PAMR Reference Manual

Frans Pretorius
Department of Physics
P-412, Avadh Bhatia Physics Laboratory
University of Alberta
Edmonton, AB, T6G 2J1

Contents

I. INTRODUCTION

PAMR (parallel adaptive mesh refinement) is a library of routines designed to manage *distributed grid hierarchies*, in particular, those arising from finite difference solutions of systems of partial differential equations via adaptive mesh refinement. PAMR is still very much under development, as is this document. Comments, questions, bug reports (no guarantees that they will be fixed!), etc. should be sent to

`fransp@phys.ualberta.ca`

The following sections (rather tersely) describe all the current functions and parameters offered by the library routines. At this stage only a C-language interface exits, though a fortran interface is in the works. The type **real** used throughout the interface is currently defined via (see /include/internal_opts.h)

```
#define real double
```

true and **false** refer to C-style booleans, namely zero for **false** and non-zero for **true** (though these are *not* types defined in the PAMR headers).

Arrays are required to be indexed fortran style, i.e. first index varies most rapidly. Also, all array indices passed to and returned from various PAMR functions use fortran style numbering (element 1 is the first element in the array); however example code statements below are in C, and so adjust these numbers to C style.

II. INITIALIZATION AND CLEAN-UP

A *context* refers to all internal data structures associated with a given distributed grid hierarchy. In the future support may be added for the use of multiple contexts, and transferring data between different contexts. However, a single context is sufficiently general to describe the kinds of grid hierarchies needed to solve coupled elliptic/hyperbolic PDEs using interwoven AMR and multigrid hierarchies, using (for instance) the algorithm described in [1]. The following initialization function must be called before any variables or grids can be defined:

```
int PAMR_init_context(real *q, int q_size, int dim, int *shape,
                      real *bbox, char *cp_file, int cp_rank,
                      char *v_tag, char *c_tag);
```

return value: context number (>0), or 0 if an error occurred.

q,q_size: *not yet implemented*

dim: spatial dimension of grids in hierarchy (1, 2 or 3 currently supported)

***shape:** a vector of size **dim** describing the array size of the base, vertex centered grid in the hierarchy. The base cell centered array will have a size one smaller than this along each direction.

***bbox:** a vector of size **dim** describing the coordinate bounding box $[x_1, x_2, y_1, y_2, ...]$ of the base grid

cp_file: if a non-empty string is passed, then this specifies previously saved check-point files (via *PAMR_cp()*) to restore the grid hierarchy from (thus ignoring the **dim**, **shape** and **bbox** parameters). If **cp_rank** below is -1, then a set of files *cp_file.sdf, cp_file_0.sdf, ..., cp_file_n.sdf* is expected that contain the saved grid hierarchy from a previous run on $n+1$ nodes. Note that *every* node in the current run must have access to all these files, however the number of processes in the new run does *not* need to be $n+1$.

v_tag,c_tag: A variable in PAMR can have a cell centered or vertex centered representation. A variable's name determines what kind of variable it is, guided by the **c_tag**/**v_tag** arguments, as follows. If both **c_tag** and **v_tag** are null, then *all* variables are vertex centered. If **c_tag** (**v_tag**) is null but **v_tag** (**c_tag**) is not, then all variables ending in **v_tag** (**c_tag**) are vertex (cell) centered, and all the rest are cell (vertex) centered. If both **c_tag** and (**v_tag**) are not null, then variables ending in **c_tag** (**v_tag**) are cell (vertex) centered, and variables without either appendage are vertex centered.

The reasoning behind this naming convention is as follows. A given evolution scheme for some variable will typically require either a cell centered or vertex centered discretization. If one is solving a coupled system of equations with mixed cell and vertex centered solution methods, then in general *both* representations of a variable will be needed—the primary representation in solving for the variable, the secondary one in equations for different variables that couple to the given variable, but are solved for using the other representation. The default naming conventions when neither **c_tag** or **v_tag** are supplied is for compatibility with earlier versions of PAMR.

For example, with **v_tag="_v"** and **c_tag="_c"**, **f_c** and **f_v** *should be interpreted* as referring to the *same* variable **f**, but **f_c** is the cell centered representation and **f_v** the vertex centered one. The variable **f** is exactly the same variable as **f_v**. Note that user is free to *not* follow this interpretation, however, then the **PAMR_v_to_c()** and **PAMR_c_to_v()** functions will not be of use.

