The Distributed Adaptive Mesh Refinement Code

Steven L. Liebling
Department of Physics, Long Island University – C.W. Post Campus, Brookville, New York 11548
(Dated: February 8, 2007)

This manual details how to configure, run, and use the distributed adaptive mesh refinement code.

I. INTRODUCTION

This code solves evolution equations using adaptive mesh refinement (AMR) and domain decomposition on multiple processors using Message Passing Interface (MPI). For details on AMR, see Berger and Oliger.

II. IMPLEMENTATION

The code is written in Fortran for simplicity and ease of debugging as compared to C or C++. However, it tries to emulate certain aspects of object oriented coding as might be found in a C++ code. From this perspective, the code tries to respect certain objects. Most fundamental are fields which store the values of fields over a domain. Grids contain fields and associated other data such as coordinates. Levels consist of all fields at a given resolution. Hence, level zero consists of just the coarse grid (or all the grids into which the coarse grid may be domain decomposed for distribution to all the processors), while level one consists of all children of that coarse grid.

The code is written to allow for various projects. The simplest, and perhaps oldest, project is the semilinear wave equation, called `hyperSemiL`. This is a good project to emulate and is where I generally test new features. It resides in `had/src/hyperSemiL`. Project-specific code is found in a project's directory.

The code is distributed using Message Passing Interface (MPI) in such a way that grids are sent to the various processors.

III. RUNNING

A. Basics

When one compiles `had`, you specify the name of the executable which gets placed in `had/bin`. Say we call the executable `had`. Then you would run it (on a single processor by typing

```
had id0semi
```

where `id0semi` is a parameter file which controls what the code does. The analogous call to run on some number of processors would be

```
mpirun -np 10 had id0semi
```

to run on 10 processors. These directions may be a bit different with different implementations of MPI and different queues.

B. Understanding Code Output

C. The Initial Data File

The code needs a single initial data file. The goal of having a single file was to be able to reduce the complexity of ensuring repeated runs had the same parameters. Auxiliary scripts could suffice to take different scripts and process them into a single one if the file gets too big or awkward.

There are three sets of parameters, all defined in respective `setup` files. These sets consist of parameters from the `amr` side, the `hyper` side, and the project's side. I'll describe here the `amr` parameters and maybe a few from the `hyper` side. It is presumed that you know about a given project's parameters (or else can refer to some other documentation).

Note that I play around with the file `id` often so its state may not be a good one to first play with the code. So, I have added to the distribution a sample initial data file `id_sample` which should always be a representative data file one can start from.

Parameters

	Coarse Level Parameters
`nx0, ny0, nz0`	number of points in each direction for the coarse level
`nt0`	number of time steps to be taken by the code (in terms of coarse level steps)
`minx0/maxx0`	coordinate bounds of coarse level; constrained by grid spacings in each direction must be same
	AMR Time Integration
`lambda`	ratio of $\Delta t/\Delta x$ (loosely the Courant factor)
`num_evol_iters`	number of iterations for time step algorithm (default is 3 for RK3)
`ghost_width`	width (in points) of boundary region on child which gets set by parent level
`bound_width`	width (in points) of overlap region which gets sync'ed among processors (default == 3)
`linearbounds`	type of interpolation for child boundary data: > 0 uses *linear* interpolation $= 0$ uses *cubic* interpolation
	Refinement Parameters
`refine_factor`	ratio of resolutions between parent and child levels (must be even)
`refine_period`	number of evolution steps before re-refinement occurs. For FMR, set to large number
`allowedl`	number of levels allowed (0 for unigrid); maximum set in `glob.inc`
`ethreshold`	threshold value for error for a point to be flagged for refinement
`shadow`	binary toggle for whether to use the self shadow hierarchy (== 1 turns on); currently requires *linearbounds* == 0
`minefficiency`	target efficiency to which clustering algorithm strives (number of flagged points / total number of pts)
`mindim`	minimimum linear number of points in any direction for a grid (used by clusterer and domain decomposer)
`refine_level_X`	for each level "X", specifies either: < 0 creates FMR level at X+1 about the center with linear dimensions 1/refine_level_X > 0 multiplies the error for this level (default is **1**
	Output Parameters
`output_dim`	Allows for dimensionally reduced output such as a 2D slice through middle of coarse level **1,2, or 3** picks one dimension to be output (1D, 2D, or 3D) **4,5, or 6** corresponds to sum of dimensions to be output (e.g. 1D and 3D = 4) **7** outputs 1D and 2D dimensional SDFs
`output_style`	determines how data is output: **1** data at a certain level is output to level-specific SDF file (used for domain decomposition mode) **4** data at this level and all finer levels are output to level-specific file (useful for AMR)
`output_XXXX`	binary flag (0–off or 1–on) which determines whether the field "XXXX" is output
`output_level_X`	for each level "X", the number of evolution steps before outputting. If 0, no output for that level
	Checkpoint Parameters
`chkpt_period`	determines frequently (in coarse level time steps) to save state (default is **0** for off)
`chkpt_readstate`	Toggles whether to read state from file. Default is off (**0**)
	Elliptic Solver
	Excision

TABLE I: Description of parameters for the `AMR` side of `had`.

