These are my notes on how Amrex works and on the modifications I've made. I've mainly included information that can't be found from Amrex's own docs: https://amrex-codes.github.io/amrex/docs_html/index.html. Folder **amr_particles** contains a simple simulation which propagates particles in x-direction. Simulation grid is refined based on how the number of particles each cell contains. Amr-class in amr_particles/Src replaces the default AMReX_Amr class.

Compiling Amrex

See https://amrex-codes.github.io/amrex/docs_html/BuildingAMReX.html#building-libamrex
In command line:

```
./configure
```

Set parameters you want in **GNUmakefile**:

```
USE_OMP = TRUE
USE_PARTICLES = FALSE
```

```
make
make install
```

make install creates a file amrex.pc in foldertmp_install_dir/lib/pkconfig/ which contains compile flags needed by amrex.

Example Makefile:

```
AMREX_LIBRARY_HOME ?= [AMReX library path]

LIBDIR := $(AMREX_LIBRARY_HOME)/lib
INCDIR := $(AMREX_LIBRARY_HOME)/include

COMPILE_CPP_FLAGS ?= $(shell awk '/Cflags:/ {$$1=$$2=""; print $$0}' $(LIBDIR)/pkgconfig/amrex.pc)
COMPILE_LIB_FLAGS ?= $(shell awk '/Libs:/ {$$1=$$2=""; print $$0}' $(LIBDIR)/pkgconfig/amrex.pc)

CFLAGS := -I$(INCDIR) $(COMPILE_CPP_FLAGS)
LFLAGS := -L$(LIBDIR) $(COMPILE_LIB_FLAGS)

all: g++ -o main.exe main.cpp $(CFLAGS) $(LFLAGS)
```

Makefile in amr_particles worked on my laptop but I wasn't yet able to compile in Puhti.

Basic Amrex classes

(More in https://amrex-codes.github.io/amrex/docs_html/Basics_Chapter.html)

- **Initialize (argc, argv)** and **Finalize()**
- **Vector, Array4, GpuArray** (these matter when using GPUs)
- **ParallelDescriptor** (MPI interface)
- **ParmParse** reads command line and input-file arguments
- **Box** is defined by two **IntVect** corners, **BoxList**

- **RealBox** defines the physical domain**, Geometry** defines the problem domain (geom.Domain) and, for example, periodicity (geom.periodicity),
- **BoxArray** is global collection of boxes.
    - For advanced users, AMReX provides functions performing the intersection and complements of a `BoxArray` and a `Box`. These functions are much faster than a naive implementation of performing intersection of the `Box` with each `Box` in the `BoxArray`.
    - If one needs to perform those intersections, functions `amrex::intersect`, `BoxArray::intersects` and `BoxArray::intersections` should be used.
    - For excluding certain areas use `amrex::complementIn` and `BoxArray::complementIn`.
- **DistributionMapping** defines how data-boxes (Fabs), given a BoxArray, are distrubuted among MPI processes. Distribution strategies included in Amrex are SFC, KnapSack and RoundRobin.
- **BaseFab,** FArrayBox (=BaseFab<float>)
- **FabArray,** MultiFab (=FabArray<FArrayBox>)
- **MFIter i**terates through local fabs
- **For / ParallerFor** iterates through cells in fabs.
- **Debugging (** `DEBUG=TRUE` in makefile)

Class hierarchy in amr_particles/Src

`Amr` (implements `AmrCore`)

➔ **`AmrLevel`**

➔ **`CellFabArray`** (implements `FabArray<CellFab>`)

➔ **`CellFab`** `= BaseFab< std::shared_ptr<Cell> >`

➔ **`Cell`**

Amr handles level creation, regridding and load balancing. Levels go from 0 to amr.finestLevel() can be accessed by amr[level]. Level 0 grid is created by **defBaseLevel** and all others by **bldFineLevels.** Data should be initialized immediately after defBaseLevel because finer levels are filled using base level.

AmrLevel a container for a grid of a single refinement level. and other level specific data, for instance boundaries. Tagging for refinement also happens here.

CellFabArray is a global container shared by all MPI processes. BoxArray and DistributionMapping define how data is distributed.