The following function frees all resources associated with context *num*:

```
void PAMR_free_context(int num);
```

III. VARIABLE DEFINITION

The following functions allow one to define and query the set of variables (grid functions) that will comprise the hierarchy.

```
int PAMR_def_var_full(char *name, int in_amrh, int in_mgh, int num_tl, int amr_inject,
                      int amr_interp, int amr_bdy_interp, int amr_sync, int mg_inject,
                      int mg_interp, int mg_sync, int mg_noinj_to_amr, int regrid_transfer,
                      int c_to_v, int v_to_c, int *phys_bdy_type);
```

parameter	description	intended representation (Vertex/Cell)
PAMR_NO_INJECT	no injection	—
PAMR_STRAIGHT_INJECT	straight injection	VC
PAMR_HW_RESTR	half-weight restriction	VC
PAMR_FW_RESTR	full-weight restriction	VC
PAMR_NN_AVERAGE	nearest-neighbour averaging over parent cells	CC
PAMR_NN_ADD	nearest-neighbour addition over parent cells	CC

TABLE I: Current fine to coarse level injection operators (defined in **/include/pamr.h**)

return value: the starting grid function number **sgfn** of the variable. A grid function number (GFN) is an index, starting at 1, into an array of pointers to actual grid function data, as returned by *PAMR_get_g_gfs()*, for instance. Time levels $[1, 2, ..., \mathbf{num\_tl}]$ (1 being the most advanced) of grid function data in the AMR hierarchy are mapped to GFN's $[\mathbf{sgfn}, \mathbf{sgfn}+1, ..., \mathbf{sgfn}+\mathbf{num\_tl}-1]$ respectively. If **in_mgh** is **true**, the grid function data in the multigrid (MG) hierarchy is mapped to GFN $\mathbf{sgfn}+\mathbf{num\_tl}$.

name: the variable's unique name

in_amrh: whether this variable exists in the AMR hierarchy (**true** or **false**)

in_mgh: whether this variable exists in the MG hierarchy (**true** or **false**)

num_tl: if (**in_amrh**), the number of time levels associated with this variable

amr_inject: if (**in_amrh**), the kind of fine to coarse level injection to be performed in the AMR hierarchy. See Table I for the current options. Note that PAMR_NN_AVERAGE is for intensive quantities (such as the density in a given cell), while PAMR_NN_ADD is for extensive quantities (such as mass in a given cell).

amr_interp: if (**in_amrh**), the kind of coarse to fine level interpolation to be performed in the AMR hierarchy. See Table II for the current options.

amr_bdy_interp: if (**in_amrh**), the kind of coarse to fine level interpolation to be performed on AMR boundaries. See Table II for the current options.

amr_sync: if (**in_amrh**), whether this grid function's ghost regions should be synchronized in the AMR hierarchy. See Table III for the current options.

mg_inject: if (**in_mgh**), the kind of fine to coarse level injection to be performed in the MG hierarchy. See Table I for the current options.

mg_interp: if (**in_mgh**), the kind of coarse to fine level interpolation to be performed in the MG hierarchy. See Table II for the current options.

mg_sync: if (**in_mgh**), whether this grid function's ghost regions should be synchronized in the MG hierarchy. See Table III for the current options.

mg_noinj_to_amr: if (**in_mgh** and **in_amrg**), this option (when **true**) disables injection from a coarse level to a fine level in the MG hierarchy *if* the fine level also exists in the AMR hierarchy (as levels that exist in both hierarchies share grid function memory).

regrid_transfer: if (**in_amrh**), the kind of transfer operation to perform during a regrid of the AMR hierarchy. See Table IV for the current options.

c_to_v: the kind of operation to perform when transferring from a cell to vertex centered representation. See Table V for the current options.

v_to_c: the kind of operation to perform when transferring from a vertex to cell centered representation. See Table V for the current options.

phys_bdy_type: a pointer to an array of $(2 \times dim)$ elements. *option not yet implemented*

parameter	description	intended representation (Vertex/Cell)
PAMR_NO_INTERP	no interpolation	—
PAMR_SECOND_ORDER	second order polynomial	VC
PAMR_FOURTH_ORDER	fourth order polynomial	VC
PAMR_SECOND_ORDER_ENO*	second order polynomial with ENO slope limiter	CC
PAMR_FOURTH_ORDER_ENO*	fourth order polynomial with ENO slope limiter	CC
PAMR_FIRST_ORDER_CONS	first order conservative	CC
PAMR_MC	second order MC limiter	CC
PAMR_FIRST_ORDER_EXTENSIVE	first order extensive	CC

TABLE II: Coarse to fine level interpolation operators (defined in **/include/pamr.h**) Asterisks denote methods which are not yet implemented.

parameter	description
PAMR_NO_SYNC	no synchronization
PAMR_SYNC	enforce single-valuedness in overlap regions

TABLE III: Current inter-level synchronization options (defined in **/include/pamr.h**)

The following is a short-cut to *PAMR_def_var_full()*, and defines a new variable **name** with *identical* parameters to those of the most recent variable defined. The new variable's starting GFN is returned.

```
int PAMR_def_var_brief(char *name);
```

