Forestclaw : Programming paradigms

Donna Calhoun (Boise State University)

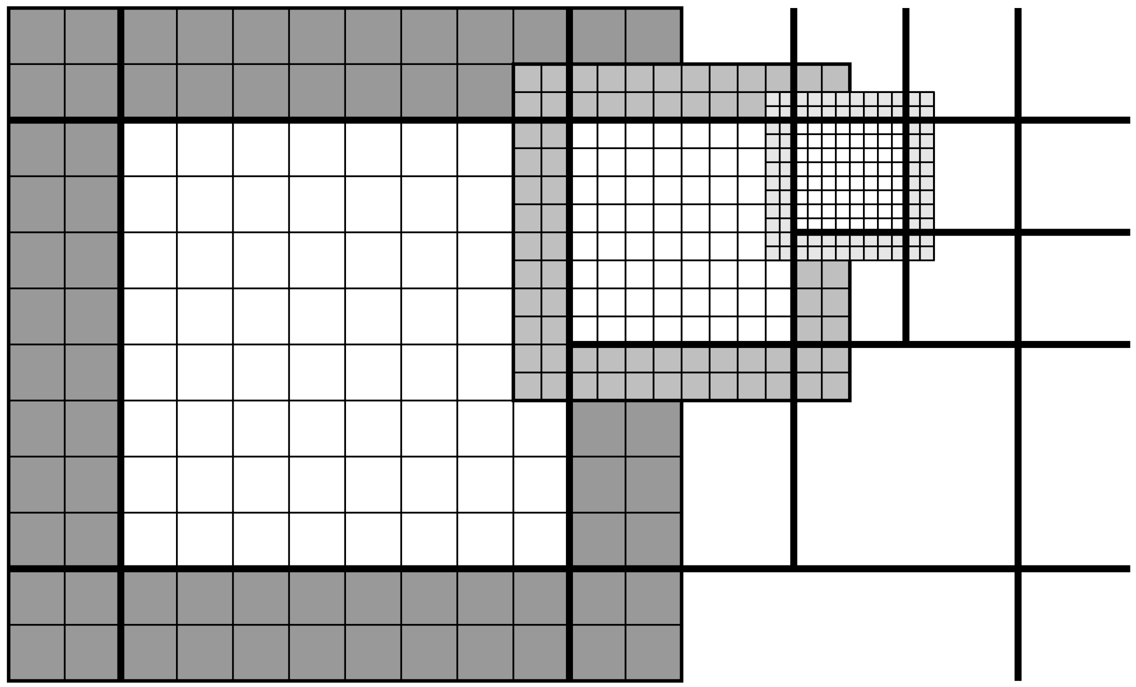
Carsten Burstedde, Univ. of Bonn, Germany

p4est Summer School

July 20 - 25, 2020

Bonn, Germany (Virtual)

ForestClaw : a PDE layer



In the “clawpatch” patch (used for finite volume solvers), each p4est quadrant is occupied by a single logically Cartesian grid, stored in contiguous memory, including ghost cells.

ForestClaw is a **p4est PDE layer**.

- Written mostly in object-oriented C
- Core routines are agnostic as to patch data, solvers used, etc.
- Most aspects of the PDE layer, including type of patch used, solver, interpolation and averaging, ghost-filling, can be customized
- Support for legacy codes
- Several extensions include Clawpack extension, GeoClaw, Ash3d and others.
- FV solvers and meshes are available as applications.

ForestClaw philosophy

- Enable users to port existing Cartesian grid codes to highly scalable, parallel adaptive environment.
- Starting point : Users are experts in their application and solvers, and have put much thought and work into developing their codes
- To the greatest extent possible, users should be able to leverage any existing code they have already developed. Encourage re-use of legacy Cartesian codes.
- If the programming paradigm is clear enough, users can reason about their interaction with the code, and can be involved in technical details of getting their application running.
- Most users are not experts in computer science, nor do they want to be. So language constructs need to be reasonably simple, i.e. limit use of C++. Emphasize procedures over objects. Don't try to invent DSLs that are meaningless to everyone but the developer.
- Encourage mixed programming, i.e. Fortran+C.

