Imogen Programming Reference

Erik Keever

September 23, 2015

1 Overview

Imogen, as a standalone program, operates under the classic input-processing-output model:

- Input - The generation of the initial conditions $\vec{\psi}(\vec{x}, t = 0)$ on a grid $\Omega \in \mathbb{R}^d$ of space dimension d, with $\vec{\psi} = \{\rho, \vec{v}, P\}$ (or equivalent), and the definition of boundary conditions $\vec{\psi}(\partial\Omega, t > 0)$. Result: A complex structure, or string naming the file containing same.
- Processing - Calling imogen() which evolves the initial conditions forward in time as the input structure indicates
- Output - The post-run analysis of the evolved state

At rough count, Imogen is roughly 15000 lines each of Matlab and CUDA. The objective of this development manual is to help other developers gain a thorough sense of what goes in in Imogen and where it is done.

Several procedures, especially the definition of boundary conditions, weave an unpleasantly convoluted path through the execution's tapestry.

2 Initialization

The initialization part of Imogen (leading up to calling `imogen()`) is summarized in figure **??**.

2.1 First steps

Key files: `run/imogen, run/parImogenLoad.m, run/starterRun.m`

There are three ways that Imogen can be started from terminal:

- `./imogen serial run.m S [GPUs]`
- `./imogen parallel run.m S [GPUs] np`
- `./imogen cluster run.m S [GPUs] nNodes ppn queue np`

All of them result in one or more instances of Matlab being invoked and told to run `parImogenLoad(runFile, logFile, alias, gpuSet)`.

It is also possible to drive it directly from within an interactive instance by calling `parImogenLoad()`, in the form `parImogenLoad(runFile,'','',gpuSet,'anything')`. The 5th argument, simply by