After definition, variable attributes can be changed or queried via the following two functions. The arguments have identical meaning to those described in *PAMR_def_var_full()*, and the boolean return code indicates if the operation was successful.

```
int PAMR_set_var_attribs(char *name, int in_amrh, int in_mgh, int num_tl, int amr_inject,
                         int amr_interp, int amr_bdy_interp, int amr_sync, int mg_inject,
                         int mg_interp, int mg_sync, int mg_noinj_to_amr, int regrid_transfer,
                         int c_to_v, int v_to_c, int *phys_bdy_type);

int PAMR_get_var_attribs(char *name, int *in_amrh, int *in_mgh, int *num_tl, int *amr_inject,
                         int *amr_interp, int *amr_bdy_interp, int *amr_sync, int *mg_inject,
                         int *mg_interp, int *mg_sync, int *mg_noinj_to_amr, int *regrid_transfer,
                         int *c_to_v, int *v_to_c, int *phys_bdy_type);
```

parameter	description	intended representation (Vertex/Cell)
PAMR_NO_INTERP	no interpolation	—
PAMR_SECOND_ORDER	second order polynomial	VC
PAMR_FOURTH_ORDER	fourth order polynomial	VC
PAMR_SECOND_ORDER_ENO*	second order polynomial with ENO slope limiter	CC
PAMR_FOURTH_ORDER_ENO*	fourth order polynomial with ENO slope limiter	CC
PAMR_FIRST_ORDER_CONS	first order conservative	CC
PAMR_MC	MC slope limiter	CC
PAMR_FIRST_ORDER_EXTENSIVE	first order extensive	CC

TABLE IV: Regrid transfer operators (defined in **/include/pamr.h**). If transferring data between hierarchies, data is first interpolated from coarse to fine levels via the operator specified above, then data is copied from existing data at the same level. Asterisks denote methods which are not yet implemented.

parameter	description
PAMR_V_TO_C_NO_TRANSFER	no transfer
PAMR_V_TO_C_SECOND_ORDER	second order polynomial
PAMR_V_TO_C_FOURTH_ORDER*	fourth order polynomial
PAMR_C_TO_V_NO_TRANSFER	no transfer
PAMR_C_TO_V_SECOND_ORDER	second order polynomial
PAMR_C_TO_V_FOURTH_ORDER*	fourth order polynomial
PAMR_C_TO_V_SECOND_ORDER_ENO*	second order polynomial with ENO slope limiter
PAMR_C_TO_V_FOURTH_ORDER_ENO*	fourth order polynomial with ENO slope limiter

TABLE V: Transfer operators (defined in **/include/pamr.h**) when copying a function from a cell(vertex) centered representation to a vertex(cell) centered one. Asterisks denote methods which are not yet implemented.

The following set of commands can be used to *disabled*, *enable*, or query the corresponding state of a variable, respectively. A variable that is enabled (disabled) will (will not) have memory allocated for it during *subsequent global* regridding operations. Note therefore that the state of a variable will not change immediately upon the execution of one of these commands, only when the next *PAMR_compose_hierarchy()* that involves *the entire hierarchy* is performed.

```
void PAMR_disable_var(char *name);
void PAMR_enable_var(char *name);
int PAMR_is_var_enabled(char *name);
```

The following is a utility function to map a particular grid function to its grid function number (GFN).

```
int PAMR_get_gfn(char *name, int hier, int tl);
```

return: the GFN of the grid function *name*, in hierarchy *hier*, at time level *tl*. If the requested grid function doesn't exist, 0 is returned.

***name:** the name of the grid function.

hier: which hierarchy (see Table VII).

tl: the time level if in the AMR hierarchy, else 0.

The following function returns the type (cell or vertex centered) of a variable, based on the naming convention defined in **PAMR_init_context()** (note that the variable does not need to be defined).

```
int PAMR_var_type(char *name);
```

return: either **PAMR_VERTEX_CENTERED** or **PAMR_CELL_CENTERED**.

***name:** a variable name.

The following function returns the type (cell or vertex centered) of a data array corresponding to a specific grid function number (**gfn**).

```
int PAMR_gfn_var_type(int gfn);
```

return: PAMR_VERTEX_CENTERED, **PAMR_CELL_CENTERED** or -1 if **gfn** is invalid.

gfn: a grid function number (**gfn**).

IV. GLOBAL PARAMETERS

The set of functions listed here allow one to set and/or query miscellaneous grid hierarchy parameters. The argument descriptions below only describe the set-parameters—the query (get) function arguments are identical except for the usage of pass-by-reference rather than pass-by-value in some instances. Unless otherwise noted, all arguments that are pointers to objects require the *user* to pre-allocate the memory of the object.

```
void PAMR_set_lambda(real lambda);
void PAMR_get_lambda(real *lambda);
```

lambda: the CFL factor

```
void PAMR_set_rho(int *rho_sp, int *rho_tm, int num);
void PAMR_get_rho(int *rho_sp, int *rho_tm, int num);
```

***rho_sp:** an array of size **num** (up to a maximum of PAMR_MAX_LEVS) defining the spatial refinement ratio of levels 1..**num** in the hierarchy (with level 1 being the coarsest level).

