Parallel Multigrid with the DAGH package: Specifications and Applications

Matthew W. Choptuik, Scott Klasky and Manish Parashar
Center for Relativity and Dept. of Computer Science
The University of Texas at Austin
Austin, TX

Draft, September 1995

1 Introduction

In the past two decades, the multigrid technique pioneered by Achi Brandt [1, 2] has revolutionized the task of solving elliptic partial differential equations using finite-difference or finite-element methods. The advantages of using multigrid on vector or superscalar architectures are by now well known: an appropriately constructed multigrid algorithm typically provides an acceptable solution of the N algebraic equations resulting from a discretization of a general elliptic system (possibly non-linear) using $O(N)$ operation and $O(N)$ storage. Furthermore, the problem-dependent multiplicative constants implicit in this time-complexity estimate are generally quite smal—often a few to a few 10's. Thus, in practice, a properly tuned multigrid algorithm often provides the most efficient means to solve elliptic problems, particularly as the problem size, N, becomes large. In addition, multigrid methods can be naturally accommodated into adaptive-mesh-refinement (AMR) schemes such as the algorithm proposed by Berger and Oliger [3].

A key feature of multigrid algorithms is that the computational work tends to be dominated by local operations on uniform data structures (typically represented, for example, as multi-dimensioned FORTRAN arrays). Thus, at least in principle, these methods appear to be good candidates for parallelization and there is by now considerable evidence that careful implementations can lead to good performance on parallel architectures, including distributed memory machines.

In this context, the following report summarizes the requirements which have been identified and implemented for support of multigrid methods within the DAGH package [4, 5]. Under the general auspices of the Binary Black Holes Grand Challenge project, DAGH has been developed by one of us (Parashar) and J. .C Browne to provide low-level support for parallel, adaptive, grid-based computations. In addition to providing continuing documentation of the types of multigrid algorithms supported by the DAGH infrastructure, the "Requirement Specifications" section of this document constitutes an example of the type of information a computational scientist should provide in order to expedite the incorporation of support for his/her algorithms within DAGH.

2 Requirement Specifications

2.1 DAGH Overview

A principal design criterion of DAGH (if not *the* principal criterion) was to provide support for parallelization and adaptivity of grid-based calculations arising in numerical relativity in a manner which would minimally impact the way in which numerical relativists coded their difference schemes. The requirements in this regard are fairly straightforward and are best understood by way of a simple example.

Consider the two dimensional wave equation with unit propagation speed:

$$\frac{\partial^2 \phi\,(x,y,t)}{\partial t^2} = \frac{\partial^2 \phi\,(x,y,t)}{\partial x^2} + \frac{\partial^2 \phi\,(x,y,t)}{\partial y^2}, \tag{1}$$

to be solved on the unit square, subject to homogeneous boundary conditions

$$\phi\,(0,y,t) = \phi\,(1,y,t) = \phi\,(x,0,t) = \phi\,(x,1,t) = 0. \tag{2}$$

Replacing the continuum domain with a uniform finite-difference mesh,

$$(x_i, y_j) \quad = \quad ((i-1)\,h, (j-1)\,h) \qquad i,j = 1,2,\ldots,N \tag{3}$$

$$t^n \quad = \quad n\lambda h \qquad\qquad n = 0,1,\ldots \tag{4}$$

where the mesh-spacing, h, is given by $h = 1/(N+1)$, i and introducing the standard finite difference notation:

$$\phi^n_{i,j} \equiv \phi\,((i-1)\,h, (j-1)\,h, n\lambda h)\,, \tag{5}$$

a suitable finite-difference discretization of (1) is

$$\frac{\phi^{n+1}_{i,j} - 2\phi^n_{i,j} + \phi^{n-1}_{i,j}}{\lambda^2 h^2} = \frac{\phi^n_{i+1,j} + \phi^n_{i-1,j} + \phi^n_{i,j+1} + \phi^n_{i,j-1} - 4\phi^n_{i,j}}{h^2} \qquad i,j = 2,3,\ldots N-1 \tag{6}$$

This difference equation can be trivially solved for the advanced value $\phi^{n+1}_{i,j}$:

$$\phi^{n+1}_{i,j} = 2\phi^n_{i,j} - \phi^{n-1}_{i,j} + \lambda^2\left(\phi^n_{i+1,j} + \phi^n_{i-1,j} + \phi^n_{i,j+1} + \phi^n_{i,j-1} - 4\phi^n_{i,j}\right) \qquad i,j = 2,3,\ldots N-1 \tag{7}$$

providing the means to *update* the grid-function values $\phi^{n+1}_{i,j}$ given retarded grid-function values $\phi^n_{i,j}$ and $\phi^{n-1}_{i,j}$. A FORTRAN 77 subroutine which implements (7) might look like this:

```
      subroutine update_phi(phinp1,phin,phinm1,n,lambda)
         integer        n
         real*8         phinp1(n,n),   phin(n,n),   phinm1(n,n)
         real*8         lambda

         integer        i,             j

         do j = 2 , n - 1
            do i = 2 , n - 1
               phinp1(i,j) = 2.0d0 * phin(i,j) - phinm1(i,j) + lambda**2 * (
     &            phin(i+1,j) + phin(i-1,j) + phin(i,j+1) + phin(i,j-1) -
     &            4.0d0 * phin(i,j)                                         )
            end do
         end do

         return
      end
```

Though simple, this example illustrates the basic features which we believe are common to the majority of numerical relativity codes that have been constructed by Alliance members. Key features include:

- The use of a regular mesh structures; either uniform (constant mesh spacing in each coordinate direction) as in the example, or simple-graded (mesh spacings can vary but coordinates remain cartesian products of 1-dimensional vectors of coordinates). [1]
- The use of a number of *grid functions* (`phinp1, phin, phinm1` in the example) defined on the computational grid. These grid functions are most easily represented and manipulated as multi-dimensional (for the space-dimensions) `FORTRAN` arrays and it is often convenient to use distinct grid functions for different time levels ($n+1, n, n-1$ etc.) of the same physical quantity.
- The use of finite-difference expressions which couple unknowns *locally*, with an *a priori* specified radius of communication in each spatial dimension. In the example above, the "interactions" are "nearest-neighbor" for a radius of communication of 1 in all directions.

A subprogram such as `update_phi`, which takes a number of grid functions as input, performs local (and generally uniform), finite-radius operations on the inputs to produce one or more output grid-functions is generically called an *update* routine. In many ways, `DAGH`'s primary *raison d'etre* is to provide support for automatic parallelization and adaptivization of *update* routines. If and when `DAGH` distributes storage associated with a given grid-function across multiple processors, it automatically creates *ghost regions*—these may be viewed as *read-only* boundary regions which duplicate corresponding *read-write* values on adjacent processors. Updates *use* ghost values, but don't modify them; in this way an effective and general means of parallelizing a significant class of calculations (including most explicit finite-difference computations) is provided. In many cases, then, `DAGH`-incorporation of a numerical-relativity algorithm consisting of a series of updates conforming to the structure sketched will require very little modification of code, and usually *no* modification of inner-loop code. Conversely, the task of preparing requirement specifications for adding support to `DAGH` will generally reduce to identifying those components of the algorithm which cannot be viewed as finite-radius, explicit, grid-function updates. [2]

2.2 Multigrid Overview

Consider the two dimensional boundary value problem:

$$\frac{\partial^2 u\,(x,y)}{\partial x^2} + \frac{\partial^2 u\,(x,y)}{\partial y^2} = f\,(x,y,) \qquad \text{on} \qquad [0\ldots 1]\times[0\ldots 1] \tag{8}$$

with $f(x,y)$ a specified function, and $u(x,y)$ subject to boundary conditions:

$$u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0. \tag{9}$$

Adopting previously-defined finite difference notation, a suitable discretization of (8) is

$$\left[L^h u\right]_{i,j} \equiv \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{h^2} = f_{i,j} \qquad i,j = 2,3,\ldots N-1 \tag{10}$$

[1] Although the use of more than one mesh ("staggering") has been fairly commonplace in numerical relativity work, current Alliance policy is to employ only a *single* mesh per calculation

[2] At the same time, it is only prudent for the numerical relativist who wants to implement his algorithm using `DAGH` to consider reformulating his/her procedures, where feasible, to use existing support.

