FORTRAN_NAME(evolvec_, EVOLVEC, evolvec)
extern "C" {
  void f_evolveC(FI(DOUBLE), FI(DOUBLE),
                 DOUBLE *, DOUBLE *, DOUBLE *);
}

#define f_evolveP FORTRAN_NAME(evolvep_, EVOLVEP, evolvep)
extern "C" {
  void f_evolveP(FI(DOUBLE), FI(DOUBLE),
                 DOUBLE *, DOUBLE *, DOUBLE *);
}
```

```
#define f_restrict_amr FORTRAN_NAME(restrict2d2_, RESTRICT2D2, restrict2d2)
extern "C"
{
  void f_restrict_amr (FI(DOUBLE), FI(DOUBLE), BI, DOUBLE, INTEGER);
}

#define f_prolong_amr FORTRAN_NAME(prolong2d2_, PROLONG2D2, prolong2d2)
extern "C"
{
  void f_prolong_amr (FI(DOUBLE), FI(DOUBLE), BI, DOUBLE *, INTEGER *);
}

#define f_pythnorm FORTRAN_NAME(pythnorm_, PYTHNORM, pythnorm)
extern "C"
{
  void f_pythnorm(BI, FDI(DOUBLE), FDI(DOUBLE), DOUBLE *);
}
```

```
#helpon
#########################################################################
#                                                                       #
# Makefile for the DAGH applications                                    #
#                                                                       #
# Allowable external targets:                                           #
#                                                                       #
#       - help                  print out this top banner message       #
#       - clean                 remove all files generated by the make  #
#                                                                       #
# Author:  Manish Parashar <    parashar@cs.utexas.edu>                     #
#                                                                       #
#########################################################################
#helpoff

help:
        @awk '/#helpon/, /#helpoff/' Makefile |          \
        egrep -v "(helpon|helpoff)" | more


# Top directory for the DAGH system !!
DAGH_HOME = /u8/mitra/NDAGH

# Include architecture specific makefile
include make.defn

#########################################################################
# Application Makefile !!!                                              #
#########################################################################

.SUFFIXES: .C .c .F .o .a .f .f9 .cpp

# Define application FORTRAN sources and objects
APP_F_SRC = transport.f
APP_F_OBJ = $(APP_F_SRC:.f=.o)

# Define application C++ sources and objects
APP_CC_SRC = tportseq.C
APP_CC_OBJ = $(APP_CC_SRC:.C=.o)
```

```
APPOBJ = $(APP_F_OBJ) $(APP_CC_OBJ)

# Application specific Flags
C++APPFLAGS = -g
CAPPFLAGS = -g
F77APPFLAGS = -g
F90APPFLAGS = -g

# Application specific libraries
APPLIB =

EXEC =          tportseq

$(EXEC):        $(APPOBJ)
                $(RM) $(EXEC)
                $(C++LINK) $(C++FLAGS) -o $@ \
                $(APPOBJ) $(APPLIB) $(LDLIBS)


########################################################################
#                                                                      #
# Clean up the application files, any core files, the compiler created  #
# template repository !                                                #
#                                                                      #
########################################################################

clean:
                $(RM) *.o core $(EXEC)
                $(RM) -r $(REPOSITORY)
```

---

tportseq Output

```
0: Iteration: 0 Max: 0.999838 Min: 1.60381e-28 Norm: 0.37756
0: Iteration 1
0: Iteration: 1 Max: 0.999892 Min: -1.63113e-12 Norm: 0.378644
0: Iteration 2
0: Iteration: 2 Max: 1.00004 Min: -2.6177e-10 Norm: 0.379713
0: Iteration 3
0: Iteration: 3 Max: 1.00025 Min: -3.53729e-09 Norm: 0.38078
0: Iteration 4
0: Iteration: 4 Max: 1.00036 Min: -2.99928e-08 Norm: 0.381843
0: Iteration 5
0: Iteration: 5 Max: 1.00035 Min: -1.89972e-07 Norm: 0.382904
0: Iteration 6
0: Iteration: 6 Max: 0.999794 Min: -6.77409e-07 Norm: 0.383963
0: Iteration 7
0: Iteration: 7 Max: 1.0002 Min: -1.48459e-06 Norm: 0.38502
0: Iteration 8
0: Iteration: 8 Max: 1.00009 Min: -4.43301e-06 Norm: 0.386074
0: Iteration 9
0: Iteration: 9 Max: 1.00086 Min: -7.90516e-06 Norm: 0.387127
0: Iteration 10
0: Iteration: 10 Max: 1.00037 Min: -1.34036e-05 Norm: 0.388178
0: Iteration 11
0: Iteration: 11 Max: 1.00023 Min: -3.08746e-05 Norm: 0.389228
```

```
0: Iteration 12
0: Iteration: 12 Max: 1.00066 Min: -4.11237e-05 Norm: 0.390278
0: Iteration 13
0: Iteration: 13 Max: 1.00089 Min: -6.74204e-05 Norm: 0.391326
0: Iteration 14
0: Iteration: 14 Max: 1.00072 Min: -0.00012189 Norm: 0.392375
0: Iteration 15
0: Iteration: 15 Max: 1.00008 Min: -0.000152613 Norm: 0.393423
0: Iteration 16
0: Iteration: 16 Max: 1.00059 Min: -0.000184307 Norm: 0.394472
0: Iteration 17
0: Iteration: 17 Max: 0.999626 Min: -0.000326237 Norm: 0.395522
0: Iteration 18
0: Iteration: 18 Max: 1.00126 Min: -0.000440295 Norm: 0.396573
0: Iteration 19
0: Iteration: 19 Max: 1.00043 Min: -0.000513856 Norm: 0.397625
0: Iteration 20
0: Iteration: 20 Max: 1.00033 Min: -0.000585009 Norm: 0.398678
0: Iteration 21
0: Iteration: 21 Max: 1.00034 Min: -0.000850148 Norm: 0.399733
0: Iteration 22
0: Iteration: 22 Max: 1.00085 Min: -0.00115768 Norm: 0.40079
0: Iteration 23
0: Iteration: 23 Max: 1.00091 Min: -0.00134723 Norm: 0.401847
0: Iteration 24
0: Iteration: 24 Max: 1.00022 Min: -0.00148882 Norm: 0.402906
0: Iteration 25
0: Iteration: 25 Max: 1.00031 Min: -0.00163139 Norm: 0.403965
0: Iteration 26
0: Iteration: 26 Max: 0.99858 Min: -0.00215691 Norm: 0.405024
0: Iteration 27
0: Iteration: 27 Max: 1.0011 Min: -0.00276501 Norm: 0.406082
0: Iteration 28
0: Iteration: 28 Max: 1.00035 Min: -0.00316275 Norm: 0.407136
0: Iteration 29
0: Iteration: 29 Max: 1.00001 Min: -0.00342366 Norm: 0.408187
0: Iteration 30
0: Iteration: 30 Max: 0.999397 Min: -0.00365088 Norm: 0.40923
0: Iteration 31
0: Iteration: 31 Max: 1.00032 Min: -0.00390608 Norm: 0.410265
0: Iteration 32
0: Iteration: 32 Max: 1.00057 Min: -0.00494788 Norm: 0.411288
0: Iteration 33
0: Iteration: 33 Max: 0.999783 Min: -0.00594481 Norm: 0.412295
0: Iteration 34
0: Iteration: 34 Max: 0.999453 Min: -0.00658501 Norm: 0.413283
0: Iteration 35
0: Iteration: 35 Max: 0.997667 Min: -0.00701968 Norm: 0.414247
0: Iteration 36
0: Iteration: 36 Max: 1.00046 Min: -0.00739178 Norm: 0.415182
0: Iteration 37
0: Iteration: 37 Max: 0.999749 Min: -0.00775585 Norm: 0.416083
0: Iteration 38
0: Iteration: 38 Max: 0.999152 Min: -0.00834066 Norm: 0.416943
0: Iteration 39
0: Iteration: 39 Max: 0.99793 Min: -0.0101194 Norm: 0.417757
0: Iteration 40
0: Iteration: 40 Max: 0.999382 Min: -0.0114227 Norm: 0.418516
0: Iteration 41
0: Iteration: 41 Max: 0.999767 Min: -0.0122665 Norm: 0.419214
0: Iteration 42
```