Programming paradigms in ForestClaw

Paradigms

- Iterators
- Callbacks
- Virtual tables
- Encapsulated *extension libraries* for defining how patches get updated, and how data within a patch is stored.

Extension libraries

- A solver library can update a solution on a single grid, or, in the case of an elliptic solver, return a solution on the mesh hierarchy. Solver libraries are typically wrappers for legacy code.
- Solvers work together with *patch libraries.*
- Configuration parameters for solvers and patch types (cell-centered, node centered, etc) are contained within the library,
- Composibility : Libraries are design not to clash with each other, so multiple versions of the same library can be compiled together for selection at run-time.

Solver libraries : time stepping

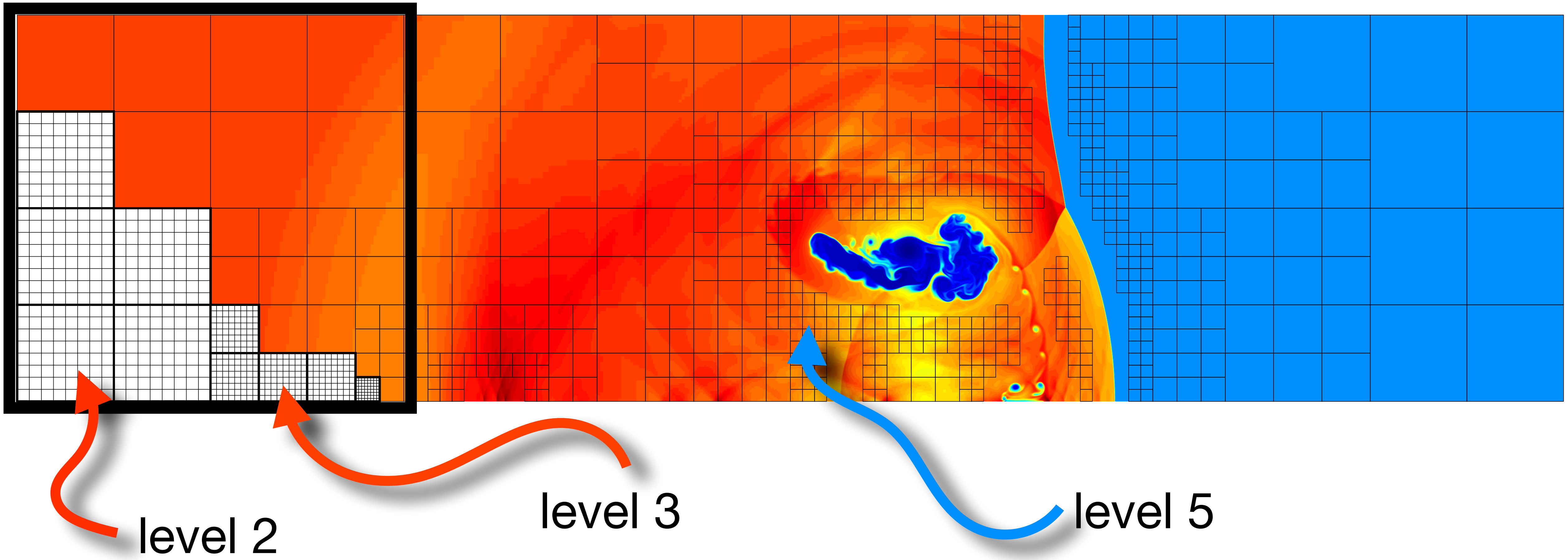
We have an existing Cartesian grid solver

- Let's assume it is an explicit time stepping solver.
- Furthermore, we have a time stepping loop that looks something like this :

```
Choose a time step dt,
for k = 1, M
   Take a single time step
   Output results
   Compute some diagnostics
```