An *iterative* method for solving (10) starts with some initial guess, $u^{(0)}$ and the applies some update procedure $u^{(n+1)} = U[u^{(n)}, u^{(n-1)}, \ldots]$ such that

$$\lim_{n\to\infty} u^{(n)} = u \tag{11}$$

Relaxation techniques constitute an important subset of iterative methods. Typical of these is *Gauss-Seidel* (GS) relaxation in which, for difference systems such as (10), one visits each mesh point, (x_i, y_j) and adjusts the corresponding unknown value, $u_{i,j}$ so that the single equation

$$\left[L^h \tilde{u}\right]_{i,j} = f_{i,j}, \tag{12}$$

associated with the mesh point, is instantaneously satisfied. Here, $\tilde{u}$ denotes the stored values of the entire grid function (i.e. the instantaneous computed approximation of the grid function). A general way of updating an individual unknown $u_{i,j}^{(n)}$ in order to satisfy (12) (or improve the estimate in the non-linear case) is to first compute the *residual* component, $\tilde{r}_{i,j}$:

$$\tilde{r}_{i,j} := \left[L^h \tilde{u}\right]_{i,j} - f_{i,j} \tag{13}$$

and then perform a single-Newton step:

$$\tilde{u}_{i,j}^{(n+1)} := \tilde{u}_{i,j}^{(n)} - \delta\tilde{u}_{i,j}^{(n)} = \tilde{u}_{i,j}^{(n)} - \frac{\tilde{r}_{i,j}}{J_{i,j}} \tag{14}$$

where $J_{i,j}$ is the diagonal component of the Jacobian associated with (10):

$$J_{i,j} := \frac{\partial \left[L^h \tilde{u}\right]_{i,j}}{\partial u_{i,j}}. \tag{15}$$

A complete iteration of a relaxation technique such as GS, in which every mesh point is visited once, is known as a *relaxation sweep*. Asymptotically, the rate of convergence of the scheme (the rate at which the residual, $\tilde{r}$, gets driven to zero) is independent of the particular order in which the individual unknowns, $u_{i,j}$, are visited during a sweep: in practice, however this order is quite important. In particular, for radius-1 equations such as (10), so called Red-Black [3] ordering, in which all values $u_{i,j}$ such that $\mathrm{mod}(i+j, 2) = 0$ are first updated followed by all values for which $\mathrm{mod}(i+j, 2) = 1$, has at least two key advantages. First, the scheme vectorizes, the import of which clearly depends on the target architecture. Second, and more germane to the current discussion, Red-Black Gauss-Seidel (RBGS) can be used to construct particularly efficient multigrid solvers. Here's a `FORTRAN` routine which implements RBGS for the system (10):

```
      real*8 function relax_rb(u,f,n,h,rb)
         integer       n,          rb
         real*8        u(n,n),     f(n,n),     h

         real*8        hm2,        jacelm1,    res
         integer       nupdates,   i,          ist,       j

         relax_rb = 0.0d0
```

[3] in analogy with the red and black squares on a chess- or checker-board

```
      nupdates = 0
      hm2      =  1.0d0 / (h * h)
      jacelm1  = -0.25d0 * h * h
      ist = rb + 1
      do j = 2 , n - 1
         do i = ist  + 1 , n - 1 , 2
            res = hm2 * (u(i+1,j) + u(i-1,j) + u(i,j+1) + u(i,j-1) -
&                        4.0d0 * u(i,j)) - f(i,j)
            u(i,j)   = u(i,j) - jacelm1 * res
            relax_rb = relax_rb + res * res
            nupdates = nupdates + 1
         end do
         ist = 3 - ist
      end do
      relax_rb = sqrt(relax_rb) / nupdates
      return
   end
```

Each invocation of this routine effects either a red or black half-sweep, depending on the value of the input argument `rb`; the sequence

```
   resnrm = relax_rb(u,f,n,h,0)
   resnrm = relax_rb(u,f,n,h,1)
```

results in a complete RBGS sweep. Clearly this is an update routine of the type discussed above.

It has long been known that relaxation techniques, such as RBGS, although easy to implement, suffer from slow convergence, particularly in the limit $h \to 0$. Heuristically, and viewing the process of residual annihilation (equivalent to convergence of iterative procedure) in the frequency domain; relaxation tends to quickly damp out short-wavelength[4] residual components, while long-wavelength components are damped much more slowly. In other words, relaxation tends to be an efficient *smoother* of residuals rather than an effective *annihilator* of them. This observation is central to understanding the multigrid technique: once a problem is smooth on the scale of the mesh, it can be well-approximated on a mesh with coarser mesh-spacing, H (in practice, we will always take $H = 2h$). Relaxation on this coarser mesh can be used to efficiently damp those residual components which have wavelengths comparable to H, then an even-coarser problem can be posed, etc. As one progresses up the implied hierarchy of ever-coarsening grids, the amount of computational work expended in relaxation decreases geometrically so that, especially for multi-dimensional problems, the amount of work required to perform the *coarse grid correction* is small compared to the relaxation work needed to smooth the system on the finest level.

Besides the relaxation technique used to smooth residuals, then, there are two other key operations in a multigrid algorithm: the initiation and termination of a coarse grid correction. These are illustrated here in the form of the so-called *Correction Scheme* (CS), which is applicable only to linear systems such as (10). Derivation of the equations begins with the observation that given an approximate solution, $\tilde{u}^h$, of (10), with associated residual, $\tilde{r}^h$, determining the actual solution u^h is equivalent to finding a correction δu^h defined by

$$u^h = \tilde{u}^h + \delta u^h \tag{16}$$

[4] defined with respect to the mesh spacing, h

and which thus satisfies:

$$\left[L^h \delta u^h\right]_{i,j} = -\tilde{r}^h_{i,j}. \tag{17}$$

If the residual has been smoothed (as, we have claimed, is readily accomplished via relaxation techniques); a coarse-grid version of (16) may be sensibly posed:

$$\left[L^{2h} \delta u^{2h}\right]_{i,j} = -I^{2h}_h \tilde{r}^h_{i,j}. \tag{18}$$

Here, I^{2h}_h is a fine-to-coarse transfer operator, commonly known as a *restriction* operator—a specific example will be given shortly. Equation (18) is used to initiate and perform a coarse grid correction (CGC); the CGC is complete when an acceptable solution to the coarse-grid problem has been attained. The fine-grid function is then updated using the prescription:

$$\tilde{u}^h := \tilde{u}^h + I^h_{2h} \delta \tilde{u}^{2h}, \tag{19}$$

where $\delta \tilde{u}^{2h}$ is the (approximate) solution of (18) and I^h_{2h} is another transfer operator (this time coarse-to-fine), known as a *prolongation*.