***rho_tm:** same as **rho_sp** but for the temporal refinement ratio

```
void PAMR_get_dxdt(int lev, real *dx, real *dt);
```

lev: a level number of the current AMR hierarchy (1..PAMR_MAX_LEVS)

real *dx: an array of size **dim** reals (**dim** is the number of spatial dimensions) that will be filled, upon return, with the mesh spacing of level **lev**. **dx[i]** cannot be set directly, but is calculated from the current base grid information and spatial refinement ratios.

real *dt: a real that will be set, upon return, with the time discretization scale of level **lev**. The base level **dt** is calculated as the minimum of over **i** of **lambda*dx[i]**, where **i** runs from 1 to **dim**; higher level **dt**'s are calculated using the base level **dt** and temporal refinement ratio.

```
void PAMR_set_ghost_width(int *ghost_width);
void PAMR_get_ghost_width(int *ghost_width);
```

***ghost_width:** an array of size **dim**, where **ghost_width[i-1]** defines the number of ghost *cells* that are needed at artificial grid boundaries introduced in the **i** direction via the grid distribution process. In other words, **ghost_width[i-1]** is interpreted as the width of the ghost region in units of the grid spacing.

```
void PAMR_set_periodic_bdy(int *periodic);
void PAMR_get_periodic_bdy(int *periodic);
```

***periodic:** an array of size **dim** defining whether boundary **i** is periodic (**periodic[i-1]=true**) or not (**periodic[i-1]=false**, which is the default). If a boundary is periodic, grids adjacent to it along the right edge (larger coordinate value) are extended by an amount proportional to the corresponding ghost width. All PAMR communication functions then use this extra domain to enforce the periodicity of the domain across the boundary.

```
void PAMR_set_min_width(int *min_width);
void PAMR_get_min_width(int *min_width);
```

***min_width:** an array of size **dim** defining the minimum number of points allowed for any distributed grid component. In other words, no grid will be split into components with a size along dimension **i** smaller than that given by **min_width[i-1]**.

```
void PAMR_set_MG_coarse_width(int *min_width);
void PAMR_get_MG_coarse_width(int *min_width);
```

***min_width:** an array of size **dim** defining the minimum number of points allowed for any grid in the multigrid hierarchy (independent of how that grid might be distributed across a network). In other words, these parameters defined the coarsest possible grid in the multigrid hierarchy.

parameter	description
PAMR_GDM_GRID_BY_GRID	individually splits *each* grid in a level among the n nodes in the parallel environment
	[default is to collectively divide the memory of all grids in a level across the nodes]
PAMR_GDM_ALIGN	forces *all* child grid boundaries to be aligned with the mesh of a parent
	grid, and does so by increasing the local ghost width, if needed
PAMR_GDM_NO_OVERLAP	clips all overlapping grids in the sequential hierarchy prior to distribution
	(i.e., the only overlap after *PAMR_compose_hierarchy()* will be the ghostzones)

TABLE VI: Current grid distribution options (defined in **/include/pamr.h**). Each one of these options behaves as an on/off switch (default is off), and the **method** parameter passed to *PAMR_set_gdm()* is the bitwise-or (|) of the desired set of flags.

parameter	description
PAMR_AMRH	the AMR hierarchy
PAMR_MGH	the MG hierarchy

TABLE VII: Current grid hierarchies that can exist within a context (defined in **/include/pamr.h**).

```
void PAMR_set_interp_buffer(int interp_buffer);
void PAMR_get_interp_buffer(int *interp_buffer);
```

***interp_buffer:** an array of size **dim** defining an additional "communication buffer" to place around a coarse grid during a coarse-to-fine interpolation step. This does not affect the size of the region updated on the fine grid, rather it allows the use of interior interpolation stencils for a broader set of points on the fine grid (for example, set to 1 for second order interpolation, and 2 for fourth order).

```
void PAMR_get_global_bbox(real *bbox);
```

***bbox:** a vector describing the coordinate bounding box $[x_1, x_2, y_1, y_2, ...]$ of the base grid, as originally specified in *PAMR_init_context()*

```
void PAMR_set_gdm(int method);
void PAMR_get_gdm(int *method);
```

method: specifies how grids are distributed in a parallel environment; see Table VI for the current options.

```
void PAMR_set_trace_lev(int lev);
```

lev: the level of verbosity of PAMR output for *PAMR debugging* purposes (i.e. these traces will probably not be useful in debugging any numerical code). 0 is no output, 1 is a function trace, and options greater than 1 can produce disastrous amounts of output.

```
int PAMR_get_max_lev(int hier);
```

return: the current finest level number (1 is coarsest) containing grids in the hierarchy specified by **hier**

hier: which hierarchy (see Table VII).

```
int PAMR_get_min_lev(int hier);
```

return: the current coarsest level number (will usually be 1) containing grids in the hierarchy specified by **hier**

hier: which hierarchy (see Table VII).