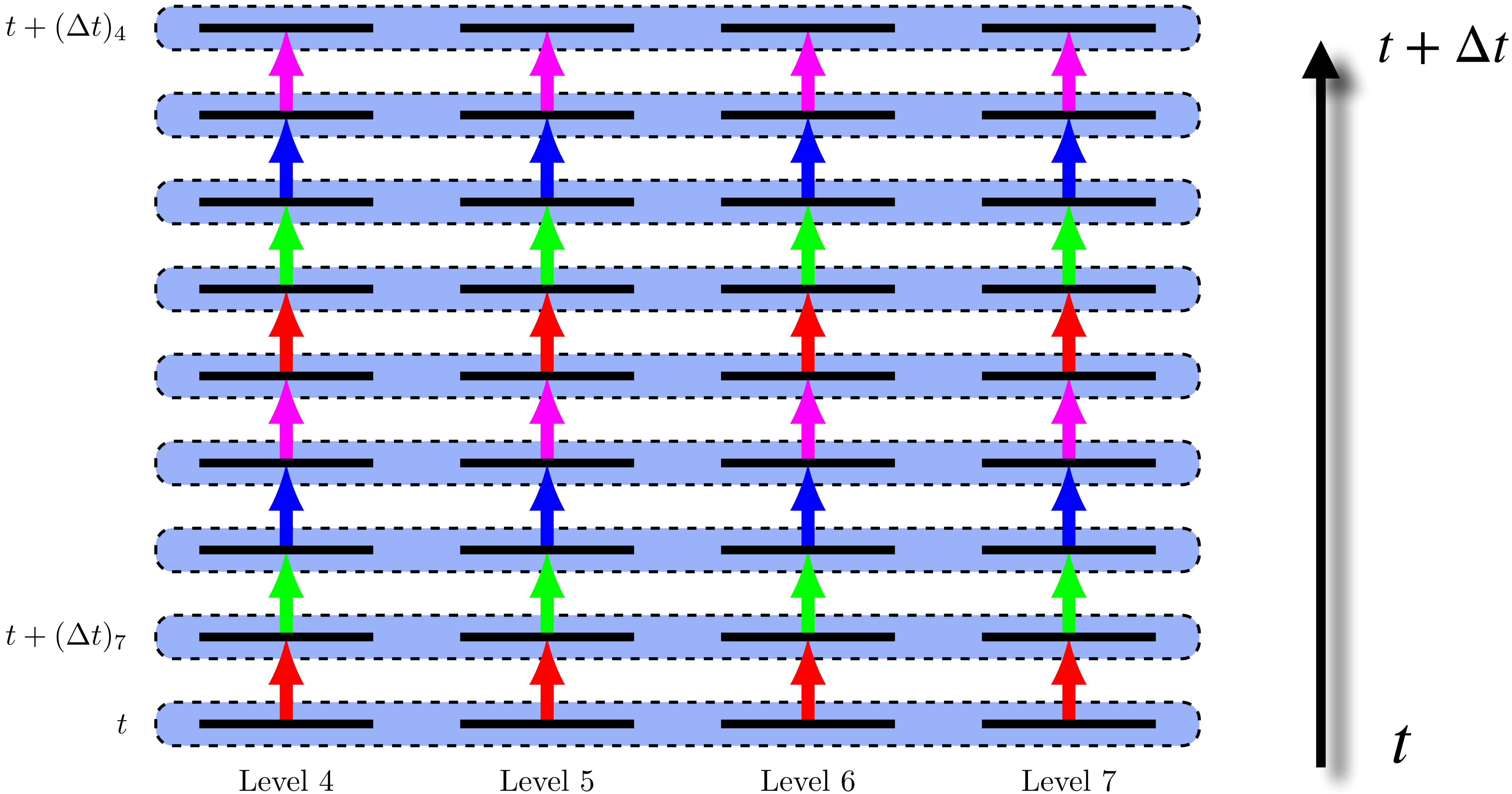
- The time step may depend on a CFL constraint, or some other constraint needed for stability.
- What does this loop look like on an AMR hierarchy?
- Focus on the single time step