```
0: Iteration: 42 Max: 0.998865 Min: -0.0128286 Norm: 0.419842
0: Iteration 43
0: Iteration: 43 Max: 1.00527 Min: -0.0133079 Norm: 0.420391
0: Iteration 44
0: Iteration: 44 Max: 1.01845 Min: -0.0137234 Norm: 0.420853
0: Iteration 45
0: Iteration: 45 Max: 1.02976 Min: -0.0141922 Norm: 0.421219
0: Iteration 46
0: Iteration: 46 Max: 1.03902 Min: -0.0162226 Norm: 0.421478
0: Iteration 47
0: Iteration: 47 Max: 1.0461 Min: -0.0181308 Norm: 0.421621
0: Iteration 48
0: Iteration: 48 Max: 1.05094 Min: -0.0194847 Norm: 0.421638
0: Iteration 49
0: Iteration: 49 Max: 1.05367 Min: -0.0203235 Norm: 0.42152
0: Iteration 50
0: Iteration: 50 Max: 1.05482 Min: -0.020917 Norm: 0.421256
0: Iteration 51
0: Iteration: 51 Max: 1.05549 Min: -0.02144 Norm: 0.420836
0: Iteration 52
0: Iteration: 52 Max: 1.05566 Min: -0.0219078 Norm: 0.420252
0: Iteration 53
0: Iteration: 53 Max: 1.05604 Min: -0.023547 Norm: 0.419495
0: Iteration 54
0: Iteration: 54 Max: 1.0566 Min: -0.0269177 Norm: 0.418554
0: Iteration 55
0: Iteration: 55 Max: 1.05629 Min: -0.0294586 Norm: 0.417423
0: Iteration 56
0: Iteration: 56 Max: 1.05656 Min: -0.0313051 Norm: 0.416094
0: Iteration 57
0: Iteration: 57 Max: 1.05682 Min: -0.0325164 Norm: 0.414561
0: Iteration 58
0: Iteration: 58 Max: 1.05636 Min: -0.0333609 Norm: 0.412816
0: Iteration 59
0: Iteration: 59 Max: 1.05698 Min: -0.0340333 Norm: 0.410856
0: Iteration 60
0: Iteration: 60 Max: 1.05646 Min: -0.0346867 Norm: 0.408676
0: Iteration 61
0: Iteration: 61 Max: 1.05678 Min: -0.0354413 Norm: 0.406273
0: Iteration 62
0: Iteration: 62 Max: 1.05691 Min: -0.0360333 Norm: 0.403646
0: Iteration 63
0: Iteration: 63 Max: 1.05584 Min: -0.0366247 Norm: 0.400794
0: Iteration 64
0: Iteration: 64 Max: 1.05737 Min: -0.0374171 Norm: 0.397717
Bye !
```

---

tportpar.C

```
//-------------------------------------------------------------------
//
// DAGH Driver for the Transport Application
// Unigrid Parallel Operation
//
//-------------------------------------------------------------------
```

```
#include <DAGH.h>
#include "tportfortran.h"

// If you have xgraph running on your system insert the path name
// to its directory. Redefine H3eDISPLAY for the machine you are
// running your program. Otherwise leave the comments in place.
/*
#define H3eXGRAPH "/u5/parashar/xgraph/xgraph"
#define H3eDISPLAY "tardis.ticam.utexas.edu:0.0"
*/

void main(INTEGER argc, CHARACTER *argv[])
{
#ifndef DAGH_NO_MPI
   MPI_Init (&argc, &argv);
#endif

  const INTEGER nx = 32;
  const INTEGER ny = 32;

  const INTEGER iterations = 2*nx;

  // Set up the Grid Structure
  INTEGER shape[2];
  DOUBLE bb[2*2];

  shape[0] = nx;
  bb[0] = 0.0;
  bb[1] = 1.0;

  shape[1] = ny;
  bb[2] = 0.0;
  bb[3] = 1.0;

  GridHierarchy GH(2,DAGHVertexCentered,1);
  SetBaseGrid (GH,bb,shape);
  SetBoundaryWidth(GH,1);
  ComposeHierarchy(GH);

  // Setup the GridFunctions....
  INTEGER t_sten = 1, s_sten = 1;
  GridFunction(2)<DOUBLE> u("u",t_sten,s_sten,GH);
  SetTimeAlias(u,1,-1);
  SetBoundaryType(u,DAGHBoundaryShift);

  INTEGER me = MY_PROC(GH);
  INTEGER num = NUM_PROC(GH);

  INTEGER Level = 0;
  INTEGER Time = CurrentTime(GH,Level);
  INTEGER TStep = TimeStep(GH,Level);

  DOUBLE dt, dx, dy, tzero = 0.0;
  const DOUBLE dtfac = 0.2;
  const DOUBLE sqrt3 = 1.7321;

  dx = 1.0/(nx - 1.0);
  dy = 1.0/(ny - 1.0);
  dt = dtfac *dx / sqrt3;

  forall(u,Time,Level,c)
```

```
    f_initial(FA(u(Time,Level,c,DAGH_Main)),&tzero,&dx,&dy);
  end_forall

  INTEGER currentiter = 0;

  cout << me << ": Iteration: " << currentiter
            << " Max: " << MaxVal (u,Time,Level,DAGH_Main)
            << " Min: " << MinVal (u,Time,Level,DAGH_Main)
            << " Norm: " << Norm2 (u,Time,Level,DAGH_Main)
            << endl;

// Remove the comments if you have xgraph running
/*
  BBox viewbbX (2, 0, ny/2, nx, ny/2, 1);
  BBox viewbbY (2, nx/2, 0, nx/2, ny, 1);

  View (u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
        H3eXGRAPH, H3eDISPLAY, DAGH_Main);
  View (u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
        H3eXGRAPH, H3eDISPLAY, DAGH_Main);
*/

  for (currentiter = 1; currentiter <= iterations; currentiter++) {

      cout << me << ": Iteration " << currentiter
      << endl;

      // The evolved level will be at the new time
      forall(u,Time,Level,c)
        u(Time-TStep,Level,c,DAGH_Main) = u(Time,Level,c,DAGH_Main);
      end_forall

      forall(u,Time,Level,c)
        f_evolveP(FA(u(Time,Level,c,DAGH_Main)),
                  FA(u(Time-TStep,Level,c,DAGH_Main)),
                  &dt,&dx,&dy);
      end_forall

      Sync(u,Time,Level,DAGH_Main);

      forall(u,Time,Level,c)
        f_evolveC(FA(u(Time,Level,c,DAGH_Main)),
                  FA(u(Time-TStep,Level,c,DAGH_Main)),
                  &dt,&dx,&dy);
      end_forall

      Sync(u,Time,Level,DAGH_Main);

      BoundaryUpdate(u, Time, Level, DAGH_Main);

      cout << me << ": Iteration: " << currentiter
                << " Max: " << MaxVal (u,Time,Level,DAGH_Main)
                << " Min: " << MinVal (u,Time,Level,DAGH_Main)
                << " Norm: " << Norm2 (u,Time,Level,DAGH_Main)
                << endl;

// Remove the comments if you have xgraph running.
/*
      if (currentiter % 10 == 0) {
        View (u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
              H3eXGRAPH, H3eDISPLAY, DAGH_Main);
```

```
        View (u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
              H3eXGRAPH, H3eDISPLAY, DAGH_Main);
        }
*/
  }

#ifndef DAGH_NO_MPI
   DAGHMPI_Finalize();
#endif

  cout << "Bye !" << endl;
}
```

---

tportamr.C

```
//----------------------------------------------------------------------
//----------------------------------------------------------------------
// AMR Driver for the transport equation
//----------------------------------------------------------------------
//----------------------------------------------------------------------

//----------------------------------------------------------------------
// Flags
//----------------------------------------------------------------------
#define VizServer 0     // Who will write to stdout

// Remove the comments if you have XGRAPH running.
/*
#define H3eXGRAPH  "/u5/parashar/xgraph/xgraph"
#define H3eDISPLAY "telluride.ticam.utexas.edu:0.0"
*/

//----------------------------------------------------------------------
// DAGH includes
//----------------------------------------------------------------------
#include "DAGH.h"
#include "DAGHCluster.h"

//----------------------------------------------------------------------
// Application Includes
//----------------------------------------------------------------------
#include "tportamr.h"
#include "tportfortran.h"

//----------------------------------------------------------------------
// AMR parameters
//----------------------------------------------------------------------
INTEGER MaxLev = 1;             // Maximum level of refinement
INTEGER BufferWidth = 1;        // Buffer width used by clusterer
INTEGER BlockWidth = 1;         // Minimum granularity used by clusterer
DOUBLE MinEfficiency = 0.7;     // Minumum efficiency of clustering
GFTYPE Thresh = 0.0;            // Truncation error threshold for regriding
INTEGER RegridEvery = 4;        // How oftern do I regrid ?
INTEGER NumIters = 0;           // Number of timesteps on the base grid
INTEGER OutEvery = 0;           //
```

```
void main(INTEGER argc, CHARACTER *argv[])
{

//-----------------------------------------------------------------
// If I using MPI initialize it...
//-----------------------------------------------------------------
  cout << "Initializing MPI\n";
  MPI_Init( &argc, &argv ); // Initialize MPI