As already hinted, the coarse-grid-correction idea can be applied recursively, so that, for example, relaxation sweeps are applied to smooth (18), then an even-coarser problem is posed on a grid with mesh-spacing $4h$, etc. Eventually, the algorithm encounters a system which is so coarse that there are only a very few unknowns and the computational cost of solving the system exactly (to the level of round-off error) is negligible. Once this very coarse system has been solved, equation (19) is employed recursively to update the various grid-functions on increasingly fine levels. It is convenient to label the L levels, ℓ, of discretization from $1, 2 \ldots, L$, where $l = L$ is the finest level (the original problem) and $l = 1$ is the coarsest. The process of successively relaxing on levels $l = L, L-1, \ldots 2$, solving a very coarse system at $l = 1$, and then updating grid-functions from $l = 2, 3, \ldots L$ using (19) is called a *V-cycle*. One final point is in order. The process of "prolonging" (typically linear interpolation) a coarse grid correction to the corresponding fine grid will generically induce new short-wavelength components in the fine-grid unknown and the fine-grid residual. Thus, after (19) is applied, the multigrid method calls for additional smoothing sweeps which will effectively annihilate these components.

2.3 Pseudo-code of Correction Scheme (CS) V-cycle

In preparing requirement specifications it will usually be very useful for the numerical relativist to write a *pseudo-code* form of his/her algorithm. The pseudo-code should aim to make as clear as possible:

- All basic data structures required by the algorithm
- All basic operations/functions performed on the data structures during execution of the algorithm.

In delineating operations/functions, emphasis should be placed on identifying distinct communication patterns since it is such considerations which figure most highly when implementing support in `DAGH`. Keeping these guidelines in mind a pseudo-code form of the correction scheme described above follows:

`PROCEDURE CSMain`
Determine problem parameters (size, convergence criterion, etc.)

```
   Define f[nx,ny]
   u := 0; cycle = 0
   DO UNTIL converged
      Vcycle(u,f,nx,ny,h,cycle,pre,post); cycle = cycle + 1
   END DO
END PROCEDURE CSMain

PROCEDURE Vcycle(u,f,nx,ny,h,whichcycle,pre,post)
   IF whichcycle = 1 THEN
      This is first invocation, set up internal data structures
      Compute depth of hierarchy, L, from nx, ny
      DO l = L ...  1
         Allocate storage for grid functions du[l], rhs[l], res[l]
         Define h[l], nx[l], ny[l]
         IF l = L THEN
            du[l] = u; rhs[l] = f
         END IF
      END DO
   END IF
   Multigridding proper begins here
   DO l = L ...  2
      IF l <> L OR whichcycle > 1 THEN
         Apply pre smoothing sweeps
         DO pre TIMES Relax(du[l],rhs[l],nx[l],ny[l],h[l])
      END IF
      Set up coarse-grid system: compute negative of fine-grid residuals, then restrict
      Compute_NegResiduals(res[l],du[l],rhs[l],nx[l],ny[l],h[l])
      Restrict(res[l-1],res[l],nx[l-1],ny[nl-1])
      du[l-1] := 0
   END DO
   Solve coarsest system
   DO Relax(du[1],rhs[1],nx[1],ny[1],h[1]) UNTIL converged
   DO l = 2 ...  L
      Prolong correction, using res[l] as temporary
      Prolong(res[l],du[l-1],nx[l-1],ny[l-1])
      Update fine-grid function
      u[l] := u[l] + res[l]
         Apply post smoothing sweeps
         DO post TIMES Relax(du[l],rhs[l],nx[l],ny[l],h[l])
   END DO
   Update and return new estimate of u
   u := u[l]
END PROCEDURE Vcycle

PROCEDURE Prolong(uf,uc,nxc,nyc)
   Routine implements bi-linear 2:1 interpolation of uc to uf
   ic, jc:  coarse-grid indices
```

```
   if, jf:  fine-grid indices
   FORALL ic, jc
     uf[2*ic-1,2*jc-1] := uc[ic,jc]
   END FORALL
   FORALL even  if AND odd jf
     uf[if,jf] := 0.5 * (uf[if+1,jf] + uf[if-1,jf])
   END FORALL
   FORALL if AND even jf
     uf[if,jf] := 0.5 * (uf[if,jf+1] + uf[if,jf-1])
   END FORALL
END PROCEDURE Prolong


PROCEDURE Restrict(uc,uf,nxf,nyf)
   Routine implements 2:1 half-weighted coarsening with boundary injection
   ic, jc:  coarse-grid indices
   if, jf:  fine-grid indices
   FORALL ic, jc
     if = 2 * ic - 1; jf = 2 * jc - 1
     uc[ic,jc] = 0.5 * (uf[if,jf] + 0.25 *
               (uf[if+1,jf] + uf[if-1,jf] + uf[if,jf+1] + uf[if,jf-1]))
   END FORALL
   Inject boundary values
   FORALL ic
     uc[ic,1] := uf[2*ic - 1,1]; uc[ic,nyc] := uf[2*ic - 1,nyf];
   END FORALL
   FORALL jc
     uc[1,jc] := uf[1,2*jc - 1]; uc[nxc,jc] := uf[nxf,2*jc - 1];
   END FORALL
END PROCEDURE Restrict


PROCEDURE Relax(u,rhs,nx,ny,h)
   Perform a red sweep ..
   Relax_rb(u,rhs,nx,ny,h,0)
   ... then a black one.
   Relax_rb(u,rhs,nx,ny,h,1)
END PROCEDURE Relax


PROCEDURE Relax_rb(u,rhs,nx,ny,h,rb)
   See FORTRAN code for relax_rb above for more details
   FORALL i, j such that mod(i+j,2) = rb
     res := (u[i+1,j] + ...  - 4 u[i,j]) / (h * h) - rhs[i,j]
     u[i,j] := u[i,j] - res / (-4 / (h * h))
   END FORALL
END PROCEDURE Relax_rb
```

```
PROCEDURE Compute_NegResiduals(res,u,rhs,nx,ny,h)
   FORALL Interior i, j
      res[i,j] := -(u[i+1,j] + ...  - 4 u[i,j]) / (h * h) + rhs[i,j]
   END FORALL
   FORALL Boundary i, j
      res[i,j] := 0
   END FORALL
END PROCEDURE Compute_NegResiduals
```

This pseudo-code has been purposefully written with a level of detail which may not be required by all applications. For example, the precise form of calculations in an inner loop are not particularly important compared to, say, the radius of the finite difference calculation it represents. In any case, with the pseudo-code in hand we can assert the following minimal requirements for multigrid support.

2.4 Requirement Specifications Deduced from CS Pseudo-code

1. **Hierarchical (Temporary) Storage:** As written, `Vcycle` takes as input the current estimate of the fine-grid function, `uf` and the fine-grid right-hand-side, `f`. The first invocation of `Vcycle` requires instantiation of a hierarchy of grid-functions:

   - `du[l], rhs[l], res[l],     l = 1, 2, ... L`

   The level `L` grid functions have the same shape (`[nx,ny]`) as the inputs, `u` and `f`. Grid functions defined on coarser levels `l < L`, have shape `[nx[l+1])/2 + 1,ny[l+1])/2 + 1]` but have the same spatial domain as `u` and `f`. From the AMR point of view, the requirements may be viewed as those for a uniform coarse grid with $L-1$ levels of $2:1$ refinement. Each level of refinement consists of a single grid which covers the entire coarse-grid domain. Different multigrid algorithms may require fewer or more auxiliary grid functions per level; the 3 required here is close to a lower bound. In almost all cases, this storage can be viewed as temporary; information in the auxiliary hierarchy cannot, in general, be reused from one invocation of `Vcycle` to another.