Solver libraries : time stepping



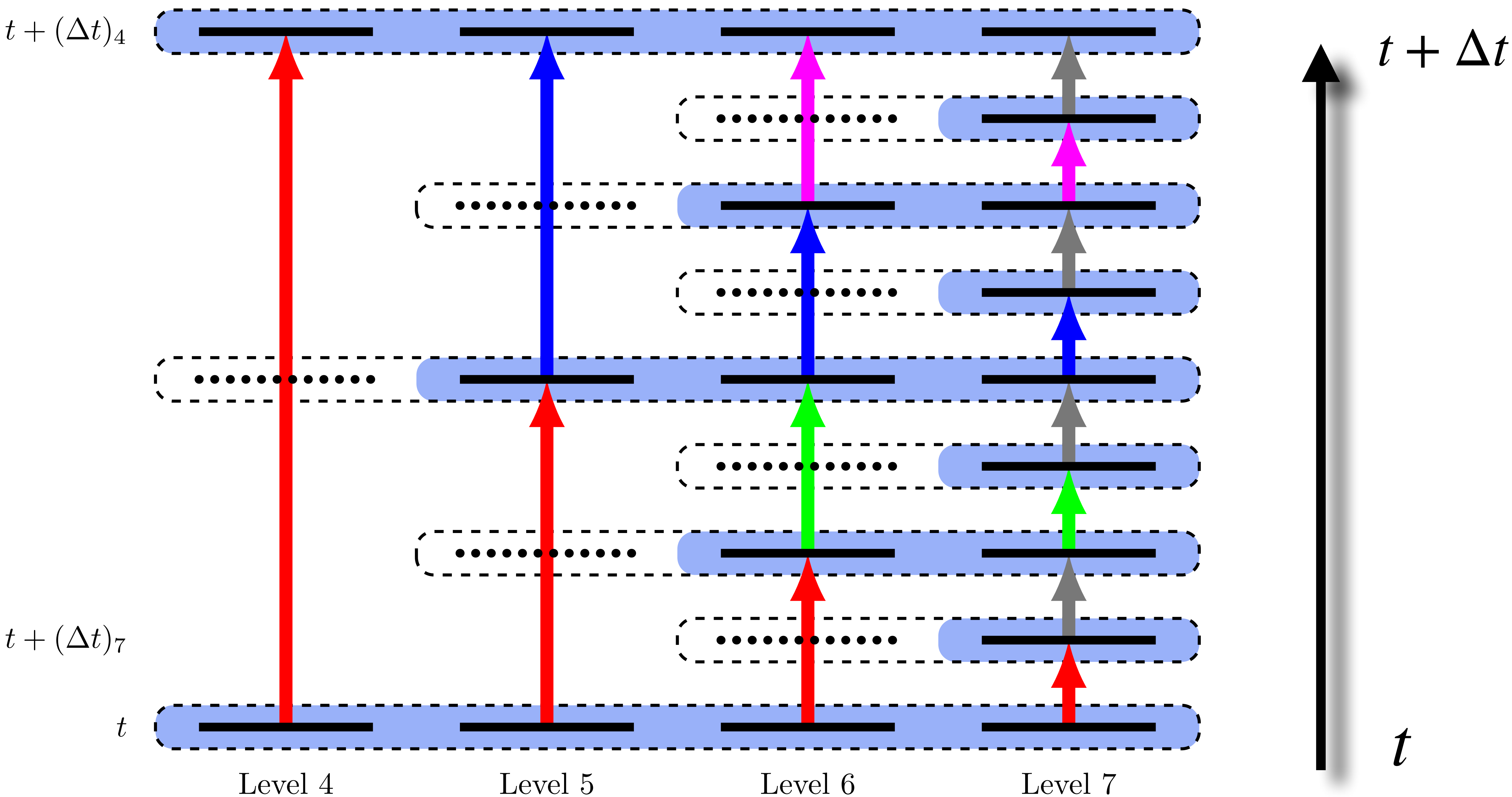
- For hyperbolic problems, the time step is often limited by cell size.
- Global time stepping : One time step Δt for all grids.
- Local time stepping : Time step size depends on cell size
- Benefits of local time stepping depend on the problem

Global time stepping



- Arrows of the same color indicate recursive calls
- Blue boxes indicate parallel ghost cell exchanges

Local time stepping



- Arrows of the same color indicate recursive calls
- Blue boxes indicate parallel ghost cell exchanges