//-----------------------------------------------------------------
// Local variable declarations
//-----------------------------------------------------------------
  INTEGER  nx, ny;
  DOUBLE  dx, dy, dt;                    // Grid spacings
  DOUBLE xmin, xmax, ymin, ymax;         // Grid extent
  DOUBLE cfl;                            // Initial data parameters

//-----------------------------------------------------------------
// Read grid properties and initial data parameters from file
//-----------------------------------------------------------------
   INTEGER niters = 0;
   INTEGER ml, regride, buffw, blkw, outevery;
   GFTYPE thresh = 0;
   DOUBLE mineff = 0;
   INTEGER d = 0;

   cout << "****Reading in parameters****" << endl << flush;
   f_readinput(&nx, &ny,     // Number of grid points
               &xmin, &xmax, // X Coords range
               &ymin, &ymax, // Y Coords range
               &cfl,         // Courant factor
               &niters,      // Number of iterations
               &ml,          // Max levels
               &thresh,      // Threshold
               &regride,     // Regrid every
               &mineff,      // Min efficiency
               &buffw,       // Buffer width
               &blkw,        // Min block width
               &outevery);   // Output every time step

   MaxLev = ml;
   RegridEvery = regride;
   BufferWidth = buffw;
   BlockWidth = blkw;
   MinEfficiency = mineff;
   Thresh = thresh;
   NumIters = niters;
   OutEvery = outevery;

//-----------------------------------------------------------------
// Calculate dx, dy and dt from inputs
//-----------------------------------------------------------------
  dx  = (xmax - xmin) / (DOUBLE)(nx-1);
  dy  = (ymax - ymin) / (DOUBLE)(ny-1);
  dt = cfl*dx;

//-----------------------------------------------------------------
// Setup 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,2);
   SetBoundaryWidth(GH,1);
   SetBoundaryType(GH,DAGHNoBoundary);

   ComposeHierarchy(GH);

//-----------------------------------------------------------------
// Processor information
//-----------------------------------------------------------------
  INTEGER me = MY_PROC(GH);
  INTEGER num = NUM_PROC(GH);


//-----------------------------------------------------------------
//  Declare grid functions
//-----------------------------------------------------------------
   INTEGER  t_sten = 1;
   INTEGER  s_sten = 1;

   GridFunction(DIM)<GFTYPE> u("u", t_sten, s_sten, GH,
                               DAGHCommFaceOnly, DAGHHasShadow);
   SetTimeAlias(u,1,-1);
   SetBoundaryType(u,DAGHBoundaryShift);

   GridFunction(DIM)<GFTYPE> temp("temp", t_sten, s_sten, GH,
                                  DAGHCommFaceOnly, DAGHHasShadow);
   SetTimeAlias(temp,1,-1);

   foreachGF(GH,GF,DIM,GFTYPE)
     SetProlongFunction(GF, (void *) &f_prolong_amr);
     SetRestrictFunction(GF, (void *) &f_restrict_amr);
   end_foreachGF

//-----------------------------------------------------------------
// Initialize Level = 0 (Base Grid)
//-----------------------------------------------------------------
  INTEGER Level = 0;
  INTEGER Time = CurrentTime(GH,Level,DAGH_Main); // Current Time at Level
  INTEGER TStep = TimeStep(GH,Level,DAGH_Main);   // Time Step at Level

//-----------------------------------------------------------------
// Set up initial data on previous step (Main & Shadow)
//-----------------------------------------------------------------
  GFTYPE tzero = 0.0;

  forall(u,Time,Level,c)
    f_initial(FA(u(Time,Level,c,DAGH_Main)),&tzero,&dx,&dy);
  end_forall
```

```
  forall(u,Time,Level,c)
    f_initial(FA(u(Time,Level,c,DAGH_Shadow)),&tzero,&dx,&dy);
  end_forall


//-----------------------------------------------------------------
// Dump out the initial data
//-----------------------------------------------------------------

// Remove the comments if you are using XGRAPH
/*
  const int s = StepSize(GH,Level,DAGH_Main);
  const BBox gbb = GH.glbbbox();
  const int snx = gbb.upper(0);
  const int sny = gbb.upper(1);
  BBox viewbbX(2,0,sny/2,snx,sny/2,s);
  BBox viewbbY(2,snx/2,0,snx/2,sny,s);

  View(u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
       H3eXGRAPH,H3eDISPLAY,DAGH_Main);
  View(u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
       H3eXGRAPH,H3eDISPLAY,DAGH_Main);
*/

//-----------------------------------------------------------------
// Begin Evolution
//-----------------------------------------------------------------
  if (me == VizServer)
    cout << "Starting main loop" << endl << flush;

  INTEGER currentiter = 0;

  for (currentiter=1; currentiter < NumIters; currentiter++) {

      bno_RecursiveIntegrate(GH, Level, u, temp, dt, cfl);

      Time = CurrentTime(GH,Level,DAGH_Main); // Current Time at Level
      TStep = TimeStep(GH,Level,DAGH_Main);   // Time Step at Level

      cout << me << " -> Iteration: " << currentiter
           << " MaxVal: " << MaxVal(u,Time,Level,DAGH_Main)
           << " MinVal: " << MinVal(u,Time,Level,DAGH_Main)
           << endl;

// Remove the comments if you are running XGRAPH
/*
      if (currentiter%10 == 0) {
        View(u,Time,Level,DAGH_X,viewbbX,0,DAGHViz_XGRAPH,
             H3eXGRAPH,H3eDISPLAY,DAGH_Main);
        View(u,Time,Level,DAGH_Y,viewbbY,0,DAGHViz_XGRAPH,
             H3eXGRAPH,H3eDISPLAY,DAGH_Main);
      }
*/

  } // End loop over number of iterations

 cout << "Finalizing MPI\n";
 DAGHMPI_Finalize();

} // End main
```

```
//-----------------------------------------------------------------------
//
// Recursive Integrate
//
//-----------------------------------------------------------------------
void bno_RecursiveIntegrate(
                GridHierarchy &GH,
                INTEGER const Level,
                GridFunction(DIM)<GFTYPE> &u,
                GridFunction(DIM)<GFTYPE> &temp,
                DOUBLE &dt, DOUBLE const &cfl)
{
//----------------------------------------------------------------
// Processor Info
//----------------------------------------------------------------
  INTEGER me = MY_PROC(GH);
  INTEGER num = NUM_PROC(GH);

//----------------------------------------------------------------
// Select number of iterations to run based on current grid level
//----------------------------------------------------------------
  INTEGER NoIterations = 0;
  if (Level==0)
      NoIterations = 1;
  else
     NoIterations = RefineFactor(GH);

//----------------------------------------------------------------
// Take recursive steps..
//----------------------------------------------------------------
  for (INTEGER i = 0; i < NoIterations; i++) {

    INTEGER Time = CurrentTime(GH,Level,DAGH_Main);
    INTEGER TStep = TimeStep(GH,Level,DAGH_Main);

//----------------------------------------------------------------
// Is it regridding time?
//----------------------------------------------------------------
    if (Level < MaxLevel(GH) &&
        StepsTaken(GH,Level,DAGH_Main)%RegridEvery == 0 &&
        StepsTaken(GH,Level,DAGH_Main) != 0) {

       // Truncation error estimation using DAGH_Main
       wave_trunc_est(GH,Time,Level,u,temp);

       // Regrid using truncation error estimate
       bno_RegridSystemAbove(GH,temp,Level,Thresh);

    } // End if statement Level...