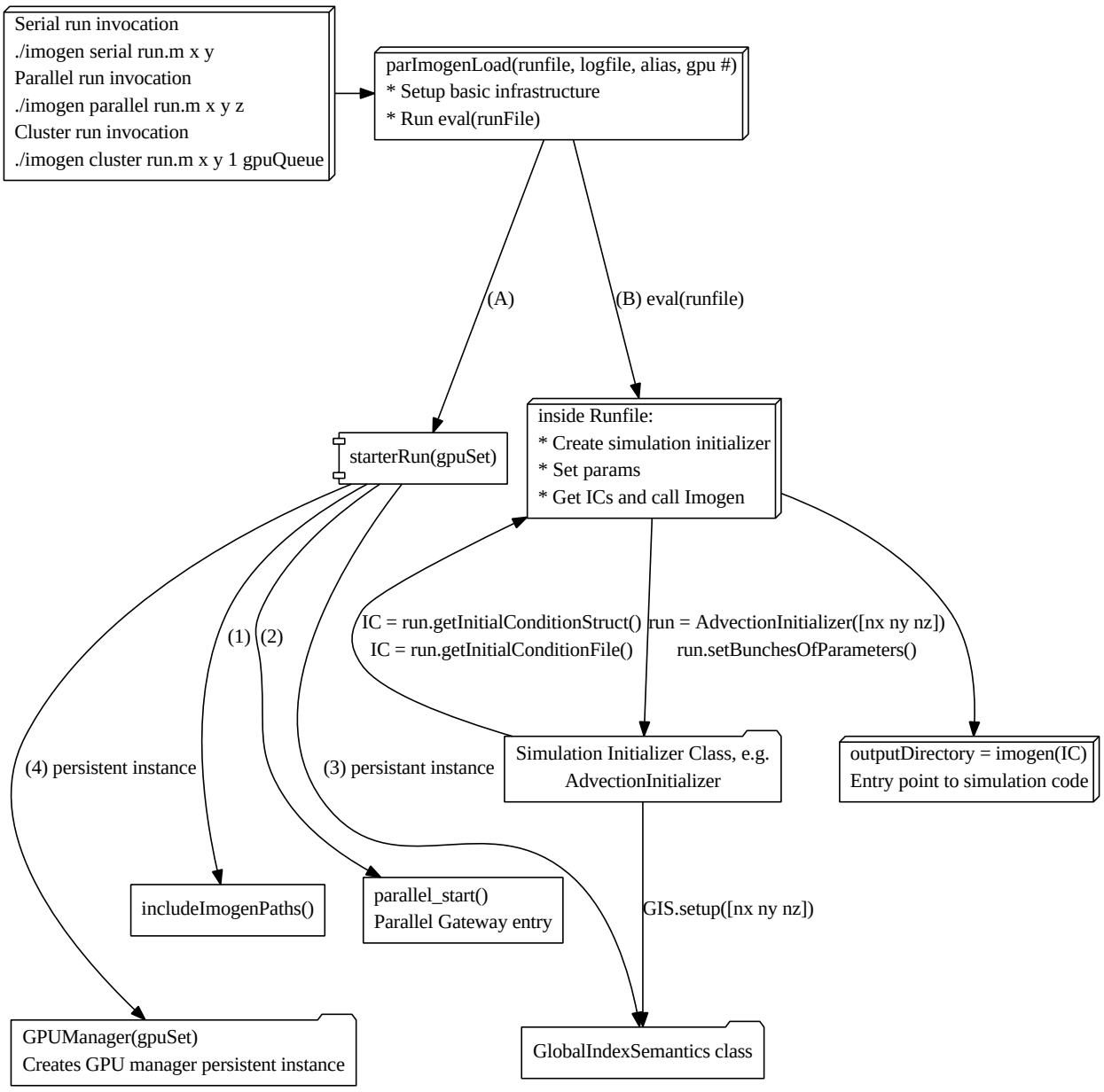

Figure 1: GraphViz sketches for us roughly how we get to the imogen() call

its existence, tells parImogenLoad not to perform several finalizing tasks upon finish (most notably, `MPI_Finalize`).

`parImogenLoad` first calls `starterRun` which:

- Finds and calls `includeImogenPaths`
- Calls `parallel_start()` to initialize MPI and start PGW
- Creates the persistent instance of `GlobalIndexSemantics`
- Creates the persistent instance of the GPUManager (`gpuclass/GPUManager.m`) & sets the GPU distribution parameters

TO FACILITATE DEBUGGING OF COMPILED CODE:

- Uncomment a `debugSpin()` line in `starterRun`
- Attach debugger
- Debugger will probably stop in a random place: in GDB, cont followed by ctrl-C brings me to the debug spin line

If the standing up of infrastructure is successful, `parImogenLoad` then does `eval(runFile)`.

Upon successful exit from a scripted run (one with only 4 arguments to parImogenLoad), it will reset the GPUs, delete all persistent data structures, **and call** `MPI_Finalize`.

2.2 The Runfile

Key files: `run/run_*.m, experiment/Initializer.m`

The process of generating initial conditions is of great complexity and variety. Rather than end up writing our own (inevitably broken, buggy, badly-implemented) script language to specify them, they are generated by a Matlab script.

Once the Runfile is called, basic setup has been completed (GPUs and MPI are ready to be talked to). Because it's called and run as a Matlab script, the Runfile is free to do whatever it wants. This is demonstrated in the `full_test_suite` runfiles, which actually run multiple simulations and analyze all of them to check convergence orders.

Imogen has many kinds of simulations. Their setup is defined by their initializer classes. These follow a 'SimulationtypeInitializer' naming convention and inherit from the Initializer master class.

Once the Runfile creates an initializer, there follows a great deal of run-specific code to set all the parameters for the simulation.

Finally, the initial conditions are requested (in the form of a gigantic structure).

The process is summarized:

- `run = SimtypeInitializer(resolution);`
- (bunches of parameter-setting customness)
- Two methods of preparing a simulation:
  - `IC = run.saveInitialCondsToStructure();`
  - `IC = run.saveInitialCondsToFile();`

- `outputDataPath = imogen(IC);`

The outcome of the two IC-getting functions is the same. There are two depending on whether memory or speed is more important: If the IC is fetched as a structure, and passed to `imogen()`, then it's going to continue to occupy memory until `imogen()` returns!

If the resolution is small, having an extraneous copy of the ICs floating around in host memory is not a problem, and it will always be faster to pass the struct than do a complete write to disk & read from disk.