Time stepping algorithm

Require: Grids at all levels at time t must have valid ghost cells values.
for $k = 1$ to $2^{\ell_{max}-\ell_{min}}$ **do**
  ADVANCE_SOLUTION$(\ell_{max}, (\Delta t)_{\ell_{max}})$ — Advance solution on finest level
  if multirate **then** — Multirate
    if $k < 2^{\ell_{max}-\ell_{min}}$ **then**
      Find largest integer $p \geq 0$ such that 2^p divides k.
      $\ell_{time} = \ell_{max} - p - 1$ — Intermediate synchronization
      UPDATE_GHOST$(\ell_{time} + 1)$
    end if
  else — Global time stepping
    UPDATE_GHOST(ℓ_{min})
  end if
end for
UPDATE_GHOST(ℓ_{min}).

procedure ADVANCE_SOLUTION(level $= \ell$, dt_stable $= \Delta t$)
  for all grids g on level ℓ **do**
    Update solution $Q^{n+1} = Q^n + \Delta t\, F(Q^n, t_n)$.
  end for
  if $\ell > \ell_{min}$ **then**
    if multirate **then**
      if levels ℓ and $\ell - 1$ are time synchronized **then**
        ADVANCE_SOLUTION$(\ell - 1, 2\Delta t)$
        TIME_INTERPOLATE$(\ell - 1, t + 2\Delta t)$
      end if
    else
      ADVANCE_SOLUTION$(\ell - 1, \Delta t)$
    end if
  end if
end procedure

Recursive advance, followed by a time interpolation

Single coarse grid time step

```
double fclaw2d_advance_all_levels(fclaw2d_global_t *glob,
                                  double t, double dt) {
    initialize_timestep_counters(glob,&ts_counter,t,dt);

    for(int nf = 0; nf < ts_counter[maxlevel].total_steps; nf++)
       double maxcfl =
            advance_level(glob,maxlevel,nf,maxcfl,ts_counter);
}
double advance_level(fclaw2d_global_t *glob, int level, int nf,
              double maxcfl, fclaw2d_timestep_counters* ts_counter) {
     double cfl = fclaw2d_update_single_step(glob,level,t,dt);
     maxcfl = fmax(maxcfl,cfl);
     if (level > domain->local_minlevel) {
         double dtc = ts_counter[level-1].dt_step;
         double cfl = fclaw2d_update_single_step(glob,level-1,t,dtc);
         maxcfl = fmax(maxcfl,cfl);
     }
}
```

- Time step counter manages global/local time stepping

Iterators and call-back functions

```
double fclaw2d_update_single_step(fclaw2d_global_t *glob, int level,
                                  double t, double dt) {
    /* Store time step, t in struct ss_data */
    fclaw2d_global_iterate_level(glob, level, cb_single_step, &ss_data);
}
```

- A “functional iterator” which loops over all grids on a level.
- Iterator interacts with p4est data structure to extract quads.
- The “callback function” is called for each grid.
- This iterator is used in many contexts, not just time stepping

Iterators and call-back functions

```
void cb_single_step(fclaw2d_domain_t *domain,
                    fclaw2d_patch_t *patch,
                    int blockno, int patch, void *user) {
   /* Extract dt, t, other data from `user` struct */
   double maxcfl =
       fclaw2d_patch_single_step_update(glob, patch, blockno, patchno,
                                        t, dt, &ss_data->buffer_data);
   /* Compare maxcfl to global max;  store in user data */
}
```

- **Call-back** function called for each patch on processor
- User solver is called from *fclaw2d_patch_single_step_update.*
- Assumes patch can be **updated independently** from other patches (wouldn't be appropriate for an elliptic solver, for example)
- The *patch* struct stores **solution data** in virtualized patch types (think: void*).

Virtual tables

```
double fclaw2d_patch_single_step_update(fclaw2d_global_t *glob,
                                        fclaw2d_patch_t *patch,
                                        int blockno, int patchno,
                                        double t, double dt, void* user)
{
    fclaw2d_patch_vtable_t *patch_vt = fclaw2d_patch_vt();
    FCLAW_ASSERT(patch_vt->single_step_update != NULL);
    double maxcfl =
          patch_vt->single_step_update(glob, patch, blockno, patchno,
                                       t, dt, user);
    return maxcfl;
}
```

- Virtual tables are structs that store typedef'ed function pointers.
- Facilitates polymorphism.
- Virtual tables are accessible from anywhere; no need to create objects.