2. **Intra-Level Grid-Function Updates:** Routines `Relax`, `Relax_rb`, and `Compute_NegResiduals` implement this type of operation. The requirements for `Relax_rb`, and `Compute_NegResiduals` are precisely those for the generic update routine described in section 2.1. In the specific example implemented above, the communication radius is 1 and ghost regions will need to be updated after each invocation of `Relax_rb`. (`Compute_NegResiduals` is non-iterative and can execute entirely without communication.) Different difference equations may demand larger-radius communications, and different relaxation schemes may produce more complicated communication patterns which will have to be identified on a class-by-class basis.

3. **Inter-Level Grid-Function Updates:** Routines `Prolong` and `Restrict` implement level-to-level communications. Both of these operations can be done entirely locally provided all storage associated with a given spatial location is assigned to the same processor. Note that these routines both have communication radii of 1, but for `Prolong` that radius is with respect to the coarse-grid function as well as the fine-grid function. As with the relaxation routines, different prolongation and relaxation schemes may have different communication radii but should have the same basic structure.

4. **Minimization of Communication Overhead**: On a serial machine, a complexity analysis of a typical multigrid algorithm proceeds on the basis that run-time is dominated by relaxation work. For example, for the 2-d model problem described above, assume that the fine grid has $N = n^2$ points. Then a relaxation sweep on the finest grid (level L) takes approximately cN work. Sweeps on levels $L-1, L-2, \ldots$ take $cN/4, cN/16, \ldots$ operations so an invocation of `Vcycle` expends

$$W_V \approx cN\,(\mathtt{pre} + \mathtt{pst})\left(1 + \frac{1}{4} + \frac{1}{16} + \cdots\right) \approx \frac{4}{3}cN\,(\mathtt{pre} + \mathtt{pst}) = O(N)$$

work. The fact that work (and thus execution time) becomes geometrically negligible on coarser levels is crucial for the $O(N)$ work estimate. However, in a naive parallel implementation, communication must be performed at each level. Furthemore, the communication work/time will tend to be constant—certainly *not* geometrically decreasing—and will thus quickly dominate the computational time as the algorithm advances to coarser levels. We require low-level support to circumvent this problem.

3 `DAGH` Implementation

3.1 `DAGH`: An Infrastructure for Parallel Adaptive Mesh-Refinement Techniques

The `DAGH` (Distributed Adaptive Grid Hierarchy) data-management infrastructure provides support for developing parallel adaptive applications based on adaptive mesh refinement (AMR) techniques. The infrastructure consists of two components:

1. A set of programming abstractions in which computations on dynamic hierarchical grid structures are directly implementable. The appropriate programming abstractions are a hierarchy of scalable distributed dynamic grids and a set of operations on this grid hierarchy. The operations include creation, partitioning, computations on the grid such as stencil operations, communication among grid partitions at a single level and communication among grids at different levels of the hierarchy.

2. A set of distributed dynamic data-structures that support the implementation of the abstractions in parallel execution environments and preserve efficient execution while providing transparent distribution of the grid hierarchy across processing element execution environment.

The fundamental programming abstractions provided by the infrastructure are:

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

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

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

Details about the abstractions using listed above and the syntax of the programming interface provided by `DAGH` can be found in [5].

3.2 `DAGH` Multigrid Support

The `DAGH` infrastructure has been enhanced to support for parallel multigrid at each level of the AMR grid hierarchy. The idea is to enable the user to exploit multigrid techniques to update grid-functions at a particular level of the AMR hierarchy. `DAGH` enables the user to setup a distributed multigrid hierarchy of successively coarser grids over each AMR level. Coarsening can be restricted to a particular dimension (as required by line-multigrid techniques). The coarsest level of the multigrid hierarchy depends on the current decomposition of the AMR level. Once the multigrid hierarchy has been setup, each level of the hierarchy can be indexed using an additional index that specifies the multigrid level. All operators defined on the AMR hierarchy can be used for the multigrid hierarchy using the multigrid level index.

`DAGH` multigrid support is illustrated below using sample code from a 2-D parallel multigrid solver The discussion below assumes familiarity with the `DAGH` programming interface. `GH` is an instance of the GridHierarchy Class and is used to denote the current AMR hierarchy. `U` is an instance of the GridFunction class and denotes an application variable defined on `GH`. `t` is the current time-step.

3.2.1 Setting up the Multigrid Hierarchy

The multigrid hierarchy at level `lamr` of the AMR hierarchy is setup in two steps:

1. Identify the number of the multigrid levels that can be created. This depends on the refinement of the AMR level and the current decomposition of the AMR grid at that level. The number of multigrid levels is returned by the `MultiGridLevels` call:

   ```
   mglevels = MultiGridLevels(GH, lamr);
   ```

2. Setup data-structures and allocate data for the multigrid levels. This consists of laying successively coarser levels over `lamr`. As mentioned above coarsening can be performed in every dimension or along a specified axis. The related calls are:

   ```
   SetupMultiGrid(U,t,lamr,mglc,mglf);
   SetupMultiGrid(U,t,axis,lamr,mglc,mglf);
   ```

   where `mglc` is the index of the coarsest multigrid level and `mglf` is the index of the finest multigrid level; `mglf`$-$`mglc`$+1=$ `mglevels`. `axis` specifies the direction for a one-dimensional coarsening.

3.2.2 Indexing the Multigrid Hierarchy

Multigrid level `mgl` of component grid `c` at level `lamr` of the AMR hierarchy can be indexed as follows:

```
U(t,lamr,c,mgl)
```

A similar indexing interface (using a combination of the AMR and multigrid levels) is defined to enable most AMR operators to be applied to each multigrid level. For example, `DAGH` reduction operators can be applied to levels of the multigrid hierarchy as follows:

```
Sum(U,t,lamr,mgl)
Product(U,t,lamr,mgl)
MaxVal(U,t,lamr,mgl)
MinVal(U,t,lamr,mgl)
Norm2(U,t,lamr,mgl)
```

3.2.3 Synchronizing Ghost Regions

Ghost regions around local partitions of the distributed multigrid hierarchy can be updated or synchronized using the following operators:

```
ReadGhostRegion(U,t,lamr,mgl);
WriteGhostRegion(U,t,lamr,mgl);
Sync(U,t,lamr,mgl);
```

Ghost region updates can be limited to a particular axis and direction:

```
ReadGhostRegion(U,t,lamr,mgl,axis,dir);
WriteGhostRegion(U,t,lamr,mgl,axis,dir);
Sync(U,t,lamr,mgl,axis,dir);
```

where `axis` is one of `X_AXIS, Y_AXIS, Z_AXIS`, and `dir` can be `FORWARD, BACKWARD, or BOTH`.

3.2.4 Releasing Multigrid Storage

Storage associated with the multigrid hierarchy at AMR level `lamr` can be released as follows:

```
ReleaseMultiGrid(U,t,lamr);
```

3.3 Multigrid Performance Issues

As pointed out in Section2.4, a key performance issue in the implementation parallel multigrid applications is balancing communication and computation at the coarse levels. As one moves to coarser levels, the computational works successively decreases but the number of intra-level communications remains the same. One way to the overcome communication time/computation time imbalance is to replicate computations at the coarsest levels. Replication of multigrid levels within an AMR framework however will require dynamic gather and scatter communications which can in turn offset the advantages of the replication. The current implementation of the `DAGH` infrastructure concentrates on optimizing the decomposition of the AMR hierarchy and does not perform any optimizations on the multigrid hierarchy. We are, however considering an incremental gather scheme that incrementally replicates coarsest multigrid levels. In this scheme, data at the coarse multigrid levels is piggy-backed on ghost communication at finer multigrid levels and incrementally distributed to the relevant processors. Total replication is completed only when the depth of the multigrid hierarchy is greater than a threshold which depends on the current distribution of the AMR grid.