V. HIERARCHY CONSTRUCTION

Once the context has been initialized and all the variables have been defined, *PAMR_compose_hierarchy()* will distribute and allocate the local memory for a given AMR grid hierarchy. Regridding is performed via subsequent calls to *PAMR_compose_hierarchy()*, which will copy/interpolate (by at most one level) data from like time levels of the old to new hierarchy as specified in the variable definitions. A multigrid (MG) hierarchy can be constructed on top of an existing AMR hierarchy via a call to *PAMR_build_mgh()*, and subsequently removed with *PAMR_destroy_mgh()*. See [1] for a discussion of a MG hierarchy and how to use it. Regridding cannot be performed when a MG hierarchy exists.

NOTE: PAMR does *not* interpolate in time when initializing new fine grids, or portions thereof, in a multi time-level hierarchy. This means that if certain time levels are not in sync (which is the case for past time levels in Berger and Oliger style AMR), it is the user's responsibility to do the appropriate temporal interpolation, in *new* portions of the fine grid, afterward. In existing regions of the fine grid data is copied from the old level, and this data will (presumably) be at the correct time. In the future support may be added for certain standard temporal interpolation schemes; however, temporal interpolation is a purely local operation (network-wise) and so is not an essential feature for a grid distribution package like PAMR to provide.

```
void PAMR_compose_hierarchy(int min_lev, int max_lev, int num, int *lev, real *bbox, real t);
```

> **min_lev,max_lev:** grids on levels from **min_lev** to **max_lev** are to be allocated (or reallocated) as specified via the [**lev**,**bbox**] arrays. During a regrid operation, if no grids are specified on a given level, then that level is removed from the hierarchy.
>
> **num:** the number of new grids
>
> ***lev:** an array of size **num**, where **lev[i-1]**$\in 1, 2, ...$ is the level (1 is coarsest) of new grid **i** ($\in 1, 2, ...$**num**).
>
> ***bbox:** an array of size **2*dim*num**, where [**bbox[2*dim*(i-1)],...,bbox[2*dim*(i)-1]**] stores the coordinate bounding box $[x_1, x_2, y_1, y_2, ...]$ of new grid **i**.
>
> **t:** the coordinate time of the new levels.

```
void PAMR_build_mgh(int min_lev, int max_lev, int tl);
```

> **min_lev:** the minimum (coarsest) AMR level to partake in the MG hierarchy.
>
> **max_lev:** the maximum (finest) AMR level to partake in the MG hierarchy. Note that in general the MG hierarchy will contain coarser levels than the AMR hierarchy. The coarsest level in the MG hierarchy is always labeled level 1; hence some level $\ell > 1$ in the MG hierarchy will correspond to level **min_lev** of the AMR hierarchy.
>
> **tl:** the time level of the AMR hierarchy that will be used to populate the MG hierarchy.

```
void PAMR_destroy_mgh();
```

VI. COMMUNICATION

PAMR provides several basic communication functions: *synchronize* grid functions that overlap on a given level (**PAMR_sync**), *interpolate* grid functions from a coarse to the next finer level (**PAMR_interp**,**PAMR_AMR_bdy_interp**), *inject* grid functions from a fine to the next coarser level (**PAMR_inject**), and copy values from a vertex to cell centered representation (**PAMR_v_to_c**) and vice-versa (**PAMR_c_to_v**). Note that **PAMR_AMR_bdy_interp_c** was introduced to permit a different interpretation of the width argument in the cell-centered and vertex-centered cases (see below). By default, the particular grid functions involved and the kind of injection/interpolation operations performed are those specified in the variable definitions; however these defaults can be overwritten using *transfer bits*, controlled via the **..._tf_bits** functions described below. Each grid function at each time level in both the AMR and MG hierarchies (i.e. each GFN) has a unique transfer bit associated with it, which specifies how the corresponding grids are treated during a communication operation (the particular values depend upon the operation, and are listed in Tables I, II, III and IV). By default, PAMR's communication functions call **PAMR_clear_tf_bits**, which turns off communication for all grid functions, then **PAMR_set_tf_bits** before doing any communication. To set custom transfer bits using any combination of

PAMR_clear_tf_bits, **PAMR_set_tf_bits** and **PAMR_set_tf_bit**, the user must call **PAMR_freeze_tf_bits** *prior* to invoking any communication functions; this stops the communication functions from over-writing the custom transfer bits. Afterward, **PAMR_thaw_tf_bits** should be called to return to the default communication scheme.

```
int PAMR_sync(int l, int tl, int hierarchy, int AMR_bdy_width);
```

return: **true** for success, **false** for failure.

l: level of the hierarchy to sync.

tl: if the AMR hierarchy, the time level to sync.

hierarchy: which hierarchy—see Tab. VII.