Virtual tables

```
struct fclaw2d_patch_vtable
{
    /* Creating/deleting/building patches */
    fclaw2d_patch_new_t                     patch_new;
    fclaw2d_patch_delete_t                  patch_delete;
   ....
    /* Solver functions */
    fclaw2d_patch_initialize_t              initialize;
    fclaw2d_patch_physical_bc_t             physical_bc;
    fclaw2d_patch_single_step_update_t      single_step_update;
    ....
  }
```

- Structs containing virtual tables are closest thing to an “object” in ForestClaw
- Pointers are set by solvers, patch libraries (more on that later), or the user.
- Function pointer signature is hard-wired.

Virtual tables

```
void fc2d_clawpack46_solver_initialize()
{
    fclaw2d_patch_vtable_t*           patch_vt = fclaw2d_patch_vt();
    fc2d_clawpack46_vtable_t*  claw46_vt = clawpack46_vt_init();
    ...
    /* These could be over-written by user specific settings */
    patch_vt->initialize                         = clawpack46_qinit;
    patch_vt->setup                              = clawpack46_setaux;
    patch_vt->physical_bc                        = clawpack46_bc2;
    patch_vt->single_step_update                 = clawpack46_update;
   ...
   claw46_vt->is_set = 1;
}
```

- These functions operate on a single patch only
- Encapsulated solver libraries assign values to function pointers.
- Users can easily swap in their own customized instances.

Solver libraries

```
static
double clawpack46_update(fclaw2d_global_t *glob, fclaw2d_patch_t *patch,
                         int blockno, int patch, double t, double dt,
                         void* user) {
   fc2d_clawpack46_vtable_t*  claw46_vt = fc2d_clawpack46_vt();
   ....
   claw46_vt->b4step2(glob, patch, blockno, patchno, dt);

   double maxcfl = clawpack46_step2(glob, patch, blockno, patch, t, dt);

   claw46_vt->src2(glob, patch, blockno, patchno, t, dt);

   return maxcfl;
}
```

- Explicit solver library only sees data on individual patches.
- Solver library can have its own virtual table.

Solver libraries

```
double clawpack46_step2(fclaw2d_global_t *glob, fclaw2d_patch_t *patch,
                        int blockno, int patchno, double t, double dt) {
    ...
    int mx, my, mbc;
    double xlower, ylower, dx, dy;
    fclaw2d_clawpatch_grid_data(glob, patch,&mx,&my, &mbc,
                                &xlower, &ylower, &dx, &dy);
    double *qold, meqn;
    fclaw2d_clawpatch_soln_data(glob, patch, &qold, &meqn);
    ...
    CLAWPACK46_STEP2_WRAP(&maxm, &meqn, &maux, &mbc, clawopt->method, ...,
      claw46_vt->fort_rpt2, claw46_vt->flux2, block_corner_count, &ierror);

    return maxcfl;
}
```

Legacy code called here

- Call-backs wrap legacy code.
- Patch data stored in an object that knows about data layout on a grid. For Clawpack, this is stored in a cell-centered “Clawpatch”.

References on time stepping

- Time stepping on AMR grids is a niche area in a much larger industry devoted to multi-rate time stepping.   (A. Sandu, Virginia Tech, D. Ketcheson (KAUST) and many others)
- References to early papers out of LBL offer best description of how local time stepping for AMR is done. See for example, papers by LBL group on projection methods (Almgren, Bell, Colella and others).
- Most time stepping assumes single step method; multi-step methods are more challenging (and not widely used by AMR community) when meshes are dynamically evolving
- A few more recent papers describe multi-stage methods, but little is known about how best to implement additive RK methods on AMR meshes.
- Classic problem : Experts in time stepping do not routinely develop ideas in complex AMR codes. Exception : C. Woodward (LLNL) works closely with AMReX team.

Patch libraries

- Solver libraries encapsulate details of a specific solver. These interact with ForestClaw core routines mainly through an update function.
- To update, however, solvers need patch meta-data, solution data, and knowledge of the data layout in memory.
- These details, and most other of the details of AMR are encapsulated in "**patch libraries**".
- Patch libraries describe how data is stored in the quadrant - cell-centered, node-centered, number of fields, and so on
- Tagging routines, ghost exchange, parallel halo exchange, data exchange, interpolating, averaging between grids are all encapsulated in a patch library.
- The patch routines in the ForestClaw core routines virtualize this patch functionality.