4 Applications

4.1 2D Correction Scheme

As an illustration of the use of the `DAGH` support in an acutal multigrid application, we reproduce in this section most of the source code needed to solve the problem described in 2.2 and pseudo-coded in 2.3. Section 4.1.1 contains the `C++` code while most of the important `FORTRAN` routines are listed in 4.1.2. Finally, section 4.1.3 lists the `Makefile` and typical output from the application which was executed on a cluster of workstations.

4.1.1 C++ code

The following code implements the top levels of the application. Contents include:

- **Routines:** `main, mg2dlcs, smooth`
- **Header File:** `mgfortran.h`

The header file, which is included by all three of the `C++` routines defines the interface from the `C++` to the `FORTRAN` routines.

```
/* ============================================================================
   main():  Implements 'Procedure CSMain'
============================================================================ */
#include "DAGH.h"
#include "mgfortran.h"
extern DOUBLE mg2dlcs(
             GridFunction(2)<DOUBLE> &,
             GridFunction(2)<DOUBLE> &,
             GridFunction(2)<DOUBLE> &,
             DOUBLE const,
             INTEGER const, INTEGER const,
             INTEGER const, INTEGER&,
             INTEGER const, INTEGER const,
             DOUBLE&);
#define DIM 2
#define MaxLev 1
#define TRACE 0

void main(INTEGER argc, char *argv[])
{
   MPI_Init( &argc, &argv );
   DOUBLE xmin, xmax, ymin, ymax;
   INTEGER lamr;
```

```
INTEGER nx,ny;
DOUBLE hamr;
DOUBLE epsilon;
INTEGER ncycle,pre,pst;
                        /* Determine problem parameters from an input file */
f_readinput(&xmin, &xmax,
            &ymin, &ymax,
            &lamr,
            &nx, &ny, &hamr,
            &ncycle, &pre, &pst);
                                              /* Set up the Grid Structure */
INTEGER shape[DIM];
double bb[2*DIM];
shape[0] = nx-1;
bb[0] = xmin;
bb[1] = xmax;
shape[1] = ny-1;
bb[2] = ymin;
bb[3] = ymax;
                                                  /* Set up DAGH structure */
GridHierarchy GH(DIM,NON_CELL_CENTERED,MaxLev);
SetBaseGrid (GH,bb,shape);
ComposeHierarchy(GH);
                                          /* Set up stencil and time steps */
INTEGER t_sten = 0; INTEGER s_sten = 1;
GridFunction(DIM)<DOUBLE> u("u",t_sten,s_sten,GH);
GridFunction(DIM)<DOUBLE> rhs("rhs",t_sten,s_sten,GH);
GridFunction(DIM)<DOUBLE> res("res",t_sten,s_sten,GH);
GridFunction(DIM)<DOUBLE> ans("ans",t_sten,s_sten,GH);
GridFunction(DIM)<DOUBLE> err("err",t_sten,s_sten,GH);
INTEGER me = MY_PROC(GH);
INTEGER num = NUM_PROC(GH);
                                                           /* Local arrays */
DOUBLE *x = new DOUBLE[nx];
DOUBLE *y = new DOUBLE[ny];
INTEGER t = 0, l = 0;
                                                      /* Set up u and rhs */
forall(u,t,l,c)
   f_initial(
             FBA(u(t,l,c)), x, y,
             FDA(u(t,l,c)), FDA(rhs(t,l,c)),
             FDA(ans(t,l,c)), &hamr);
end_forall
                                                 /* delete local storage */
delete [] x;
delete [] y;
DOUBLE norm2 = (DOUBLE) 100;
DOUBLE norme = (DOUBLE) 100;
```

```
   INTEGER cycle = 0;
   INTEGER ncyc = 0;
   INTEGER ll = 0;
   INTEGER tt = 0;
   INTEGER  nl = MultiGridLevels(GH,ll);
                                                       /*  Peform ncycle Vcycles */
   for (cycle=0;cycle<ncycle;cycle++) {
      norm2 = mg2dlcs ( u, rhs, res, hamr, ll, tt,
                        nl, ncyc, pre, pst,epsilon );
                                                  /* Determine e= |u^h-u(exact)| */
      forall(u,t,l,c)
         err(t,l,c) = 0.0;
         err(t,l,c)+=  ans(t,l,c);
          err(t,l,c)-=  u(t,l,c);
      end_forall
      norme = Norm2(err,t,l);
                                                            /* Display |r| , |e| */
      if (me == 0) {
         cout << "Cycle: " << ncyc << " " << norm2 << " " << norme;
         cout << "\n" << flush;
      }
   }
  MPI_Finalize();
 }

/* ==============================================================================
   mg2dlcs():  Implements 'Procedure Vcycle'
=============================================================================== */
#include "DAGH.h"
#include "mgfortran.h"
extern DOUBLE smooth(
                     GridFunction(2)<DOUBLE> &,
                     GridFunction(2)<DOUBLE> &,
                     GridFunction(2)<DOUBLE> &,
                     DOUBLE &,
                     INTEGER const, INTEGER const, INTEGER const);
DOUBLE mg2dlcs(
               GridFunction(2)<DOUBLE> &u,
               GridFunction(2)<DOUBLE> &rhs,
               GridFunction(2)<DOUBLE> &res,
               DOUBLE const hamr,
               INTEGER const lamr, INTEGER const t,
               INTEGER const nl, INTEGER& ncycle,
               INTEGER const pre, INTEGER const pst,
               DOUBLE &epsilon )
{
   INTEGER me = MY_PROC(GH);
   INTEGER num = NUM_PROC(GH);
```

```
DOUBLE const del = 0.01;
INTEGER lf = nl-1;
INTEGER lc = 0;

if (ncycle == 0) {
   SetupMultiGrid(u,t,lamr,lc,lf);
   SetupMultiGrid(rhs,t,lamr,lc,lf);
   SetupMultiGrid(res,t,lamr,lc,lf);
}

DOUBLE *h = new DOUBLE[nl];
h[lf] = hamr;
for (INTEGER l=lf-1;l>=lc;l--) {
   h[l] = h[l+1]*2.0;
}
DOUBLE norm2 = (DOUBLE) 1.0;
DOUBLE norm = (DOUBLE) 1.0;