1. Unigrid Mode

Set `allowedl` to zero. The value of `output_style` won't matter.

2. AMR (serial or parallel

Set `allowedl` to any value (presumably max). Run as described in Section III A. Likely want `output_style` equal to four, unless debugging. A refinement factor of two generally works the best. It's not as efficient as a higher value, but keeping things smooth seems best for two. I've used a refinement period of 4.

3. Domain Decomposition

To get the code to run in this mode is not obvious. One must set `refine_period` to a value of 999 or greater because you don't want any re-refinement. Likewise, you only want one level of refinement, `allowedl` = 1. With these two set, the code automatically clusters such that the refined level covers the entire domain and that there are the same number of subgrids as there are processors. To control the resolution, one then adjusts the refinement factor.

4. Checkpointing

The checkpoint routine outputs files with names alternating between 'hello' and 'goodbye' so that there are potentially two checkpointed states to recover from. When `chkpt_readstate` is set to (**1**) the routine reads the 'hello' state; when set to (**2**), it reads the 'goodbye' state.

The restarted code needs to run on the same number of processors as the original checkpoint source. Modifications to the parameter file when `chkpt_readstate` is not (**0**) have no effect with the exception of the `nt0` parameter.

CAVEATS: sdf files from the original run will be written-over by the restarted state. Move original sdf output to a different directory to avoid undefined output results.

D. Shadow Hierarchy

A key ingredient to the B&O algorithm is knowing where refinement is needed. This is accomplished by determining an estimate of the error in the solution for each location and time.

The simplest such solution is by using some function of the evolved fields, such as a density or gradient. To do this, you can simply set `gr_error` in the project's `comp_amr_error()` routine.

B&O specify another way, called truncation error estimation (TRE). To get an error estimation of the truncation error, one takes a two time steps on a given grid and compares the obtained values with values found by taking a single step on a grid with half the resolution. This difference is indicative of the truncation error.

Where does this coarse grid come from? Well, in their algorithm, you make it, take the step, and then you're done with it. Instead, you can simply continually maintain a separate hierarchy from the one on which you're computing your solutions which differs by a factor of two in resolution. Then, to compute the TRE, you simply subtract the two hierarchies at any given time. This is a `shadow hierarchy`.

But do we really need a separate hierarchy? After all, say you want the error on a fine grid. You want to compare your solution to a coarser one, but instead of going to the shadow, you can go to the parent grid. Your solution hierarchy then serves as your shadow hierarchy as well in what is called a `self-shadow hierarchy` (as per Frans).

So how does `had` implement a self-shadow?

If you want unigrid, you still specify `allowedl=0`. If `allowedl` is greater than 0, then level 0 is essentially only shadow. Your coarsest level **is** the level 1. Hence, if `shadow=1`, then level 1 is forced to cover the entire computational domain. So it doesn't make sense to have `allowedl=1` if `shadow` is turned on.

So to use the shadow hierarchy, you would specify `shadow=1`, `allowedl > 1`, and `linearbounds=0`. I would also set `output_level_0 :=0` and `output_level_1 := 1` so that you look at output on level 1.

The only tricky is the initial data. As mentioned, levels 0 and 1 simply get created using the project's initial data routines to set their values. Then one step on level 0 and two steps on level 1 are taken. Subtracting, you get an error. The advanced solutions are discarded (and the levels are reinitialized). If level 2 needs to be created, `had` creates it. Do we need a level 3? A step on level 1 and two steps on level 2 are now taken. This process continues.

E. Compile Time Configuration

As can be seen in the Makefile, certain environment variables must be set in order to compile. The code requires some installation of MPI and `bbhutil` which is installed with the `rnpl` distribution. In the future, the code may require `rnpl` itself and certainly the current version of the unigrid code requires it. Generally, these are set in the `.cshrc`:

LSV — On some systems, the `bbhutil` library links to the `sv` library if installed. If `sv` is not installed, then set this to blank otherwise set it to "-lsv".

IMPI — Sometimes the MPI installation has include files in a certain spot, in which case it might set as "-I/usr/local/vmi/mpich/include". Otherwise you can leave it blank.

LMPI — the MPI libraries might have different names depending on the platform or distribution. For MPICH, it's generally set as "-lmpich".

MPIF77 — MPICH creates a wrapper for f77 which helps with the libraries. This is the name of this. Examples include "mpif77" and "/usr/local/mpich/bin/mpif77." If the distribution lacks such a wrapper, just set to "f77."

LOCALIB — Points to where installed libraries can be found. On systems I control, I install them such that I set this variable to "-L/usr/local/lib", but on supercomputing site I put this in my home directory and set this in my .cshrc as "-L${HOME}/local/lib".

There are some variables in the code one might want to change at compile time.

arena – Located in mem.inc, this describes the size of the memory chunk from fields and such are allocated. Generally one sets it so that the resulting executable size is a bit below the maximum for each processor.

There are other parameters in the .inc files one may want to change, but if one got to the point where they would need to be changed, the user probably already understands what they do.

IV. ASSOCIATED TOOLS

`DV`, `XVS`, `RNPL`, `SDFapps`.