CellFab is local container of data defined by a rectangular Box. Each MPI process has 1 or more CellFab. Around the valid region of CellFab are predefined number of ghost cells. By default, Amrex does not directly connect ghost cells to other local CellFabs and cell data has to be therefore

copied every cycle. For efficiency, cells are referred to by **shared_ptr<Cell>**. (shared_ptr keeps a count of owners and deallocates Cell-data if it has owner count drops to zero.) Pointers of local ghost cells are connected to neighbour fabs during **AmrLevel::init().**

Cell is the user defined datatype.

In addition **Interpolater** is needed for communication between levels.

At least the following functions have to implemented for Amr to work:

- `Amr::ErrorEst` and `ManualTagsPlacement` tag cells for refinement during regrid.
- `CellFabArray::FillPatchSingleLevel` fills grid from cells of source grid using ParallelCopy (similarly to FillBoundary)
- `Interpolater::interp` and `Interpolater::average`
- `Amr::FillAllBoundaries` is an example on how to update ghost cells and can be replaced by more specialized functions.
- `get_mpi_datatype` in Cell.

Iteration through cells

```
for (int lev = 0; lev <= amr.finestLevel(); lev++)
{
      AmrLevel& level = amr[lev];
      CellFabArray& cells = level.getCells();

      //#pragma omp parallel
      for (AmrIter it(level, do_tiling=false); it.isValid(); ++it)
      {
            Box box = it.validbox();
           CellFab& fab = level[it]; // or cells[it];
           Array4<Cell*> cell_array = fab.array();

            // iterate through cells
           amrex::For(box, ncomp, [&] (int i,int j,int k,int n)
            {
                 // two ways of accessing data:
                 IntVect iv(i,j,k);
                 Cell& c = *fab(iv, n);
                  // or
                  Cell& c = *cell_array(i,j,k,n);
            });
      }
}
```

For is equivalent to three for loops:

```
for (int k = lo.z; k <= hi.z; ++k) {
for (int j = lo.y; j <= hi.y; ++j) {
for (int i = lo.x; i <= hi.x; ++i) {
     f(i,j,k); // lambda function
}}}
```

ParallelFor works similarly but also includes vectorization. More looping functions can be found in AMReX_Loop.H and AMReX_GpuLaunchFunctsC.H and Amrex allows looping to be specified to run on CPUs or GPUs. For more on how to run Amrex on GPU see https://amrex-codes.github.io/amrex/docs_html/GPU.html

it.validbox() returns the valid region. This is not the same as **fab.box()** which includes ghost cells. Within the loop cells can be accessed either directly through the Cellfab or by creating an Array4. (One might be faster than the other, so this should be tested.)

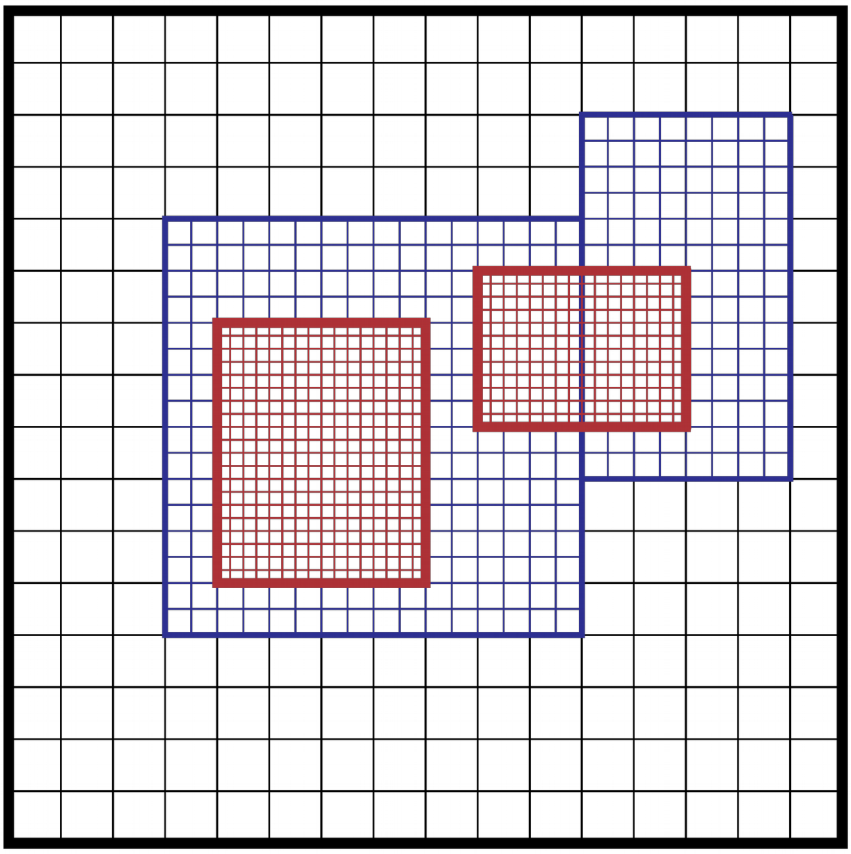
ncomp is an optional parameter if cells have multiple components.