AMR_bdy_width: if greater than 0, specifies a region about AMR boundaries to exclude from consideration. (The value of **AMR_bdy_width** is the width of this region in units of the grid spacing, *i.e.* the cell width.)

```
int PAMR_inject(int lf, int tl, int hierarchy);
```

return: **true** for success, **false** for failure

lf: the fine level of the hierarchy to inject to coarser level **lf-1**.

tl: if the AMR hierarchy, the time level to inject.

hierarchy: which hierarchy—see Tab. VII.

```
int PAMR_interp(int lc, int tl, int hierarchy);
```

return: **true** for success, **false** for failure

lc: the coarse level of the hierarchy to interpolate to finer level **lc+1**.

tl: if the AMR hierarchy, the time level to interpolate.

hierarchy: which hierarchy—see Tab. VII.

```
int PAMR_AMR_bdy_interp(int lc, int tl, int AMR_bdy_width);
```

return: **true** for success, **false** for failure

lc: the coarse level of the AMR hierarchy to interpolate to finer level **lc+1**.

tl: the time level to interpolate.

AMR_bdy_width: this function interpolates to a region of width **AMR_bdy_width** points along AMR boundaries on the fine level.

```
int PAMR_AMR_bdy_interp_c(int lc, int tl, int AMR_bdy_width_c);
```

return: **true** for success, **false** for failure

lc: the coarse level of the AMR hierarchy to interpolate to finer level **lc+1**.

tl: the time level to interpolate.

AMR_bdy_width_c: this function interpolates to a region of size **AMR_bdy_width_c** (in units of the grid spacing) along AMR boundaries on the fine level.

```
int PAMR_c_to_v(int l, int tl, int hierarchy);
```

return: **true** for success, **false** for failure

l: level of the hierarchy to copy.

tl: the time level to copy.

hierarchy: which hierarchy—see Tab. VII.

`int PAMR_c_to_v_local(int tl, int hierarchy);`

return: true for success, **false** for failure

tl: the time level to copy.

hierarchy: which hierarchy—see Tab. VII.

`int PAMR_v_to_c(int l, int tl, int hierarchy);`

return: true for success, **false** for failure

l: level of the hierarchy to copy.

tl: the time level to copy.

hierarchy: which hierarchy—see Tab. VII.

`int PAMR_v_to_c_local(int tl, int hierarchy);`

return: true for success, **false** for failure

tl: the time level to copy.

hierarchy: which hierarchy—see Tab. VII.

`void PAMR_freeze_tf_bits(void);`

`void PAMR_thaw_tf_bits(void);`

`void PAMR_clear_tf_bits(void);`

`void PAMR_set_tf_bit(int gf, int val);`

gf: the grid function number (see **PAMR_get_gfn**, **PAMR_def_var_full**) whose transfer bit will be set to **val**.

val: one of the quantities in Tables I, II, III and IV, depending upon which communication function will subsequently be used.

`void PAMR_set_tf_bits(int tl, int hierarchy, int stage);`

tl: if the AMR hierarchy, the time level whose grids are involved.

hierarchy: which hierarchy—see Tab. VII.

stage: an operation number defining what stage of communication to set the transfer bits for — see Tab. VIII.

VII. GRID-FUNCTION ACCESS

Once a grid hierarchy has been allocated, the functions described here can be used to access the individual grids. A *sequential iterator* is used to loop through all grids at a given level within a hierarchy; the basic usage is demonstrated in the following C example code, which loops through all local grids at level 2 in the AMR hierarchy:

```
int valid;

valid=PAMR_init_s_iter(2,PAMR_AMRH,0);
while(valid)
{
   // do stuff with current grid

   valid=PAMR_next_g();
}
```

parameter	description
PAMR_TF_SYNC	synchronize
PAMR_TF_COMPOSE	regrid
PAMR_TF_INJECT	inject
PAMR_TF_INJECT_TO_MG_LEVEL	inject to a MG level that does not exist in the AMR hierarchy
PAMR_TF_INTERP	interpolate
PAMR_TF_BDY_INTERP	boundary interpolate
PAMR_TF_BDY_INTERP_C	boundary interpolate for CC variables
PAMR_TF_MGH_INIT	inject only functions common to both MG and AMR MG hierarchies
PAMR_TF_C_TO_V	copy from cell centered to vertex center representations
PAMR_TF_V_TO_C	copy from vertex centered to cell center representations

TABLE VIII: Options for the **stage** variable of **PAMR_set_tf_bits**.