//----------------------------------------------------------------
// Take a step on the DAGH_Main hierarchy
//----------------------------------------------------------------
    DOUBLE dagh_dx = DeltaX(GH,DAGH_X,Level,DAGH_Main);
    DOUBLE dagh_dy = DeltaX(GH,DAGH_Y,Level,DAGH_Main);
    DOUBLE dagh_dt = dagh_dx * cfl;

    forall(u,Time,Level,c)
      f_evolveP(FA(u(Time+TStep,Level,c,DAGH_Main)),
                FA(u(Time,Level,c,DAGH_Main)),
```

```
                &dagh_dt,&dagh_dx,&dagh_dy);
    end_forall

    Sync(u,Time,Level,DAGH_Main);

    forall(u,Time,Level,c)
      f_evolveC(FA(u(Time+TStep,Level,c,DAGH_Main)),
                FA(u(Time,Level,c,DAGH_Main)),
                &dagh_dt,&dagh_dx,&dagh_dy);
    end_forall

    Sync(u,Time,Level,DAGH_Main);

    BoundaryUpdate(u,Time,Level,DAGH_Main);

    Sync(u,Time,Level,DAGH_Main);

//------------------------------------------------------------------
// Take a step on the DAGH_Shadow hierarchy
//------------------------------------------------------------------
    if (StepsTaken(GH,Level,DAGH_Main) % RefineFactor(GH) == 0){

      dagh_dx = DeltaX(GH,DAGH_X,Level,DAGH_Shadow);
      dagh_dy = DeltaX(GH,DAGH_Y,Level,DAGH_Shadow);
      dagh_dt = dagh_dx * cfl;

       // First update DAGH_Shadow from DAGH_Main
       forall(u,Time,Level,c)
         u(Time,Level,c,DAGH_Shadow) = u(Time,Level,c,DAGH_Main);
       end_forall

       Sync(u,Time,Level,DAGH_Shadow);

       forall(u,Time,Level,c)
         f_evolveP(FA(u(Time+TStep,Level,c,DAGH_Shadow)),
                   FA(u(Time,Level,c,DAGH_Shadow)),
                   &dagh_dt,&dagh_dx,&dagh_dy);
       end_forall

       Sync(u,Time,Level,DAGH_Shadow);

       forall(u,Time,Level,c)
         f_evolveC(FA(u(Time+TStep,Level,c,DAGH_Shadow)),
                   FA(u(Time,Level,c,DAGH_Shadow)),
                   &dagh_dt,&dagh_dx,&dagh_dy);
       end_forall

       Sync(u,Time,Level,DAGH_Shadow);

       BoundaryUpdate(u,Time,Level,DAGH_Shadow);

       Sync(u,Time,Level,DAGH_Shadow);

    } // Ends if statement for DAGH_Shadow

//------------------------------------------------------------------
// If we are not at the finest level then go to next level
//------------------------------------------------------------------
    if (Level < FineLevel(GH)) {

       // Recursive Integration of next level in hierarchy
```

```
       bno_RecursiveIntegrate( GH, Level+1, u, temp, dt, cfl );

    } // End if (Level < FineLevel(GH))

//-----------------------------------------------------------------
// Move from current time to previous.
//-----------------------------------------------------------------
    CycleTimeLevels(u,Level);

//-----------------------------------------------------------------
// Increment time step on current level
//-----------------------------------------------------------------
    IncrCurrentTime(GH,Level,DAGH_Main);

//-----------------------------------------------------------------
// Restriction on DAGH_Main
//-----------------------------------------------------------------
    if (Level < FineLevel(GH)) {
      Time = CurrentTime(GH,Level,DAGH_Main);
      INTEGER myargc = 0; GFTYPE myargs[1];
      Restrict(u, Time, Level+1, Time, Level, myargs, myargc, DAGH_Main);
      Sync(u, Time, Level, DAGH_Main);
    } // end if statement for level

  } // End For loop

} // End bnoRecursiveIntegrate




//-----------------------------------------------------------------
// Regrid GridHierarchy at Level
//-----------------------------------------------------------------
void bno_RegridSystemAbove(GridHierarchy &GH,
                           GridFunction(DIM)<GFTYPE>amp; u,
                           INTEGER const Level, GFTYPE const thresh)
{
  INTEGER me = MY_PROC(GH);
  INTEGER num = NUM_PROC(GH);

//-----------------------------------------------------------------
// Compute levels to regrid at - from the finest down to Level
//-----------------------------------------------------------------
  INTEGER flev = FineLevel(GH);
  INTEGER lev = 0;
  if (flev == 0) lev = 0;
  else if (flev == MaxLev-1) lev = flev-1;
  else lev = flev;

//-----------------------------------------------------------------
// Cluster at refine
//-----------------------------------------------------------------
  BBoxList bblist;

  for ( ; lev > Level; lev--) {
    INTEGER Time = CurrentTime(GH,lev,DAGH_Main);
    INTEGER TStep = TimeStep(GH,lev,DAGH_Main);

    DAGHCluster2d(u, Time, lev,
                  BlockWidth, BufferWidth,
```

```
                   MinEfficiency, thresh,
                   bblist, bblist, DAGH_Main);

    Refine(GH,bblist,lev);

  } // Ends for loop over levels

//------------------------------------------------------------------
// Recompose the GridHierarchy
//------------------------------------------------------------------
  RecomposeHierarchy(GH);

} // bno_RegridSystemAbove
```

---

grid.f

```
c-------------------------------------------------------------------------
c grid.f
c
c Grid-to-Grid operations for AMR
c  - Prolongation operators
c  - Restriction operators
c-------------------------------------------------------------------------
c  Routine to convert from global array coordinates to local array coordinates
      integer function getindx(loc, lb, stride )
         implicit none

         integer   loc, lb, stride

         getindx = (loc - lb)/stride + 1
         return
      end
c-------------------------------------------------------------------------
c Prolongation operator: prolong2d2 (2D 2nd order)
c  USES INTEGER Lower and Upper bounds for use with DAGH
c
c  - Uses linear interpolation where necessary
c Interface:
c   shapef(2) := shape of fine grid function
c   shapec(2) := shape of coarse grid function
c
c   uf(,) := fine grid function
c   uc(,) := coarse grid function
c
c   lbc(2) := lower bound for coarse grid
c   ubc(2) := upper bound for coarse grid
c   lbf(2) := lower bound for fine grid
c   ubf(2) := upper bound for fine grid
c   lbr(2) := lower bound for region prolongation desired
c   ufr(2) := upper bound for region prolongation desired

      subroutine prolong2d2(
     .                                 uc, lbc, ubc, shapec,
     .                                 uf, lbf, ubf, shapef,
```

```
     .                                lbr, ubr, args, argc)
        implicit none

        integer   shapef(2), shapec(2)

        real*8   uf(shapef(1), shapef(2)),
     .           uc(shapec(1), shapec(2))

        integer  lbf(2), ubf(2),
     .           lbc(2), ubc(2),
     .           lbr(2), ubr(2),
     .           getindx


                     integer argc
                     real*8 args(argc)

c      Local variables

        integer   i, j, ic,jc,
     .           ilbf(2), iubf(2),
     .           ilbc(2), iubc(2),
     .           refine,
     .           ifine, icoarse,
     .           jfine, jcoarse, stridec, stridef

        real*8   hf, hc, rem(2), x, y,
     .           a11, a12, a21, a22



c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
c         Get strides coarse grid
c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        stridec = (ubc(1) - lbc(1))/(shapec(1)-1)
        stridef = (ubf(1) - lbf(1))/(shapef(1)-1)
        refine = stridec/stridef

c       write(45,*)'In prolong: stridec=',stridec,' stridef=',stridef
c       write(46,*)'In prolong: stridec=',
c     .              (ubc(1) - lbc(1))/(shapec(1)-1),
c     .                        ' stridef=',
c     .              (ubf(1) - lbf(1))/(shapef(1)-1)

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
c      Prolongation region is defined on the domain of the fine grid.
c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
c      Loop through points in prolongation region

c        write(55,*)shapec(1), shapec(2)
c        write(55,*)shapef(1), shapef(2)
c        write(55,*)'stride coarse ',stridec
c        write(55,*)'stride fine ',stridef
c        write(55,*)'Region l/u x l/u y-----------------------'
c        write(55,*)lbr(1), ubr(1),lbr(2), ubr(2)
c        write(55,*)'Coarse l/u x l/u y-----------------------'
c        write(55,*)lbc(1), ubc(1),lbc(2), ubc(2)
c        write(55,*)'Fine l/u x l/u y-----------------------'
c        write(55,*)lbf(1), ubf(1),lbf(2), ubf(2)
c        write(55,*)'*******************'
```

```
         do i = lbr(1), ubr(1), stridef
           do j = lbr(2), ubr(2), stridef

c             calculate coarse grid integer coordinates difference
            ic = i - lbc(1)
            jc = j - lbc(2)

            ifine = getindx(i, lbf(1), stridef)
            jfine = getindx(j, lbf(2), stridef)

            icoarse = getindx(i, lbc(1), stridec)
            jcoarse = getindx(j, lbc(2), stridec)

c            if(trace)then
c            write(6,*)icoarse, jcoarse, ifine, jfine
c            write(6,*)ic, jc, mod(ic,refine), mod(jc,refine)
c            write(6,*)'**********'
c            end if

c            if(trace)then
c            write(6,*)i,j,ic,jc
c            write(6,*)'m ',mod(i-1,refine),mod(j-1,refine)
c            write(6,*)'********************************** '
c            end if

            if(mod(ic,stridec) .eq. 0 .and. mod(jc,stridec) .eq.0)then
               uf(ifine,jfine) = uc(icoarse,jcoarse)
            end if

            if(mod(ic,stridec) .eq. 0 .and. mod(jc,stridec) .ne.0)then
               uf(ifine,jfine) = 0.5 * uc(icoarse,jcoarse) +
     .                           0.5 * uc(icoarse,jcoarse+1)
            end if

            if(mod(ic,stridec) .ne. 0 .and. mod(jc,stridec) .eq.0)then
               uf(ifine,jfine) = 0.5 * uc(icoarse,jcoarse) +
     .                           0.5 * uc(icoarse+1,jcoarse)
            end if

            if(mod(ic,stridec) .ne. 0 .and. mod(jc,stridec).ne. 0)then
               uf(ifine,jfine) = 0.25 * uc(icoarse,jcoarse) +
     .                           0.25 * uc(icoarse+1,jcoarse) +
     .                           0.25 * uc(icoarse,jcoarse+1) +
     .                           0.25 * uc(icoarse+1,jcoarse+1)
            end if

           end do
         end do

         return
      end
c-----------------------------------------------------------------------
c Restriction operator: restrict2d (pure injections)
c Interface:
c   shapef(2) := shape of fine grid function
c   shapec(2) := shape of coarse grid function
c
c   uf(,) := fine grid function
c   uc(,) := coarse grid function
c
c   lbc(2) := lower bound for coarse grid
```