To enable iteration with multiple threads, **`#pragma omp parallel`** must be added before MFIter/AmrIter and tiling must be enabled (optional parameter do_tiling=true). Tiling chops the iterated region so that each thread receives a different validbox which don't overlap. Note, that this is designed to be used for simpler datatypes, and in practice `#pragma omp parallel` might be better to add once inside For.

AmrIter inherits Amrex class **MFIter.** It is initialized by giving AmrLevel as an argument and an added method it.**childLID()** which returns the local id to CellFab of finer level if one exists. If region has no children -1 is returned. Note: **cells[it]** == **cells[it.index()]** where it.**index** is global index to CellFab. Local index to data is returned by it.**LocalIndex**() and children must be accessed with fine_cells.**atLocalIndex()**. List of children indexes is made during **constructCrseFineBdry.**

Example of AMR Grids

(From https://amrex-codes.github.io/amrex/docs_html/Basics.html#example-of-amr-grids)
In block-structured AMR, there is a hierarchy of logically rectangular grids. The computational domain on each AMR level is decomposed into a union of rectangular domains. Figure below shows an example of AMR with three total levels. In the AMReX numbering convention, the coarsest level is level 0. The coarsest grid (*black*) covers the domain with 16 cells. Bold lines represent grid boundaries. There are two intermediate resolution grids (*blue*) at level 1 and the cells are a factor of two finer than those at level 0. The two finest grids (*red*) are at level 2 and the cells are a factor of two finer than the level 1 cells. There are 1, 2 and 2 Boxes on levels 0, 1, and 2, respectively. Note that there is no direct parent-child connection.



FillBoundary

Updating the ghost cells between CellFabs in different MPI processes happens by calling CellFabArray::**FillBoundary().** Possible parameters are:

- `int scomp` (start component, 0 by default)
- `int ncomp` (number of components, by default all are filled)
- `IntVect &nghost` (which ghost cells to fill, all by default)
- `Periodicity &period` (which boundaries to fill periodically, by default NonPeriodic)
- `bool cross = false` (if true, corner cells are not filled)
- `bool enforce_periodicity_only = false` (true fills only periodic boundaries)

FillBoundary can be split into function **FillBoundary_nowait()** and **FillBoundary_finish()**.

In CellFabArray I added methods **get_receive_tags()** and **get_send_tags()** for a quick access to cells that are communicated by **FillBoundary**. Note that those receive and send tags of CellFabArray don't exist until FillBoundary has been called at least once.

Iteration over tags should be done like this:

```
for (auto const& kv : get_receive_tags())
{
     //int sender_rank = kv.first;
     for (auto const& tag : kv.second)
     {
          // destination box
          const Box& bx = tag.dbox;
          // source box
          const Box& bx = tag.sbox;
          // destination of data
          auto data = sfab(tag.dstIndex);

          // Loop/For/ParallelFor
          For( bx, ncomp,
          [&] (int i, int j, int k, int n) noexcept
          {
               // do something
          });
     }
}
```

The following are technical details which might be useful if you need add flexibility to FillBoundary (Amrex doesn't document this):