Patch libraries

- Very few routines in the core ForestClaw patch virtual table are assigned by functions in the solver (update, boundary conditions, auxiliary data)
- Most AMR functionality relies on virtualized functions in specific patch library.
  - ---Tagging cells for coarsening and refinement
  - ---Averaging, interpolating and copying between neighboring grids
  - ---Averaging and interpolation after regridding
  - ---Packing communication buffers for parallel exchange
  - ---Re-constituting patch data after re-partitioning.
  - ---metric terms for mapped grids
- The AMR logic guides *when* to do the above; patch library provides details on *how* to do the above.

Patch : virtual table

```
struct fclaw2d_patch_vtable
{
    /* Creating/deleting/building patches */
    fclaw2d_patch_new_t                     patch_new;
    fclaw2d_patch_delete_t                  patch_delete;
    fclaw2d_patch_build_t                   build;
    fclaw2d_patch_build_from_fine_t         build_from_fine;
   ....
    /* Ghost packing functions (for parallel use) */
    fclaw2d_patch_ghost_packsize_t          ghost_packsize;
    fclaw2d_patch_local_ghost_pack_t        local_ghost_pack;

    fclaw2d_patch_remote_ghost_build_t      remote_ghost_build;
    fclaw2d_patch_remote_ghost_unpack_t     remote_ghost_unpack;
    fclaw2d_patch_remote_ghost_delete_t     remote_ghost_delete;
    ...
    /* Plus about 40 others */
}
```

- These functions must all be defined by specific patch layout.

Example : Clawpatch

```
void fclaw2d_clawpatch_vtable_initialize(int claw_version) {
    fclaw2d_patch_vtable_t *patch_vt = fclaw2d_patch_vt();
    ...
    patch_vt->ghost_packsize        = clawpatch_ghost_packsize;
    patch_vt->local_ghost_pack      = clawpatch_local_ghost_pack;
    patch_vt->remote_ghost_build    = clawpatch_remote_ghost_build;
    patch_vt->remote_ghost_unpack   = clawpatch_remote_ghost_unpack;
    patch_vt->remote_ghost_delete   = clawpatch_remote_ghost_delete;
  ...
   clawpatch_vt->is_set = 1;
}
```

- A "clawpatch" used by the Clawpack solvers
- Defines layout as cell-centered, with either fields first or fields last in IJ ordering.
- The patch library defines how to pack and unpack parallel ghost “leaves” - halo of leaves around each processor - for parallel communication

Building a solver library

For an explicit time stepping solver :

- Define required virtual functions
- Define patch object that the solver will interact with
- Example : See *fc2d_clawpack4.6* solver library extension
- Example : See *fclaw2d_clawpatch* patch library extension

Building ForestClaw extensions

There are a lot of routines! How to proceed?

- **Step 1 :** Wrap your legacy code with a simple function that can be called from main. Get things to compile. This should involve almost no ForestClaw core routines (main + few others).
- **Step 2 :** Define a patch object with minimal functionality so code on a single grid works (nothing adaptive, no ghost exchanges, nothing parallel). This should involve core time stepping routines, but only over a single patch.
- **Step 3** : Slowly build in uniform refinement capabilities (only requires copying between grids; no averaging or interpolation; no regridding). Time stepping now over multiple patches
- **Step 4** : Add mechanisms for ghost cell exchanges between grids at different levels. Add tagging routines so grids can be adaptively refined.
- **Step 5** : Add packing and unpacking routines, and routines needed to rebuild quadrants after reconstruction. This should parallelize code.
- **Step 6** : Build options package for library so parameters can be set and registered in main registry and retrieved when needed.

What next?

- Coordinating ghost filling in parallel (surprisingly complicated)
- ForestClaw on GPUs (surprisingly easy)

Other topics I have not touch on :

- How are multiblock meshes set up in ForestClaw? (torus, cubed sphere, brick domains, disks, and so on). See numerous examples in applications/clawpack/advection.
- Option packages for configuring library extensions (.ini files with [sections]; all available as command line options)