```
c   ubc(2) := upper bound for coarse grid
c   lbf(2) := lower bound for fine grid
c   ubf(2) := upper bound for fine grid
c   lbr(2) := lower bound for region restriction desired
c   ufr(2) := upper bound for region restriction desired

      subroutine restrict2d2(uf, lbf, ubf, shapef,
     .                              uc, lbc, ubc, shapec,
     .                              lbr, ubr, args, argc)
         implicit none

         integer   shapef(2), shapec(2)

         real*8   uf(shapef(1), shapef(2)),
     .            uc(shapec(1), shapec(2))

         integer  lbf(2), ubf(2),
     .            lbc(2), ubc(2),
     .            lbr(2), ubr(2)

         integer argc
         real*8 args(argc)

c      Local variables

         integer   i, j, imin, imax, jmin, jmax,
     .            ifine, icoarse,
     .            jfine, jcoarse, refine, stridec, stridef,
     .            ilbc(2), iubc(2), getindx

         real*8   hf, hc, x, y

         stridec = (ubc(1) - lbc(1))/(shapec(1)-1)
         stridef = (ubf(1) - lbf(1))/(shapef(1)-1)
         refine = stridec/stridef

c - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
c     Find coarse domain over which to refine
c     Take three regions and select out intersection

          imin = max(lbf(1), lbc(1), lbr(1))
          imax = min(ubf(1), ubc(1), ubr(1))

          jmin = max(lbf(2), lbc(2), lbr(2))
          jmax = min(ubf(2), ubc(2), ubr(2))

          if (mod(imin-lbc(1),stridec) .ne. 0) then
             imin = imin + mod(imin-lbc(1),stridec)
          endif
          if (mod(jmin-lbc(2),stridec) .ne. 0) then
             jmin = jmin + mod(jmin-lbc(2),stridec)
          endif

c      Inject points to coarse grid from fine grid
c     Loop from lower bound to upper bound with stride of refine.
c     Convert the integer coordinates to fine and coarse grid absolute
c     coordinates...
         do i = imin, imax,stridec
            do j = jmin, jmax,stridec

               ifine = getindx(i, lbf(1), stridef)
```

```
              jfine = getindx(j, lbf(2), stridef)

              icoarse = getindx(i, lbc(1), stridec)
              jcoarse = getindx(j, lbc(2), stridec)

          if(icoarse .gt. shapec(1) .or. jcoarse .gt. shapec(2))then
               write(0,*)'ERROR in restriction: ',icoarse,jcoarse
          end if
              uc(icoarse,jcoarse) = uf(ifine,jfine)

           end do
        end do


        return
      end
c-----------------------------------------------------------------------
```

Cluster1.C

```
/*
file      Cluster1.C
date      Wed Apr 17 19:04:31 1996
author    Paul Walker
desc      A 1D "clustering" algorithm. 1D clustering is really easy.
          So this is really short.
          Clusters a grid function flag into a list of bounding
          boxes result. Assume flag is a 1D real array
          with values of 0.0 or 1.0 for flag or no flag.
*/

#include "DAGH.h"

void Cluster1(GridData(1)<short> &flag,
              double Efficiency,
              int MinWidth,
              int BufferWidth,
              BBoxList &result) {
  GridData(1)<short> newflag(flag.bbox());
  int l=-1, u=-1;

  /* First buffer! */
  newflag = 0;

  for_1(i,flag.bbox(),flag.bbox().stepsize())
    if (flag(i)) {
      BBox where(1,i,i,flag.bbox().stepsize(0));
      where.grow(BufferWidth);
      newflag.equals(1.0, where);
    }
  end_for;

  /* Now just step along looking for zero crossings or end of
     the thing.  This defines our BBox */
  if (newflag(newflag.bbox().lower(0))) {
```

```
    l = newflag.bbox().lower(0);
  }
  int i;
  for (i=newflag.bbox().lower(0)+newflag.bbox().stepsize(0);
       i < newflag.bbox().upper(0);
       i+= newflag.bbox().stepsize(0)) {
    if (newflag(i) && !newflag(i-newflag.bbox().stepsize(0))) {
      l = i;
    }
    if (!newflag(i) && newflag(i-newflag.bbox().stepsize(0))) {
      u = i-newflag.bbox().stepsize(0);
      if (u-l > MinWidth) {
        BBox add(1,l,u,newflag.bbox().stepsize(0));
        result.add(add);
      }
    }
  }
  if (u < l) {
    u = newflag.bbox().upper(0);
    BBox add(1,l,u,newflag.bbox().stepsize(0));
    result.add(add);
  }
}
```

CLuster3.C

```
#include "DAGH.h"

/*
file      Cluster3.C
date      Sat Mar 23 17:37:08 1996
author    Paul Walker
desc      A 3D Clusterer using signature clustering.
          There is also a 2-D interface
*/

/*
routine    Cluster2
date       Wed Apr 17 20:48:29 1996
author     Paul Walker
desc       A 2D interface to the recursive clustering algorithm.
*/

#include "DAGH.h"

void Cluster2(GridData(2)<short> &flag,
              double Efficiency,
              int MinWidth,
              int BufferWidth,
              BBoxList &Result) {
  BBoxList Result3D;
  BBox flat_3d(3,flag.bbox().lower(0),flag.bbox().lower(1),0,
               flag.bbox().upper(0),flag.bbox().upper(1),0,
               flag.bbox().stepsize(0),flag.bbox().stepsize(1),1);
  GridData(2)<short> newflag(flag.bbox());
```

```
  GridData(3)<short> clust_this(flat_3d);
  void Cluster3R(GridData(3)<short> &flag, double, int,
  BBoxList&);
  int flagged_points = 0;
  newflag = 0; clust_this = 0;

  BBox buffer_bbox = flag.bbox();
  buffer_bbox.grow(-BufferWidth);
  for_2(i, j, buffer_bbox, flag.stepsize())
    if (flag(i,j)) {
      flagged_points ++;
      BBox where(2,i,j,i,j,
                 flag.bbox().stepsize(0),
                 flag.bbox().stepsize(1));
      where.grow(BufferWidth);
      newflag.equals(1,where);
    }
  end_for;

  if (flagged_points) {
    for_2(i,j,flag.bbox(),flag.stepsize())
      clust_this(i,j,0) = newflag(i,j);
    end_for;
    Cluster3R(clust_this, Efficiency, MinWidth, Result3D);
  }

  /*
    Make sure that rank is appropriately set for the bboxes in
    Result!!