- **FabArrayBase::FB** is a container for information about the owners and locations of ghost cells (remote and local) and is used during FillBoundary**.**
- FB is returned by **getFB()** which takes the same arguments as FillBoundary. Amrex saves all new FBs and subsequent getFB-calls return the saved FB as long as the CellFabArray exists.
- get_receive_tags() and get_send_tags() only return tags to the most recently called FB. These should be modified to support multiple FBs if FillBoundary is used with more than one set of parameters,
- Because Cells are connected using shared_ptr, FillBoundary doesn't do any local copies though getFB still creates local tags. If for some reason local copying is needed, uncomment in function FillBoundary_nowait (in file CellFabArray.cpp) sections that contain FB_local_copy_cpu.

Regridding and Load Balancing

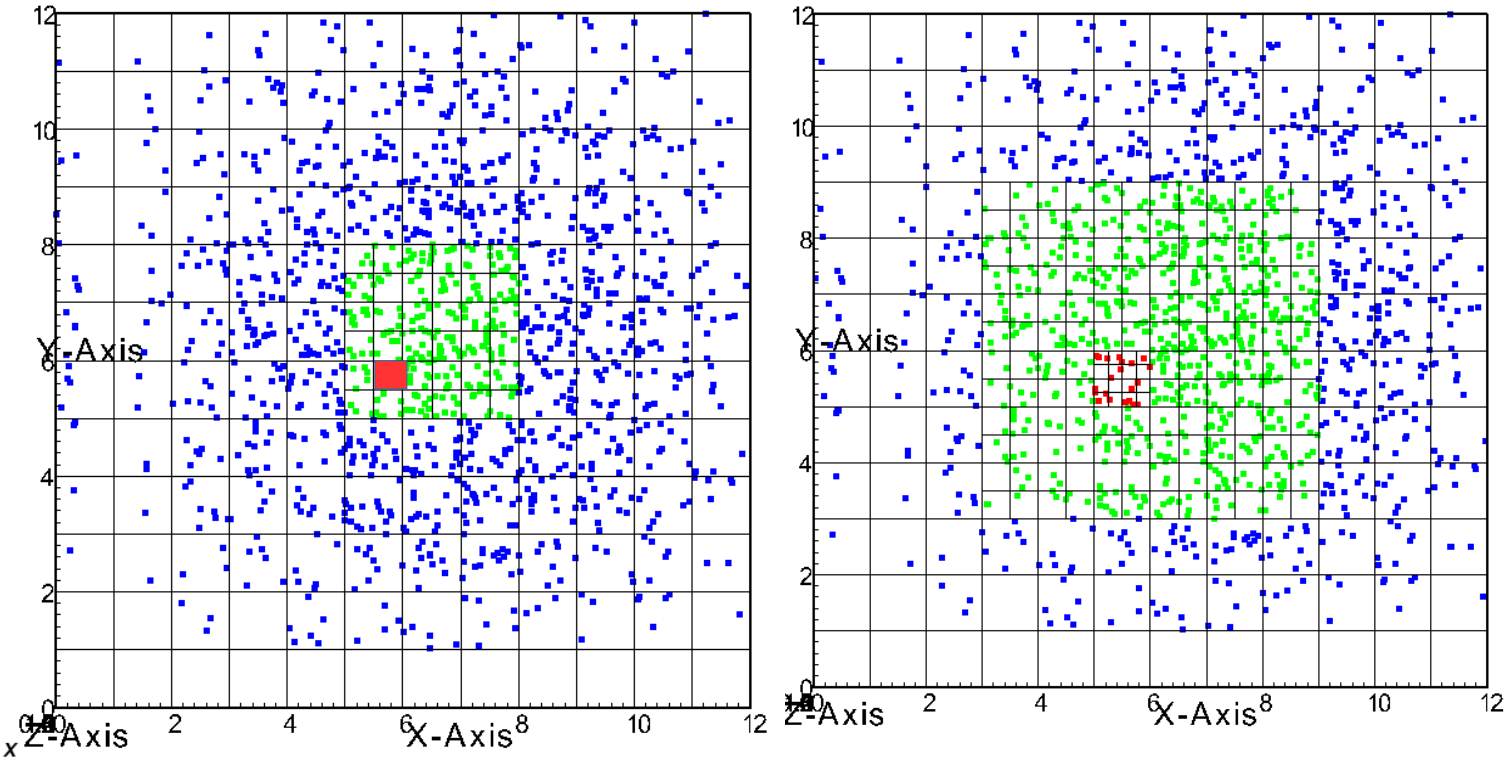
Amr::regrid() handles gridding and load balancing:

- **MakeGrids** uses funtions MakeBaseGrids and MakeNewGrids (found in AmrMesh.h) to create the new grid distribution.
- MakeNewGrids calls **ErrorEst** to tags cells for refinement. The tags are marked into a **TagArrayBox** (possible values are TagBox::CLEAR and TagBox::SET). By default all cells have CLEAR-value. Therefore if a cell needs to stay refined it must be also tagged here by SET
- **ManualTagsPlacement** is called after Amrex has added buffer tags and can be for instance used to limit which regions are refined.
- **MakeDistributionMaps** chops grids created by Amrex so that overlapping cells of different level are all owned by the same process. Cells are weighted based the level of refinement and DistributionMap is created based on these weights.
- **InstallNewLevels** creates new AmrLevels matching the new distributions and moves the data from old levels by calling **FillPatch()** which moves data around using **ParallelCopy** and **Interpolater**. ParallelCopy works similarly Fillboundary, copying local and remote data to new CellFabArray. (tags to send and received cells are returned by struct **CPC** which works equivalently to **FB**)

Grids for some or all levels can also defined by a fixed grid file, though I haven't tested how this works. Possible operations to do after regridding should be added to **AmrLevel::post_regrid.** Currently coarse-to-fine and fine-to-coarse boundaries are made there.

Remote ghost cells are not filled by regrid, therefore FillBoundary should be called immediately after it.

Some important input file parameters are **blocking_factor** forces grids to divisible by the given value, and **n_error_buf** which defines how many cells to refine around the already tagged cells.



Above picture show an interesting example on how Amrex calculates new grids.

What happened here was:

1) Cells (5,5) to (7,7) were tagged at level 0. No buffering
2) The red cell (5.5, 5.5) was tagged at level 1.
   - ➔ level 2 is buffered to so that blocking_factor is 4.
   - ➔ level 1 is buffered to avoid borders between levels 0 and 2.

Even though in inputs file **amr.n_error_buf** is 0. Amrex still add buffering, propably to make sure there is no overlap between levels 0 and 2. (If this is too much clear use ManualTagsPlacement or rewrite MakeNewGrids)

Fine/Coarse level boundaries

(Communication between two levels must implemented in **Interpolator.h**)

AmrLevel::constructCrseFineBdry creates which **BoundaryContainer**-struct in AmrLevel contains the ghost cells Interpolater should fill. To safe memory, BoundaryContainer only contains data local to the process.

Outer/inner boundaries could be similarly defined by making a BoxList containing cells of the boundary. BoundaryContainer doesn't know which CellFabs data is send to. More efficient might be to use **FabArrayBase::CopyComTag** to only save boxes and indexes in CellFabArray.

Some useful methods for Box and BoxList manipulation:

- **refine, coarsen**
- **intersect**
- **complementIn**
- **removeEmpty, removeOverlaps** (afterwards)

KNOWN ISSUES

- periodic FillBoundary does not currently work on levels greater than 0. These boundary cells should therefore not be refined
- Periodicity may cause issues on when number of processes is small (<=4)
- FillBoundary tries update data in between empty buffer cells (aka. cells with children). This might unnecessary slow down communication between processes.
- In corners, Fillboundary might attempt to copy data to same ghost cell more than once. (This should not cause other issues except being inefficient.)
- DistributionMapping strategies provided by Amrex might not be ideal because they don't seem to consider the current ownership of cells. Sometimes load balancing is therefore inefficient. (Note: mapping is calculated in MakeDistributionMaps using makeSFC)
- Simulation sometimes fails when max_level is higher than 1 and number of MPI processes is 2. (Something do with initialization.)
- MakeDistributionMaps, InstallNewLevels and constructCrseFineBdry might have bugs I haven't noticed.