When an iterator is in a "valid" state, all attributes and data associated with the corresponding grid can be accessed via the **PAMR_get_g_...** functions below. As with other **PAMR_get_...** functions, unless otherwise noted any information returned through argument pointers requires the user to preallocate the required memory for the object; i.e. PAMR simply copies the requested data to the location pointed to by the argument. Nested iteration can be performed using the internal iterator stack, accessed via **PAMR_pop_iter** and **PAMR_push_iter**.

```
int PAMR_init_s_iter(int l, int hier, int all);
```

return: **true** for success (implying that the first grid in the list of grids at this level can subsequently be accessed), **false** for failure (implying either that there are no grids at the requested level, or some invalid parameters where passed).

l: the level (1 is coarsest) of the hierarchy whose grids will be accessed

hier: which hierarchy—see Tab. VII.

all: if **all=false** then only local grids (i.e. those with function data and coordinate arrays that are allocated on the local machine) are iterated through, otherwise all grids are accessed. Note that one *cannot* access a *non-local* grid's data (via **PAMR_get_g_gfs**) or coordinate arrays (via **PAMR_get_g_x**).

```
int PAMR_next_g();
```

return: **true** for success (meaning the iterator is referring to a valid grid), **false** for failure or the iterator has reached the end of the list.

```
int PAMR_push_iter();
```

return: **true** for success, **false** for failure (internal stack overflow, or invalid iterator).

```
int PAMR_pop_iter();
```

return: **true** for success (with the current grid restored to that pointed to at the time of the corresponding push), **false** for failure (stack empty).

```
int PAMR_get_g_attribs(int *rank, int *dim, int *shape, int *shape_c, real *bbox,
                       int *ghost_width, real *t, int *ngfs, real **x, real **x_c, real **gfs);
int PAMR_get_g_rank(int *rank);
int PAMR_get_g_dim(int *dim);
int PAMR_get_g_shape(int *shape);
int PAMR_get_g_shape_c(int *shape_c);
int PAMR_get_g_bbox(real *bbox);
int PAMR_get_g_ghost_width(int *ghost_width);
int PAMR_get_g_t(real *t);
int PAMR_get_g_ngfs(int *ngfs);
int PAMR_get_g_x(real **x);
int PAMR_get_g_x_c(real **x_c);
int PAMR_get_g_gfs(real **gfs);
```

return: **true** if the current iterator is valid and hence the returned fields are valid, **false** otherwise.

***rank:** the MPI rank of the node where this grid's data is stored

***dim:** the spatial dimension of the grid

***shape:** an array of size **dim** that specifies the number of points along each dimension of the grid $[N_x, N_y, ...]$.

***shape_c:** an array of size **dim** that specifies the number of cells along each dimension of the grid (will be $[N_x - 1, N_y - 1, ...]$, if $[N_x, N_y, ...]$ is the shape of the vertex centered array).

***bbox:** an array of size **2*dim** that describes the coordinate bounding box of the grid $[x_1, x_2, y_1, y_2, ...]$.

***ghost_width:** an array of size **2*dim** indicating the size of the ghost region, in *cells*, adjacent to the $[x_{min}, x_{max}, y_{min}, y_{max}, ...]$ boundaries of the grid.

***t:** the coordinate time t.

***ngfs:** the number of grid functions.

****x:** an array of size **dim** that will be filled with pointers to local arrays containing the coordinates of the vertices (of size $N_x, N_y, ...$) *if the grid is local*, else the pointers will be set to 0.

****x_c:** an array of size **dim** that will be filled with pointers to local arrays containing the coordinates of the cell centers (of size $N_x - 1, N_y - 1, ...$) *if the grid is local*, else the pointers will be set to 0.

****gfs:** an array of size **ngfs** that will be filled with pointers to local data arrays (of size $N_x * N_y * ...$) if the corresponding grid function is allocated, else the corresponding pointer is set to 0 (if the grid is non-local, or the hierarchy to which the grid function belongs does not exist).

```
int PAMR_get_g_level(int *L);
```

return: **true** if the iterator is valid, **false** otherwise.

***L:** the level number of the current grid.

```
int PAMR_get_g_comm(int *comm);
int PAMR_set_g_comm(int comm);
```

return: **true** if the iterator is valid, **false** otherwise.

***comm:** whether this grid partakes in any communication functions (**true** or **false**).

VIII. TIME FUNCTIONS

The following functions manage the time levels within the AMR hierarchy. **PAMR_tick** advances the time of the level by its local dt, **PAMR_get_time** and **PAMR_set_time** can be used to set and read the time of a level respectively, and **PAMR_swap_tl** swaps time levels in the hierarchy.

```
real PAMR_tick(int lev);
```

return: the time after the "tick"

lev: the level to update

```
int PAMR_swap_tl(int lev, int tl1, int tl2);
```

return: **true** for success, **false** for failure

lev: the level of the AMR hierarchy

tl1: swap grids at time level **tl1** with those at time level **tl2**

tl2:

```
real PAMR_get_time(int lev);
```

return: the time associated with AMR level **lev**

lev:

```
void PAMR_set_time(int lev, real t);
```

lev: AMR level

t: time to set above level to

IX. EXCISION SUPPORT

PAMR supports excision in the sense that it will alter the interpolation and injection operators near an excised region if desired. The user must supply a routine that, when called, initializes a specified grid function to define the excised region. By default excision is off, is turned on via **PAMR_excision_on**, and can subsequently be turned off via **PAMR_excision_off**

```
int PAMR_excision_on(char *ex_mask_var, char *ex_mask_c_var,
                     void (*app_fill_ex_mask_fnc)(real *mask, real *mask_c, int dim,
                           int *shape, int *shape_c, real *bbox, real excised),
                     real excised, int initialize_now);
```

return: **true** on success, **false** otherwise.