    Manish Parashar Thu May  9 07:54:32 CDT 1996
    */
  for (BBox* tmpbb=Result3D.first();tmpbb;tmpbb=Result3D.next()) {
    BBox tmp2d(2,tmpbb->lower(),tmpbb->upper(), tmpbb->stepsize());
    Result.add(tmp2d);
  }
}

/*
   routine    Cluster3
   date       Wed Apr 17 20:48:29 1996
   author     Paul Walker
   desc
   A 3D interface to the recursive clustering algorithm.
*/

void Cluster3(GridData(3)<short> &flag,
                  double Efficiency,
                  int MinWidth,
                  int BufferWidth,
                  BBoxList &Result) {
  BBoxList tmpresult, result;
  GridData(3)<short> newflag(flag.bbox());
  void Cluster3R(GridData(3)<short> &flag, double, int,
  BBoxList&);
  int flagged_points = 0;
  newflag = 0;

  /* Buffer */
  BBox buffer_bbox = flag.bbox();
  buffer_bbox.grow(-BufferWidth);
```

```
  for_3(i, j, k, buffer_bbox, flag.stepsize())
    if (flag(i,j,k)) {
      flagged_points ++;
      BBox where(3,i,j,k,i,j,k,
                 flag.bbox().stepsize(0),
                 flag.bbox().stepsize(1),
                 flag.bbox().stepsize(2));
      where.grow(BufferWidth);
      newflag.equals(1,where);
    }
  end_for;

  if (flagged_points) {
    Cluster3R(newflag, Efficiency, MinWidth, Result);
  }
}

/*
   routine    Cluster3R
   date       Sat Mar 23 18:35:36 1996
   author     Paul Walker
   desc
   A recursive signature clustering routine.
   calls   Cluster_Prune Cluster_Slice
*/

void Cluster3R(GridData(3)<short> &flag,
               double Efficiency,
               int MinWidth,
               BBoxList &recurseOnThis) {
  void Cluster3R(GridData(3)<short> &flag, double, int, BBoxList&);
  void Cluster_Prune(GridData(1)<int> &sig, int *, int *, int);
  void Cluster_Slice(GridData(1)<int> &sig, int *, int *, int);

  int BiggestExtent = MinWidth;
  BBoxList result, tmpresult;

  int l[3], u[3], s[3];

  int i;
  for (i=0; i < 3; i++) s[i] = flag.bbox().stepsize(i);

  //  cout << "Clustering " << flag.bbox() <<"\n";
  //  cout.flush();

  /* Declare and fill in Signature Planes in each direction */
  Array(1)<int> i_sig(flag.bbox().lower(0),
                      flag.bbox().upper(0),
                      flag.bbox().stepsize(0));
  Array(1)<int> j_sig(flag.bbox().lower(1),
                      flag.bbox().upper(1),
                      flag.bbox().stepsize(1));
  Array(1)<int> k_sig(flag.bbox().lower(2),
                      flag.bbox().upper(2),
                      flag.bbox().stepsize(2));
  i_sig = 0;
  j_sig = 0;
  k_sig = 0;
  for_3(ii, jj, kk, flag.bbox(), flag.stepsize())
    i_sig(ii) += flag(ii,jj,kk);
    j_sig(jj) += flag(ii,jj,kk);
```

```
  k_sig(kk) += flag(ii,jj,kk);
end_for;

/* Check for pruning */
int ipl, ipu, jpl, jpu, kpl, kpu;
Cluster_Prune(i_sig,&ipl,&ipu,BiggestExtent*flag.bbox().stepsize(0));
Cluster_Prune(j_sig,&jpl,&jpu,BiggestExtent*flag.bbox().stepsize(1));
Cluster_Prune(k_sig,&kpl,&kpu,BiggestExtent*flag.bbox().stepsize(2));

Coords pl(3,ipl,jpl,kpl);
Coords pu(3,ipu,jpu,kpu);

if (! ((pl == flag.bbox().lower()) &&
       (pu == flag.bbox().upper()))) {
  Coords ps = Coords(3,s);
  BBox newB(pl,pu,ps);
  GridData(3)<short> subgridflag(newB);
  //  cout <<"  PRUNE :" << newB <<"\n";
  //  cout.flush();
  subgridflag = flag;
  /* Check my efficiency! */
  int flagged = 0;
  for_3(ii, jj, kk, subgridflag.bbox(), subgridflag.stepsize())
    flagged += subgridflag(ii,jj,kk);
  end_for
  if (((double)flagged / (double)newB.size()) >= Efficiency) {
    recurseOnThis.add(newB);
    return;
  } else {
    Cluster3R(subgridflag,Efficiency, MinWidth, recurseOnThis);
    return;
  }
}

/* Find the slice positions */
int istr=-1,   jstr=-1,   kstr=-1;
int islice=-1, jslice=-1, kslice=-1;
Cluster_Slice(i_sig, &islice, &istr, BiggestExtent*flag.bbox().stepsize(0));
Cluster_Slice(j_sig, &jslice, &jstr, BiggestExtent*flag.bbox().stepsize(1));
Cluster_Slice(k_sig, &kslice, &kstr, BiggestExtent*flag.bbox().stepsize(2));

//  cout << "SLICE:" << islice << " " <<
//  jslice << " " << kslice << "\n";
//  cout << "STR  :" << istr << " " <<
//  jstr << " " << kstr << "\n";
//  cout.flush();
if (islice+jslice+kslice == -3) {
  //  cout << "RETURNING " << flag.bbox() << endl
  //  << flush;
  recurseOnThis.add(flag.bbox());
  return;
}

/* Split along the line with the biggest strength */
BBox divme = flag.bbox();
if (istr > jstr && istr > kstr) {
  l[0] = divme.lower(0); l[1]= divme.lower(1); l[2] = divme.lower(2);
  u[0] = islice; u[1]= divme.upper(1); u[2] = divme.upper(2);
  tmpresult.add(BBox(3,l,u,s));
  l[0] = islice+s[0]; l[1]= divme.lower(1); l[2] = divme.lower(2);
  u[0] = divme.upper(0); u[1]= divme.upper(1); u[2] = divme.upper(2);
```

```
  tmpresult.add(BBox(3,l,u,s));
} else if (jstr > kstr) {
  l[0] = divme.lower(0); l[1]= divme.lower(1); l[2] = divme.lower(2);
  u[0] = divme.upper(0); u[1]= jslice; u[2] = divme.upper(2);
  tmpresult.add(BBox(3,l,u,s));
  l[0] = divme.lower(0); l[1]= jslice+s[1]; l[2] = divme.lower(2);
  u[0] = divme.upper(0); u[1]= divme.upper(1); u[2] = divme.upper(2);
  tmpresult.add(BBox(3,l,u,s));
} else {
  l[0] = divme.lower(0); l[1]= divme.lower(1); l[2] = divme.lower(2);
  u[0] = divme.upper(0); u[1]= divme.upper(1); u[2] = kslice;
  tmpresult.add(BBox(3,l,u,s));
  l[0] = divme.lower(0); l[1]= divme.lower(1); l[2] = kslice+s[2];
  u[0] = divme.upper(0); u[1]= divme.upper(1); u[2] = divme.upper(2);
  tmpresult.add(BBox(3,l,u,s));
}

/* foreach New bbox */
for (BBox* newB = tmpresult.first(); newB; newB = tmpresult.next()) {
  /* Figure out how many points are flagged */
  GridData(3)<short> subgridflag(*newB);
  //  cout << "Handling " << *newB << "\n";
  subgridflag = flag;
  int flagged = 0;
  double myEff;
  for_3(ii, jj, kk, subgridflag.bbox(), subgridflag.stepsize())
    //  cout << "SGF: "<< ii <<" "<< jj <<" "
    //  << kk <<" " <<subgridflag(ii,jj,kk)<<"\n";
    //  cout.flush();
    flagged += subgridflag(ii,jj,kk);
  end_for
  myEff = (double)flagged/(double)((*newB).size());
  /* Do we have ANY? */
  if (flagged) {
    //  cout <<"Flagged " << flagged << " "
    //  << *newB << "\n";
    //  cout <<"Efficiency "<< myEff << " of "
    //  << (*newB).size() <<"\n";
    //  cout.flush();
    /* Stopping Condition */
    if (myEff >= Efficiency ||
        (((*newB).size()) < 8) ||
        (islice + jslice + kslice == -3)) {
      /* No. This one stays = the list if it contains flagged points */
      /* Don't add a degenerate grid! */
      int add = 1;
      for (BBox *tmp = recurseOnThis.first(); tmp;
           tmp = recurseOnThis.next()) {
        if ((*tmp).inside((*newB).lower()) &&
            (*tmp).inside((*newB).upper())) add = 0;
      }
      if (add) {
        //  cout << "  Adding "<< *newB <<"\n";
        //  cout.flush();
        recurseOnThis.add(*newB);
      }
    } else {
      /* Yes: Recluster this sub-box */
      //  cout << "FALL: Flagged " << flagged << " "
      //  << *newB << "\n";
      //  cout <<"FALL: Efficiency "<<myEff <<" of "
```

```
        //  << (*newB).size() <<"\n";
        //  cout.flush();
        BBoxList subcluster;
        Cluster3R(subgridflag, Efficiency,
                  MinWidth, subcluster);
        for (BBox *me = subcluster.first(); me; me = subcluster.next())
          recurseOnThis.add(*me);
      }
    }
  }
  return;
}


/*
   routine    Cluster_Prune
   date       Mon Mar 25 15:12:06 1996
   author     Paul Walker
   desc
   Prunes out external zeros and returns a lower and upper bound for
   prune locations.