INTEGER pass = 0;
                                        /* Multigridding proper begin here */
for (l=lf;l>=lc;l--) {
   if (l != lf || ncycle == 0) {
      if (l == lc) {
                                       /* Solve coarse system at l = lc    */
         INTEGER cnt = 0;
         while (norm2> epsilon) {
            cnt++;
            norm = smooth(u,rhs,res,h[l],t,lamr,l);
            norm2 = Norm2(res,t,lamr,l);
            }
         }
      else {
         for (INTEGER p=1;p<=pre;p++) {
            norm = smooth(u,rhs,res,h[l],t,lamr,l);
         }
      }
   }
                                              /* Set up coarse-gird system: */
   Sync(u,t,lamr,l);
   forall(u,t,lamr,c)
      f_cmres(
               FBA(u(t,lamr,c,l)),
               FDA(u(t,lamr,c,l)), FDA(rhs(t,lamr,c,l)),
               FDA(res(t,lamr,c,l)),&h[l]);
   end_forall
   norm2 = Norm2(res,t,lamr,l);
   epsilon = del * norm2;
   Sync(res,t,lamr,l);
                                        /* now restrict the residuals .... */
```

```
      if (l != lc ) {
         forall(u,t,lamr,c)
            f_restrict(FBA(u(t,lamr,c,l)),FBA(u(t,lamr,c,l-1)),
                       FDA(res(t,lamr,c,l)),FDA(rhs(t,lamr,c,l-1))) ;
            u(t,lamr,c,l-1) = 0.0;
          end_forall
       }
   }
                              /* now cycle from the coarse to fine grid ... */
   for (l=lc+1;l<=lf;l++) {
                                     /* now prolongate up the correction */
      forall(u,t,lamr,c)
         f_prolong(FBA(u(t,lamr,c,l)),FBA(u(t,lamr,c,l-1)),
         FDA(res(t,lamr,c,l)),FDA(u(t,lamr,c,l-1))) ;
      end_forall
                                        /*     add this correction to u */
      forall(u,t,lamr,c)
         u(t,lamr,c,l) += res(t,lamr,c,l);
      end_forall
                                        /*        smooth again .......  */
      for (INTEGER p=1;p<=pst;p++) {
         norm = smooth(u,rhs,res,h[l],t,lamr,l);
      }
   }
                               /* return the norm of the residuals ....... */
   l = lf;
   Sync(u,t,lamr,l);
   forall(u,t,lamr,c)
      f_cmres(
              FBA(u(t,lamr,c,l)),
              FDA(u(t,lamr,c,l)), FDA(rhs(t,lamr,c,l)),
              FDA(res(t,lamr,c,l)),&h[l]);
   end_forall
   norm2 = Norm2(res,t,lamr,l);
   epsilon = del * norm2 ;
   ncycle++;
   return (norm2);
}

/* ======================================================================
   smooth():  Implements 'Procedure Relax'
===================================================================== */

#include "DAGH.h"
#include "mgfortran.h"

DOUBLE smooth(
             GridFunction(2)<DOUBLE> &u,
```

```
            GridFunction(2)<DOUBLE> &rhs,
            GridFunction(2)<DOUBLE> &res,
            DOUBLE &h,
            INTEGER const t, INTEGER const lamr, INTEGER const l)
{
   INTEGER pass = 0;
   DOUBLE norm2 = (DOUBLE) 1.0;

   Sync(u,t,lamr,l);

   pass = 1;
   forall(u,t,lamr,c)
      f_relax (
      FBA(u(t,lamr,c,l)),
      FDA(u(t,lamr,c,l)), FDA(rhs(t,lamr,c,l)),
      FDA(res(t,lamr,c,l)),
      &h, &pass, &norm2);
   end_forall

   Sync(u,t,lamr,l);

   pass = 2;
   forall(u,t,lamr,c)
      f_relax (
      FBA(u(t,lamr,c,l)),
      FDA(u(t,lamr,c,l)), FDA(rhs(t,lamr,c,l)),
      FDA(res(t,lamr,c,l)),
      &h, &pass, &norm2);
   end_forall
   return(norm2);
}

/* ==============================================================================
   mgfortran.h():  Defines FORTRAN interfaces.
============================================================================== */

/* Some macros  */
#ifndef BBOX_INTERFACE_LONG
#define BBOX_INTERFACE_LONG       int const *, int const *, \
                                  int const *, int const *, int const *
#endif

#ifndef BBOX_INTERFACE
#define BBOX_INTERFACE            int const *, int const *, int const *
#endif

#ifndef FORTRAN_INTERFACE
#define FORTRAN_INTERFACE(T)     T *, int const *, int const *, \
```

```
                                  int const *
#endif

#ifndef FORTRAN_DATA_INTERFACE
#define FORTRAN_DATA_INTERFACE(T)     T *
#endif

#define BI    BBOX_INTERFACE
#define BIL    BBOX_INTERFACE_LONG
#define FI    FORTRAN_INTERFACE
#define FDI    FORTRAN_DATA_INTERFACE

           /* Interface from the C++ routines to the fortran wrapper routines */

#define f_readinput FORTRAN_NAME(mg_readinput_, MG_READINPUT, mg_readinput)
extern "C" {
void f_readinput( DOUBLE *, DOUBLE *, DOUBLE *, DOUBLE *, INTEGER *,
                  INTEGER *,INTEGER *, DOUBLE *,
                  INTEGER *,INTEGER *, INTEGER * );
}

#define f_initial FORTRAN_NAME(mg_init_, MG_INIT, mg_init)
extern "C" {
void f_initial( BI, DOUBLE *, DOUBLE *,
                FDI(DOUBLE), FDI(DOUBLE), FDI(DOUBLE), DOUBLE *);
}

#define f_relax FORTRAN_NAME(mg_relax_, MG_RELAX, mg_relax)
extern "C" {
void f_relax( BI, FDI(DOUBLE), FDI(DOUBLE), FDI(DOUBLE),
              DOUBLE *, int *, DOUBLE *);
}

#define f_cmres FORTRAN_NAME(mg_cmres_, MG_CMRES, mg_cmres)
extern "C" {
void f_cmres( BI, FDI(DOUBLE), FDI(DOUBLE), FDI(DOUBLE), DOUBLE * );
}

#define f_restrict FORTRAN_NAME(mg_restrict_, MG_RESTRICT, mg_restrict)
extern "C" {
void f_restrict( BI, BI, FDI(DOUBLE), FDI(DOUBLE));
}

#define f_prolong FORTRAN_NAME(mg_prolong_, MG_PROLONG, mg_prolong)
extern "C" {
void f_prolong( BI, BI, FDI(DOUBLE), FDI(DOUBLE));
}
```