If the resolution is large and the system's GPUs are impressive, one copy of the ICs may become an appreciable fraction of system memory. Example: A node with two K80s has 24GB of device memory; It will be able to accomodate initial conditions up to just short of 12GB (The fluid step process unavoidably requires a second copy of the full fluid state). Even on modern GPU clusters, wasting 12GB of host memory per node is nontrivial. The file option avoids this... but you have to do 24GB of IO.

(FIXME: Use of /dev/shm ramdisk would avoid this)

2.3 The Initializer

Key files: `experiment/*/*Initializer.m, experiment/PotentialFieldInitializer.m,`

`experiment/RadiationSubInitializer.m, experiment/SelfGravityInitializer.m, experiment/StaticsInitializer.m`

An initializer for a simulation inherits from the master Initializer() class. The master defines all the required properties which are used by Imogen. The simulation initializer class will usually define a bunch of simulation parameters which it uses for its own edification when generating initial conditions.

All new simulation initializers must implement the following exact prototype: `function [mass, mom, ener, mag, statics, potentialField, selfGravity] = calculateInitialConditions(obj)`

- mass is a double array of size [GIS.pMySize]
- mom is a double array of size [3 GIS.pMySize]
- ener is a double array of size [GIS.pyMySize]
- mag is either a double array of size [3 GIS.pMySize] or []
- statics is a StaticsInitializer() or []
- potentialField is a PotentialFieldInitializer() or []
- selfGravity is a SelfGravityInitializer() or []

The `Initializer()` class' `saveInitialCondsTo*` call will use this function to perform the generation of initial conditions.

3 imogen()

Key files: `imogen.m, initialize.m, ../save/initializeResultPaths.m, flux/flux.m, sources/so`
`save/resultsHandler.m`

After finally generating the initial conditions, `imogen()` is finally called.

The operation of `imogen(IC)` is:

- `run = initialize(ini)`: Parses and checks the contents of all the input scalar parameters
- `initializeResultPaths(run, IC)`: Either creates the output directories, or notes that we are resuming.
- If resuming a run: Loads the output data frame and loads 'initial' conditions from it
- Uploads the initial conditions to the GPUs
- Initializes the self-gravity system (SG is broken, but compact gravitating bodies are also run under this aegis)
- If in-situ analysis is being used: calls the `inSituAnalyzer.setup()` function
- Enter the fluid dynamic loop
  - Determine CFL-constrained time step (`time/TimeManager.m:update`)
  - Call forward `flux(run, mass, mom, ener, mag, 1)` CFD sweep
  - Call `source(mass, mom, ener, mag, tFraction)`
  - Call backward `flux(run, mass, mom, ener, mag, -1)` CFD sweep
  - Check whether to save data (`save/resultsHandler.m`)
  - Check whether to run in-situ analysis
- If using in-situ analysis, call `inSituAnalyzer.finish()`

The operation of the flux sweep (`flux/flux.m`) is as follows. The flux is called twice per timestep, forward and backward, implementing the Strang operator splitting to make the one-dimensional approximate Riemann solvers multi-dimensional.

- Permute XYZ to ABC. For each:
- Reorient arrays so flux is in the stride-of-1 direction
- Do flux (`relaxingFluid`)
- Return to normal orientation
- Exchange MPI halos (`cudaHaloExchange.mex`)

The procedure of the source (`sources/source.m`) function. Nominally one might imagine that the source terms themselves must follow the ABBA splitting among each others, however this is only the case for operators that do not commute.

- Rotating frame: `cudaSourceRotatingFrame`

- Compact objects: `cudaAccretingStar`
- Stationary gravity field: `cudaSourceScalarPotential`
- Radiation: `run.fluid.radiation.solve`
- exchange MPI halos (`cudaHaloExchange.mex`)
- apply boundary conditions

4 Output/Postprocessing

5 Subsystem: Boundary Conditions

The implementation of boundary conditions is probably the most convoluted part of Imogen.

6 Subsystem: Tour of cudaCommon.cu

The giant `gpuclass/cudaCommon.cu` file implements the Multi-GPU Array (MGA) handling functionality that makes talking to multiple devices in the rest of the code relatively sane.

7 Subsystem: timing/time management

8 Subsystem: Where things are printed to log

Especially at the first stage of debugging, knowing where the code was just after the last non-error was printed is very helpful. This part of the guide will dissect a sample log file and list where the code was at that point.