*/


void Cluster_Prune(GridData(1)<int> &sig, int *pl, int *pu,
                   int bw) {
  *pl = sig.bbox().lower(0);
  *pu = sig.bbox().upper(0);
  if (*pu - *pl <= bw) {
    return;
  }
  while (sig(*pl) == 0 && *pl < sig.bbox().upper(0))
    *pl += sig.bbox().stepsize(0);
  while (sig(*pu) == 0 && *pu > sig.bbox().lower(0))
    *pu -= sig.bbox().stepsize(0);
  if (*pu - *pl < bw) {
    int diff = bw - (*pu - *pl);
    int addl, addu;
    int udist = sig.bbox().upper(0) - *pu;
    int ldist = *pl - sig.bbox().lower(0);

    addl = ((diff/sig.bbox().stepsize(0))/2) * sig.bbox().stepsize(0);
    addu = diff - addl;
    if (addu > udist) {
      addu = udist;
      addl = diff - addu;
    }
    if (addl > ldist) {
      addl = ldist;
      addu = diff - addu;
    }

    *pu += addu;
    *pl -= addl;
  }
}

/*
   routine    Cluster_Slice
   date       Mon Mar 25 15:09:21 1996
```

```
   author     Paul Walker
   desc
   Looks for a laplacian or zero slice in the internal grid area.
   Since prune has already been called, we know this will happen
   inside!
*/


void Cluster_Slice(GridData(1)<int> &sig, int *slice, int *str,
                   int bw) {
  int i, l, u, s, t, a;         // ctr, low, up, step, str, answer
  l = sig.bbox().lower(0);
  u = sig.bbox().upper(0);
  s = sig.bbox().stepsize(0);

  GridData(1)<int> lap(l,u,s);

  /* Since this box is already pruned, we don't want to split it if
     it is less than 2 the min width */
  if (u-l < 2*bw) {
    *str   = -1;
    *slice = -1;
    return;
  }

  t = -1;
  a = -1;

  /* Look for a zero near the center */
  for (i=l; i <= u-s; i+=s) {
    if (sig(i) == 0) {
      int di = (i-l < u-i ? i-l : u-i);
      if (di > t) {
        a = i;
        t = di;
      }
    }
  }
  if (a != -1) {
    *slice = a;
    *str = 10000;
    return;
  }
  t = -1;
  a = -1;

  /* Evaluate the laplacian etc */
  double malap = 0;
  for (i = l+s; i <= u-s; i+=s) {
    lap(i) = sig(i+s) + sig(i-s) - 2*sig(i);
    if (abs(lap(i)) > malap) malap = abs(lap(i));
  }

  /* And find the strongest second derivative */
  if (malap > 0) {
    // If it isn't then lap is zero everywhere so don't slice either...
    // This case isn't included here because of the (needed) <=
    // for the graze-zero-but-interesting case which still makes it
    for (i=l+bw; i<=u-bw; i+=s)
      if (lap(i+s)*lap(i) <= 0) {
```

```
        int ts = abs(lap(i+s)-lap(i));
        if (ts > t) {
          t = ts;
          a = i;
          //  cout << "PICKING slice " << i << endl
          //  << flush;
          //  cout << ts << " " << t << " "
          //  << lap(i+s) << " " << lap(i) <<
          //  endl << flush;
        }
      }
  }
  *slice = a;
  *str = t;
}
```

---

tportutil.f

```
c------------------------------------------------------------------------
c FORTRAN 77 utility routines for tranport code
c------------------------------------------------------------------------
c Routine to read in grid properties and initial data parameters
c
                subroutine readinput(nx, ny, xmin, xmax, ymin, ymax,
     .                               cfl,
     .                               niters, ml, thresh, re, mineff,
     .                               bufw, blkw, oe)
                        implicit none

                        integer nx, ny
                        real*8  xmin, xmax, ymin, ymax,
     .                          cfl, thresh,
     .                          mineff
                        integer niters, ml, re, bufw, blkw, oe, d, li
                        integer chk

c               Open file and read in parameters

                        open(unit=1,file='input.par')

                                read(1,*)nx, ny
                                read(1,*)xmin, xmax
                                read(1,*)ymin, ymax
                                read(1,*)cfl
                                read(1,*)niters
                                read(1,*)ml
                                read(1,*)thresh
                                read(1,*)re
                                read(1,*)mineff
                                read(1,*)bufw
                                read(1,*)blkw
                                read(1,*)oe

                        close(unit=1)
```

```
                        return
                end
c-----------------------------------------------------------------------

c-----------------------------------------------------------------------
      subroutine pythnorm(lb, ub, shape, truncU, truncerr,
     .                    wgtu)
c-----------------------------------------------------------------------
        implicit none

        integer lb(2), ub(2), shape(2)

        real*8  truncU(shape(1), shape(2)),
     .          truncerr(shape(1), shape(2))

        real*8  wgtu

c       Local variables

        integer i, j

        do i = 1, shape(1)
          do j = 1, shape(2)
            truncerr(i,j) = sqrt( wgtu * truncU(i,j)**2 )
          end do
        end do

        return
       end
c-----------------------------------------------------------------------
```

---

input.par

```
33 33
0.0 1.0
0.0 1.0
0.115
10
2
0.05
4
0.8
1
1
1
#
nx ny
xmin xmax
ymin ymax
cfl factor
number of iterations
maximum level of refinement
threshold for regridding from truncation error estimate
regrid after how many time steps
minimum efficiency for clustering algorithm
```

```
buffer width
minimum block width
output every
```

---

truncation.C

```
//-----------------------------------------------------------------------
// Routine to carry out a truncation error estimate
// in first order form
//-----------------------------------------------------------------------

#include "DAGH.h"
#include "tportamr.h"
#include "tportfortran.h"

#define VizServer       0       // Who will write to stdout

void wave_trunc_est(GridHierarchy &GH,
      INTEGER const t,
      INTEGER const l,
      GridFunction(DIM)<GFTYPE> &U,
      GridFunction(DIM)<GFTYPE> &trunc_err)
{

      GFTYPE normu, inormu, tnormu, enorm;

      GridFunction(DIM)<GFTYPE> truncU("truncU",U,t,l,
                                       DAGHComm,
                                       DAGHHasShadow,
                                       DAGHNoBoundary,
                                       DAGHNoAdaptBoundary);

//-----------------------------------------------------------------------
// Subtract the DAGH_Main grid from shadow pointwise
//-----------------------------------------------------------------------
      forall(truncU, t, l, c)
         truncU(t,l,c,DAGH_Shadow) = U(t,l,c,DAGH_Shadow);
         truncU(t,l,c,DAGH_Shadow) -= U(t,l,c,DAGH_Main);
      end_forall

//-----------------------------------------------------------------------
// "Relativize" wrt to norms of U, (DAGH_Main)
//
// Check if the norms are too small... If they are then we assume that
// the grid functions consist of small quantities and do not divide by
// the norm.
//-----------------------------------------------------------------------
      normu = Norm2(U, t, l, DAGH_Main);
      if (normu == 0.0) {
         inormu = 1.0;
      } else {
         inormu = 1.0 / (normu*normu);
      }

//-----------------------------------------------------------------------
```

```
// Take a Pythogorean norm of the truncation error
// trunc_err must have the same bbox properties as truncU.
//------------------------------------------------------------------------------
      forall(trunc_err, t, l, c)
        f_pythnorm(FBA(trunc_err(t,l,c,DAGH_Shadow)),
                   FDA(truncU(t,l,c,DAGH_Shadow)),
                   FDA(trunc_err(t,l,c,DAGH_Shadow)),
                   &inormu);
      end_forall

      Sync(trunc_err, t, l, DAGH_Shadow);

//------------------------------------------------------------------------------
// Prolong DAGH_Shadow trunc_err to DAGH_Main trunc_err
//------------------------------------------------------------------------------
      INTEGER myargc = 0; GFTYPE myargs[1];
      forall(trunc_err,t,l,c)
        f_prolong_amr(FDA(trunc_err(t,l,c,DAGH_Shadow)),
                      FBA(trunc_err(t,l,c,DAGH_Shadow)),
                      FDA(trunc_err(t,l,c,DAGH_Main)),
                      FBA(trunc_err(t,l,c,DAGH_Main)),
                      FBA(trunc_err(t,l,c,DAGH_Main)),
                      myargs, &myargc);
      end_forall

      Sync(trunc_err, t, l, DAGH_Main);
} /* end wave_trunc_est */
```

Makefile

```
#helpon
##########################################################################
#                                                                        #
# Makefile for the DAGH applications                                     #
#                                                                        #
# Allowable external targets:                                            #
#                                                                        #
#       - help                  print out this top banner message        #
#       - clean                 remove all files generated by the make   #
#                                                                        #
# Author:  Manish Parashar <parashar@cs.utexas.edu>                      #
#                                                                        #
##########################################################################
#helpoff

help:
	@awk '/#helpon/, /#helpoff/' Makefile |          \
	egrep -v "(helpon|helpoff)" | more