4.1.2 FORTRAN code

The routines in the first part of this section represent an intermediate layer between the C++ code presented abover, and lower-level FORTRAN code, given later in the section. It is in the lower-level routines that the actual grid-function manipulations/cacluations are performed. A key distinction between the "parallel" routines listed here and their "serial" counterparts is that the actual routines have additional arguments, generally called lb and ub for "lower bounds" and "upper bounds" repsectively. These arrays specify which rectangular subset of the grid-function's index space is to be operated on by a particular invocation of the routine.

- **Routines:** mg_init, mg_relax, mg_cmres, mg_restrict, mg_prolong, mg_readinput

```
c-----------------------------------------------------------------------------
      subroutine mg_init( lb, ub, shape, x, y, u, rhs, uxct, h )
      implicit none
      integer lb(2), ub(2), shape(2)
      real*8  u( shape(1), shape(2) )
      real*8  rhs( shape(1), shape(2) )
      real*8  uxct( shape(1), shape(2) )
      real*8  x( shape(1)),y(shape(2) )
      real*8  xmin, xmax, ymin, ymax, h
      real*8 lb1, lb2
      integer l

      xmin = -1.0d0
      ymin = -1.0d0
      xmax =  1.0d0
      ymax =  1.0d0
      lb1 = h*lb(1) + xmin
      lb2 = h*lb(2) + ymin
      call convert( shape(1), lb1, h, x )
      call convert( shape(2), lb2, h, y )
      call set_up_rhs( shape(1), shape(2), x, y, rhs )
      call set_up_u0( shape(1),shape(2), x, y, u )
      call set_up_uxct( shape(1), shape(2), x, y, uxct )

      return
      end
c-----------------------------------------------------------------------------
      subroutine mg_relax( lb, ub, shape, u, rhs, res, h, pass, nrm )
      implicit none
      integer lb(2), ub(2), shape(2)
      real*8  u(shape(1),shape(2))
      real*8  rhs(shape(1), shape(2))
      real*8  res(shape(1), shape(2))
      real*8  h, nrm
      integer pass

      call relax( u, rhs, res, shape(1), shape(2), h, lb, pass, nrm )
```

```
      return
      end
c-----------------------------------------------------------------------------
      subroutine mg_cmres( lb, ub, shape, u, rhs, res, h )
      implicit none
      integer lb(2), ub(2), shape(2)
      real*8  u( shape(1), shape(2) )
      real*8  rhs( shape(1), shape(2) )
      real*8  res( shape(1), shape(2) )
      real*8  h, norm

      call cmres( res, u, rhs, shape(1), shape(2), h )

      return
      end
c-----------------------------------------------------------------------------
      subroutine mg_restrict( lbf, ubf, shapef, lbc, ubc, shapec, uf,uc)
      implicit none
      integer lbf(2), ubf(2), shapef(2)
      integer lbc(2), ubc(2), shapec(2)
      real*8  uf( shapef(1), shapef(2) )
      real*8  uc( shapec(1), shapec(2) )

      call restrict( lbf, ubf, shapef, lbc, ubc, shapec, uf, uc )

      return
      end
c-----------------------------------------------------------------------------
      subroutine mg_prolong( lbf, ubf, shapef, lbc, ubc, shapec, uf, uc)
      implicit none
      integer lbf(2), ubf(2), shapef(2)
      integer lbc(2), ubc(2), shapec(2)
      real*8  uf( shapef(1), shapef(2) )
      real*8  uc( shapec(1), shapec(2) )

      call prolong(lbf, ubf,shapef,lbc,ubc,shapec,uf,uc)

      return
      end
c------------------------------------------------------------------------------
      subroutine mg_readinput( xmin, xmax, ymin, ymax, l,
     &                         nx, ny, h, ncycle, pre, pst )

      integer nx, ny, nz
      real*8 xmin, xmax, ymin, ymax
      real*8 h, hc
      integer l, ncycle, pre, pst
```

```
      xmin =  0.0d0
      ymin =  0.0d0
      xmax =  1.0d0
      ymax =  1.0d0
      hc = 1.0d0

      open(88,file='mg.input')
      read(88,*) l
      read(88,*) ncycle
      read(88,*) pre, pst
      close(88)

      h = hc / 2**(l-1)
      nx = 1 + (xmax-xmin)/ h
      ny = 1 + (ymax-ymin)/ h

      return
      end
```

The lower-level FORTRAN routines follow. With the exception of the addition of the bounds arguments, and some additional code needed to consistently deal with bounds, these routines are identical to their serial counterparts.

- **Routines:** relax, cmres, restrict, prolong, d2lbs_s, convert, set_up_u_xct, set_up_rhs, set_up_u0

```
c-----------------------------------------------------------------------
c     relax: Implements 'Procedure Relax_rb'
c
c     Does one-step Newton Point-GS relaxation and returns norm of
c     running ('instantaneous') residuals---checkerboard ordering.
c-----------------------------------------------------------------------
      subroutine relax( u, rhs, res, nx, ny, h, lb, rb, nrm )
      implicit none
      integer nx, ny
      real*8  u( nx, ny ), rhs( nx, ny ), res( nx, ny )
      real*8  h, nrm
      integer rb, lb(2)
      real*8   hm2, m4hm2, jacel
      integer  i, j, isw, n updates

      hm2 = 1.0d0 / (h * h)
      m4hm2 = -4.0d0 * hm2
      jacel = m4hm2

      isw = 2-mod(lb(2),2)
      if (mod(lb(1),2) .eq. 1) isw = 3-isw
```

```
      nrm = 0.0d0
      nupdates = 0
      if (rb .eq. 2) isw = 3 - isw

      do j = 2 , ny - 1
         do i = isw + 1 , nx - 1, 2

            res(i,j) = hm2 * (u(i+1,j) + u(i-1,j) +
     *                        u(i,j+1) + u(i,j-1) -
     *                        4.0d0 * u(i,j))
     *                        - rhs(i,j)

            u(i,j) = u(i,j) - res(i,j) / jacel
            nrm = nrm + res(i,j) * res(i,j)
            nupdates = nupdates + 1
         end do
         isw = 3 - isw
      end do
      nrm = sqrt(nrm/nupdates)

      return
      end
c-----------------------------------------------------------------------
c     cmres: Implements 'Procedure Compute_NegResiduals
c
c     Computes negative of the residuals with residuals defined to be
c     0 on boundary.
c-----------------------------------------------------------------------
      subroutine cmres( res, u, rhs, nx, ny, h )
      implicit none
      real*8  u(nx,ny), rhs(nx,ny), res(nx,ny)
      integer nx, ny
      integer i, j
      real*8  h, hm2

      hm2 = 1.0d0 / ( h * h )
      do j=2,ny-1
         do i=2,nx-1
            res(i,j) =rhs(i,j)- hm2 * (u(i+1,j) + u(i-1,j) +
     *                                 u(i,j+1) + u(i,j-1) -
     *                                 4.0d0*u(i,j))
         end do
      end do
      call d2lbs_s( res, nx, ny, 0.0d0 )

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

```
c     restrict: Implements 'Procedure Restrict'
c
c     2:1 half-weighted restriction with boundary injection.
c-----------------------------------------------------------------------
      subroutine restrict( lbf, ubf, shapef, lbc, ubc, shapec, uf, uc )
      implicit none

      integer   lbf(2), ubf(2), shapef(2)
      integer   lbc(2), ubc(2), shapec(2)
      real*8    uc( shapec(1), shapec(2) )
      real*8    uf( shapef(1), shapef(2) )
      integer   nxc, nyc, nxf, nyf
      integer   ic, jc, if, jf
      integer   lbfu(2), lbcu(2), ubfu(2), ubcu(2), start(2), stop(2)
      real*8    half, eighth
      parameter  ( half = 0.5d0, eighth = 0.125D+00 )

      nxc = shapec(1)
      nyc = shapec(2)
      nxf = shapef(1)
      nyf = shapef(2)
C determine where to start the restrictions
      do ic=1,2
         if (lbc(ic) .ne. lbf(ic)) then
            lbfu(ic) = 2
            lbcu(ic) = 2
            start(ic) = lbcu(ic)
         else
            lbfu(ic) = 1
            lbcu(ic) = 1
            start(ic) = lbcu(ic) + 1
         end if
         if (ubc(ic) .ne. ubf(ic)) then
            ubfu(ic) = shapef(ic) - 1
            ubcu(ic) = shapec(ic) - 1
            stop(ic) = ubcu(ic)
         else
            ubfu(ic) = shapef(ic)
            ubcu(ic) = shapec(ic)
            stop(ic) = ubcu(ic) - 1
         end if
      end do
C start the restrictions of the interior points using half weighted averages
      do jc = start(2), stop(2)
         jf = 2 * jc - lbcu(2)
         do ic = start(1),stop(1)
            if = 2 * ic - lbcu(1)
            uc(ic,jc) = half*uf(if,jf) + eighth * (
```

```
     .                  uf(if+1,jf) +uf(if-1,jf)+
     .                  uf(if,jf+1) +uf(if,jf-1) )
         end do
      end do
C see if we have to restrict the outer boundary .....
      if (lbf(1).eq.lbc(1)) then
         do jc = lbcu(2) , ubcu(2)
            jf = 2*jc - lbcu(2)
            uc(1,jc)    = uf(1,jf)
         end do
      end if

      if (lbf(2).eq.lbc(2)) then
         do ic = lbcu(1) , ubcu(1)
            if = 2*ic - lbcu(1)
            uc(ic,1)    = uf(if,1)
         end do
      end if

      if (ubf(1).eq.ubc(1)) then
         do jc = lbcu(2) , ubcu(2)
            jf = 2*jc - lbcu(2)
            uc(nxc,jc)    = uf(nxf,jf)
         end do
      end if

      if (ubf(2).eq.ubc(2)) then
         do ic = lbcu(1) , ubcu(1)
            if = 2*ic - lbcu(1)
            uc(ic,nyc)    = uf(if,nyf)
         end do
      end if

      return
      end
c-----------------------------------------------------------------------
c     prolong: Implements 'Procedure Prolong'
c
c     2:1 bi-linear interpolation.
c-----------------------------------------------------------------------
      subroutine prolong( lbf, ubf, shapef, lbc, ubc, shapec, uf, uc )
      implicit none

      integer   lbf(2),ubf(2),shapef(2)
      integer   lbc(2),ubc(2),shapec(2)
      real*8    uc(shapec(1),shapec(2))
      real*8    uf(shapef(1),shapef(2))
```