***ex_mask_var:** a vertex centered function, existing in both the AMR and MG hierarchies with only a single time level in the AMR hierarchy, that will be used to define the excised region for vertex centered variables.

***ex_mask_var_c:** a cell centered function, existing in both the AMR and MG hierarchies with only a single time level in the AMR hierarchy, that will be used to define the excised region for cell centered variables.

***app_fill_ex_mask_fnc:** a pointer to a function that, when called, will initialize the **dim** dimensional vertex array **mask** (of size **shape**) and cell array **mask_c** (of size **shape_c**), both with bounding box **bbox**, to **excised** where corresponding components of the grid are excised, and some other number elsewhere. **NOTE:** the function **app_fill_ex_mask_fnc** will be called during regridding when the internal hierarchy is only partially built, hence *none* of the PAMR structure or grid function access routines should be used by this routine.

excised: The value defining an excised point on the grid

initialize_now: Initialize the mask grid on *all* levels now if **initialize_now** is **true**, else the mask will only be initialized during the next regrid.

```
void PAMR_excision_off();
```

X. I/O ROUTINES

PAMR_save_gfn can be used to save grid functions to the local disk in SDF [2] format. **PAMR_cp** is for check-pointing, and saves the current state of the hierarchy to disk.

```
int PAMR_save_gfn(char *name, int hier, int tl, int L, real t, char *pre_tag, char *post_tag);
```

return: **true** if successful, implying that the grid function **name** in hierarchy **hier** (see Table VII), level **L** (-1 for all) and time level **tl** was appended to the SDF file

$<$ **pre_tag** $><$ **name** $>$ [**_tl**|$\_MG$] $<$ **post_tag** $>$ _ $<$ *rank* $>$ *.sdf*,

with a time parameter **t** (-1 to use the current time of the grid function). The time level **tl** is inserted for AMR hierarchy functions, and the letters "MG" for functions in the MG hierarchy. The **rank** is the MPI rank.

***name:** grid function name

hier: which hierarchy (see Table VII)

tl: the time level if the AMR hierarchy, else 0

L: the level

t: the output time to use (set to -1 to use the current grid function time)

***pre_tag:** a prefix tag for the filename

***post_tag:** a postfix tag for the filename

```
int PAMR_cp(char *cp_name, int cp_rank);
```

return: **true** if successful, **false** otherwise

***cp_name:** The base name for the set of checkpoint files to be saved; see *PAMR_init_context()*.

cp_rank: currently **cp_rank** must be -1.

XI. MISCELLANEOUS ROUTINES

PAMR_merge_bboxes is a utility routine to help programs produce the global list of bounding boxes that define a grid hierarchy (as required by **PAMR_compose_hierarchy**), from a local set of bounding boxes as would be computed via local truncation error estimates (for instance). Each node passes its local list of **local_num** bounding boxes in **local_bbox**, and a preallocated memory block in **global_bbox** that can hold up to **global_num** bounding boxes. PAMR will then merge together the local lists, joining two bounding boxes if the "efficiency" of the merge, defined as

$$efficiency = \mathrm{volume}(A \mathrm{union} B)/\mathrm{volume}(C) \tag{1}$$

is larger that **min_eff**. In the above, A and B are two local bounding boxes, and C is the smallest rectangular bounding box encompassing A and B. Upon completion, **global_num** is set to the total number of merged grids.

```
int PAMR_merge_bboxes(real *local_bbox, int local_num, real *global_bbox, int *global_num, real min_eff);
```

return: **true** for success, **false** for failure.

***local_bbox:** an array of size **2*dim*local_num** reals

local_num: the number of local bounding boxes

***global_bbox:** an array of size **2*dim*(*global_num)** reals

***global_num:** as input, the maximum number of bounding boxes the array **global_bbox** can hold, and upon completion is set to the actual number of merged bounding boxes.

min_eff: a number between 0 and 1, as described above.

PAMR_get_sgh

[1] F. Pretorius, "Numerical Simulations of Gravitational Collapse", *Univ. of British Columbia Ph.D. Thesis* (2002)
[2] R.L. Marsa and M.W. Choptuik, "The RNPL User's Guide", http://laplace.physics.ubc.ca/Members/marsa/rnpl/users_guide/users_guide.html (1995)