# Top directory for the DAGH system !!
DAGH_HOME = /u8/mitra/NDAGH

# Include architecture specific makefile
include make.defn
```

```
#########################################################################
# Application Makefile !!!                                              #
#########################################################################

.SUFFIXES: .C .c .F .o .a .f .f9 .cpp

# Define application FORTRAN sources & objects
APP_F_SRC = transport.f tportutil.f grid.f
APP_F_OBJ = $(APP_F_SRC:.f=.o)

# Define application C++ sources & objects
APP_CC_SRC = tportamr.C truncation.C Cluster3.C Cluster1.C
APP_CC_OBJ = $(APP_CC_SRC:.C=.o)

APPOBJ = $(APP_F_OBJ) $(APP_CC_OBJ)

# Application specific Flags
C++APPFLAGS = -g
CAPPFLAGS = -g
F77APPFLAGS = -g
F90APPFLAGS = -g

# Application specific libraries
APPLIB =

EXEC =          tportamr

$(EXEC):        $(APPOBJ)
                $(RM) $(EXEC)
                $(C++LINK) $(C++FLAGS) -o $@ \
                $(APPOBJ) $(APPLIB) $(LDLIBS)


#########################################################################
#                                                                       #
# Clean up the application files, any core files, the compiler created  #
# template repository !                                                 #
#                                                                       #
#########################################################################

clean:
                $(RM) *.o core $(EXEC)
                $(RM) -r $(REPOSITORY)
```

Obtaining DAGH

You can obtain the latest version of DAGH by anonymous ftp from the site *ftp.cs.utexas.edu*. Enter **anonymous** for user name and your complete e-mail address when prompted for the password. Change directory to */pub/dagh*. Specify binary transfer mode and get the file **ndagh_dist050597.tar.Z.uu**.

To uncompress and untar the file perform the following operations:

- **uudecode** ndagh_dist050597.tar.Z.uu
- **uncompress** ndagh_dist050597.tar.Z

- **tar -xvf** ndagh_dist050597.tar

This will create a directory NDAGH. Change directory to NDAGH to configure and install DAGH for the architecture you are working on.

Installation Overview

DAGH depends only on MPI and needs to know where this resides on the machine it is being installed on. In addition it also uses HDF (NCSA) and RNPL (M. Choptuik et al.) for IO.

To install DAGH

- Ensure that all required software has been installed.
- Set environment variables if necessary.
- Select the machine specific makefile. This is done using the following command:
    configure *architecture*
  where *architecture* is one of rs6000, sp2, sgi, or crayt3e.
- Edit **make.defn**. This file consists of all the definitions specific to the architecture. In particular the locations of MPI libraries must be appropriately set. If HDF or RNPL IO is used the home directory of these packages needs to be supplied.
- Compilation
    - to make the main DAGH library
        make dagh
    - to make the main DAGH IO support library with HDF and RNPL
        make daghio
    - to make both DAGH and DAGH IO libraries
        make all

  In any DAGH application, the variable *DAGH_HOME* must point to the installation of DAGH. A sample makefile (Makefile.App) is present in the utilities directory for illustration.

DAGH has been successfully installed on the following platforms:

- IBM RS6000 running AIX: single processor and network of workstations
- IBM SP2 running AIX
- SGI running IRIX: single processor and network of workstations
- SGI Power Challenge Array
- SGI O2000
- Cray T3E

Modifications may be necessary for use on other platforms.

Software prerequisites

- HDF (HDF4.0rl or more recent recommended). Available from *ftp://ftp.ncsa.uiuc.edu/HDF/*

- MPI, if DAGH is to be built to run in parallel
- RNPL (optional). Available from *ftp://helmholtz.ph.utexas.edu/pub/rnpl/rnpl.tar.Z*

Environment variables and 'configure' (GENERAL)

NOTE: *configure* is a *sh* script. DO NOT use *csh* tilde-notation such as *~/bin* to set environment variables: use *$HOME/bin* etc. instead.

If the libraries and/or header files associated with the prerequisite software (currently hdf, mpi, RNPL) have been installed in some non-canonical location (i.e. NOT in /usr/local/lib and /usr/local/include) set the environment variable(s)

- LIB_PATHS
- INCLUDE_PATHS

to define all necessary *paths* to the libraries and headers respectively before configuring. For example assuming that

```
'libdf.a'      lives in '/home/jdoe/HDF/lib'
'libmfhdf.a'   lives in '/home/jdoe/NETCDF/lib'
'df.h'         lives in '/home/jdoe/HDF/include'
'netcdf.h'     lives in '/home/jdoe/NETCDF/include'
```

then

```
setenv LIB_PATHS "/home/jdoe/HDF/lib /home/jdoe/NETCDF/lib"
setenv INCLUDE_PATHS "/home/jdoe/HDF/include /home/jdoe/NETCDF/include"
```

before configuring.

If configure gets some of the variables wrong in the makefile, just set the appropriate environment variable to the value you prefer and rerun configure.

For instance, configure looks for gcc. If it finds gcc, configure sets CC to gcc, otherwise it is set to cc. To override this behaviour (you have both gcc and cc, but want to use cc) just
setenv CC cc
and then run configure.

By default, configure will set CFLAGS in the makefiles to -g. If you would like this changed to something else, for instance -O2, simply
setenv CFLAGS -O2
and then run configure.

Environment variables and 'configure'

(GLOSSARY)

LIST of environment variables which can be used to communicate with the *configure* script

```
(a) Standard autoconfig variables

    Variable: CC
    Synopsis: Name of C compiler.
    Sample usage: The following will ensure that 'cc' is used
       for the C compiler.

       setenv CC cc

    Variable: CXX
    Synopsis: Name of C++ compiler.
    Sample usage: The following will ensure that 'CC' is used
       for the C++ compiler.

       setenv CXX CC

    Variable: CFLAGS
    Synopsis: Flags to be passed to the C compiler (the configure
       script will generally append additional flags)
    Sample usage:

       setenv CFLAGS -O2

    Variable: CPPFLAGS
    Synopsis: Flags to be passed to the C pre-processor  (the configure
       script will generally append additional flags)
    Sample usage:

       setenv CPPFLAGS -DSOME_FLAG

    Variable: CXXFLAGS
    Synopsis: Flags to be passed to the C++ compiler (the configure
       script will generally append additional flags)
    Sample usage:

       setenv CXXFLAGS -O2

    Variable: LDFLAGS
    Synopsis: Flags to be passed to the loader/linker (the configure
       script will generally append additional flags)
    Sample usage:

       setenv LDFLAGS -lrnpl

(b) BBH/DAGH specific variables

    Variable: INCLUDE_PATHS
    Synopsis: Prepends specified paths to list of places to search
       for C, C++ header files.
    Sample usage:

        setenv INCLUDE_PATHS "$HOME/hdf/include $HOME/rnpl/include"

    Variable: LIB_PATHS
    Synopsis: Prepends specified paths to list of places to search
```

```
   for libraries
Sample usage:

    setenv LIB_PATHS "$HOME/hdf/lib $HOME/rnpl/lib"

Variable: F77
Synopsis: Name of Fortran 77 compiler.  Currently set
   automatically by 'aclocal.m4' and thus can not be
   overridden by setting F77 environment variable.

Variable: F77FLAGS
Synopsis: Flags to be passed to Fortran 77 compiler.
    Setting the corresponding environment variable will
    OVERRIDE any settings normally made by 'aclocal.m4'
Sample usage:

    setenv F77FLAGS -O2

Variable: LEX
Synopsis: Defines name of lex-compatible lexer generator.
   As currently used, must be either 'lex' or 'flex.  If the
   user has both, but prefers to use flex, then he/she should

   setenv LEX flex

   before configuring

Variable: YACC
Synopsis: Defines name of yacc-compatible parser generator
   As currently used, must be either 'yacc' or 'bison -y'.  If the
   user has both, but prefers to use 'bison', then he/she should

   setenv YACC "bison -y"

   before configuring

Variable: PERL
Synopsis: Defines name of perl interpreter. As currently
   used, must evaluate to 'perl'.

Variable: RANLIB
Synopsis: Defines name of 'ranlib' command. As currently
   used, will evaluate to either 'ranlib' or 'touch'.

Variable: AR
Synopsis: Defines name of library-building command. As
   currently used,  will evaluate to either 'ar' or
   'touch'.

Variable: RPCGEN
Synopsis: Defines name of rpcgen compiler. Should
   normall evaluate to 'rpcgen'
```

Miscellaneous and machine-specific notes

```
(a) Installation prefix:

    'make' installs the programs in /usr/local(bin,lib,include) by
```

default. To override this, use the --prefix=mydir option when executing configure.

(b) On O2000 system at NCSA (modi4, etc.), before configuring

```
setenv CXX CC
setenv CXXFLAGS "-woff 1021"
```