```
      integer   lbfu(2), lbcu(2), ubfu(2), ubcu(2)

      integer   nxc, nyc, nxf, nyf
      integer   ic, jc, iff, jf


      nxc = shapec(1)
      nyc = shapec(2)

      nxf = shapef(1)
      nyf = shapef(2)

c figure out where the prolongation should start
      do ic=1,2
         if (lbc(ic) .ne. lbf(ic)) then
            lbfu(ic) = 2
            lbcu(ic) = 2
         else
            lbfu(ic) = 1
            lbcu(ic) = 1
         end if
         if (ubc(ic) .ne. ubf(ic)) then
            ubfu(ic) = shapef(ic) - 1
            ubcu(ic) = shapec(ic) - 1
         else
            ubfu(ic) = shapef(ic)
            ubcu(ic) = shapec(ic)
         end if
      end do

c injection
      do jc = lbcu(2), ubcu(2)
         jf = 2*jc - lbcu(2)
         do ic = lbcu(1), ubcu(1)
            iff = 2*ic - lbcu(1)
            uf(iff,jf) = uc(ic,jc)
        end do
      end do

c linear interpolation
      do jc = lbcu(2), ubcu(2)
         jf = 2*jc - lbcu(2)
         do iff = lbfu(1)+1, ubfu(1)-1,2
            uf(iff,jf) = 0.5d0*(uf(iff-1,jf)+uf(iff+1,jf))
        end do
      end do

      do jf=lbfu(2)+1, ubfu(2)-1,2
```

```
         do iff = lbfu(1), ubfu(1)
            uf(iff,jf) = 0.5d0*(uf(iff,jf-1)+uf(iff,jf+1))
         end do
      end do

      return
      end

c-----------------------------------------------------------------------
c     d2lbs_s:
c
c     Loads the 'boundary' of a rank-2 array 'a' with the scalar 'val'
c-----------------------------------------------------------------------
      subroutine d2lbs_s( a, nx, ny, val )
      implicit none

      integer nx, ny
      real*8  a( nx, ny ), val
      integer i,j

      do i = 1 , nx
         a(i,1) = val
         a(i,ny) = val
      end do

      do j = 1 , ny
         a(1,j) = val
         a(nx,j) = val
      end do

      return
      end

c----------------------------------------------------------------------
c     convert:
c
c     Computes world coordinate vector, 'x', from minimum value
c     'xmin' and mesh spacing 'h'.
c----------------------------------------------------------------------
      subroutine convert( nx, xmin, h, x )
      implicit none
      integer  nx
      real*8   x( nx ), h, xmin
      integer i

      do i=1,nx
           x(i) = xmin + h*(i-1)
      end do
```

```
      return
      end

c-----------------------------------------------------------------------
c     set_up_uxct:
c
c     Computes exact solution, 'uxct'
c-----------------------------------------------------------------------
      subroutine set_up_u( nx, ny, x, y, uxct )
      implicit none

      integer nx, ny
      real*8  x( nx ), y( ny ), uxct( nx, ny )
      real*8  pi
      integer i,j

      pi = 4.0d0*atan(1.0d0)

      do j = 1 , ny
         do i = 1 , nx
            uxct(i,j) = sin(pi*x(i))*sin(pi*y(j))
         end do
      end do

      return
      end
c-----------------------------------------------------------------------
c     set_up_rhs:
c
c     Defines 'rhs' (right hand side of original equation, or 'f')
c-----------------------------------------------------------------------
      subroutine set_up_rhs( nx, ny, x, y, rhs )
      implicit none

      integer nx, ny
      real*8  x( nx ),y( ny ), rhs( nx, ny )
      integer i,j
      real*8  pi

      pi = 4.0d0 * atan(1.0d0)
      do j= 1, ny
         do i= 1, nx
            rhs(i,j) = -2.0d0 * pi*pi*sin(pi*x(i))*sin(pi*y(j))
         end do
      end do

      return
```

```
      end

c-----------------------------------------------------------------------
c     set_up_u0:
c
c     Initializes unknown grid-function, 'u'.
c-----------------------------------------------------------------------
      subroutine set_up_u0( nx, ny, x, y, u )
      implicit none

      integer  nx,ny
      real*8   x( nx ), y( ny )
      real*8   u( nx, ny )
      integer  i, j

      do j= 1, ny
         do i= 1, nx
            u(i,j) = 1.0d0
         end do
      end do

      return
      end
```

4.1.3 Makefile and Sample Execution Trace

```
#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 = /u3/bbh/DAGH
# Include architecture specific makefile
include $(DAGH_HOME)/make.defn
```

```
##########################################################################
# Application Targets:                                                   #
#                                                                        #
#  mg2dlcs                                                               #
##########################################################################
# Sources & Objects
APPSRC =  mg_util.f   mgfor.f
APPOBJ = $(APPSRC:.f=.o)
# Application specific Flags
C++APPFLAGS =-O2
CAPPFLAGS = -O2
F77APPFLAGS = -O2
F90APPFLAGS = -O2
# Application specific libraries
APPLIB =
EXEC =      mg2dlcs
mg2dlcs:      $(APPOBJ) mg.o mgmain.o smooth.o
      $(RM) mg2dlcs
      $(C++LINK) $(C++FLAGS) -o $@ \
      $(APPOBJ) mg.o mgmain.o smooth.o $(APPLIB) $(LDLIBS)
clean:
      $(RM) *.o core $(EXEC); $(RM) -r $(REPOSITORY
```

A sample output showing an execution trace of the application of 5 CS V-cycles (`pre` = `pst` = 2) to a 33×33 problem follows:

```
% more mg.input
5
5
2 2 2 2
% mpi mg2dlcs
Cycle: 1 0.608413 0.995359
Cycle: 2 0.048437 1.00113
Cycle: 3 0.00311591 1.00116
Cycle: 4 0.000199787 1.00115
Cycle: 5 1.1706e-05 1.00115
```

5 Acknowledgements

This work was supported by NSF grant PHY9310083 to R.A. Matzner and NSF PHY9318152 (ARPA supplemented) to the Binary Black Holes Grand Challenge Alliance.

References

[1] A. Brandt, “Multi-level adaptive solutions to boundary-value problems.”, *Math. Comp.*, **31**, 333-390, (1977)

[2] A. Brandt. "Guide to multigrid development.” in *Lecture Notes in Mathematics*, **960**, *(Multigrid Methods)*, W. Hackbusch and U. Trottenberg, eds., , 220-312, (1982)

[3] M. J. Berger and J. Oliger, *J. Comp. Phys*, **53**, 484 (1984)

[4] M. Parashar and J. C. Browne, “An Infrastructure for Parallel Adaptive Mesh-Refinement Techniques”, `http://godel.ph.utexas.edu:/Members/parashar/Papers/daghrep.ps`, (1995)

[5] M. Parashar and J. C. Browne, “DAGH: A Data-Management Infrastructure for Parallel Adaptive Mesh-Refinement Techniques: Preliminary Users Guide”, `http://godel.ph.utexas.edu:/Members/parashar/Papers/uref.ps`, (1995)