All instructions herein assume being on an HPCC dev node.

## Building

```
# Make serial and MPI binaries
. load-modules.sh
make

# Make binaries with timing output
. load-modules.sh
make BENCH=1

# Make binaries with debugging output
. load-modules.sh
make DEBUG=1

# Compile timings from program output in timings/raw/
. load-modules.sh
make timings

# Make fine-serial timing plot and MPI scaling plot
. load-plot-modules.sh
make all-plots

# Make visualization `rumors.gif` of program output. Assumes images are in ./images
make gif
```

## Running

The serial version is run as `./mill`, and outputs to the directory `./images`.

The MPI version is run al `./mill.mpi <prefix> <x_size> <y_size>`, where `prefix` is the output directory and `x_size`, `y_size` is the size of the simulation.

If build with `BENCH=1`, both output timing information to stderr.

## MPI Implementation Principals

The original serial program was modified to use MPI mainly in the following five ways: - *Input*: To facillitate running many timing trials in parallel, an output prefix and simulation size are taken on the command line. We need to take an output prefix so that each trial may output put to a different location, else when they're running simultaneously they will be attempting to write to the same files (which would spoil the timing data). - *Initialization*: The rank 0 thread generates unique values to pass to each thread so each may uniquely seed its own random number generator. The rank 0 thread then generates a full world and sends evenly sized chunks to all other threads in rank order (chunked along the array columns since C arrays are row-major), with the thread of highest rank receiving a smaller chunk if the number of threads does not divide the world size. - *Edge Communication*: At each time step, each thread communicates its row-borders to the previous rank and next rank threads. - *Output*: Every 10 time steps (as in the serial program), each thread communicates its chunk back to the rank 0 thread, which then output an image under the prefix directory given on the command line. - *Timing*: Each thread keeps track of its own timings, and at the end

of the program an `MPI_REDUCE` instruction is used to get the maximum time across threads of each category for the rank 0 thread to then output.

# Timing

## Categories

The timing categories are:

- `total`: The total time.
- `init`: The time spent in initialization.
- `sim`: The time in the main loop devoted to the simulation itself.
- `edge_comm`: The time spent communicating edges.
- `file_io`: The time spent outputting images and doing the communication required to accomplish that.

## Serial

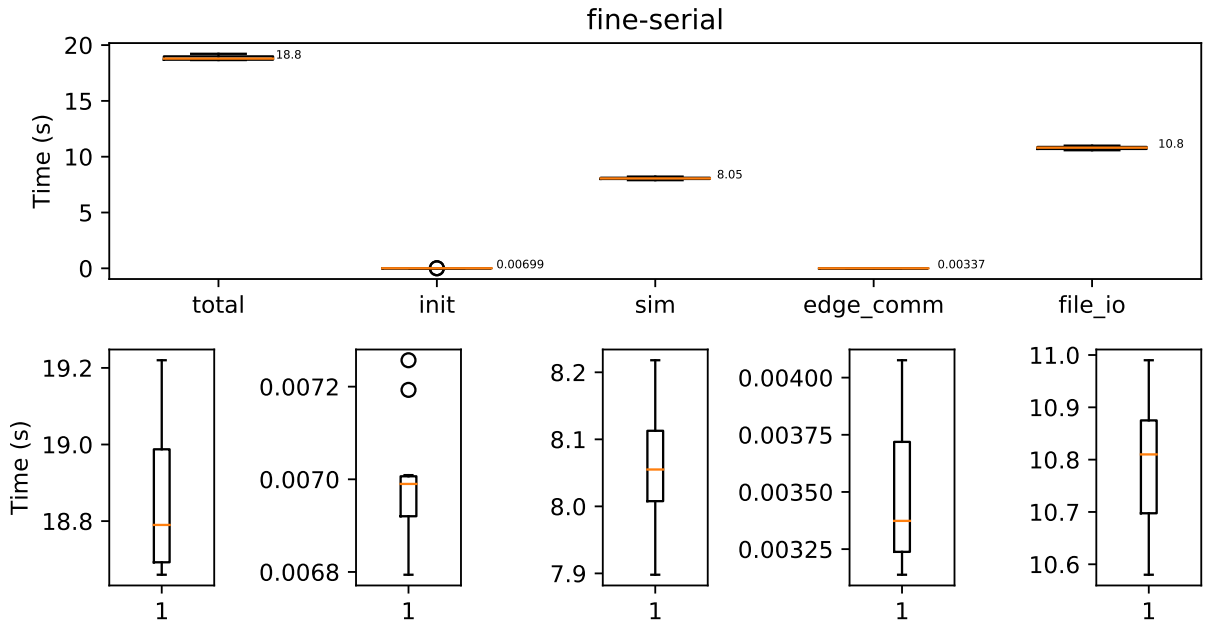
To generate the serial timing data:

```
. load-modules.sh
./mill 2>timings/raw/fine-serial.out
make plots/fine-serial.pdf
```

My trials were performed on the HPCC `dev-intel14` and are provided in `timings/compiled/fine-serial.out`. On average (in seconds):

| total | init | sim | edge_comm | file_io |
|---|---|---|---|---|
| 18.87+-0.07 | 0.007000+-0.000041 | 8.063+-0.031 | 0.00349+-0.00011 | 10.80+-0.04 |

The data can also be seen in `plots/fine-serial.pdf` with box-and-whisker plots, with all categories plotted together (with median values annotated) in the first row and separately in the second.



## MPI

To generate the MPI timings for 1, 2, 4, 16, 32, 64 threads and `x_size` = 1000, `y_size` = 500, 1000, 2000, 4000, 8000, 16000:

```
./launch-mpi-timings.sh
# ...wait for SLURM jobs to complete...
. load-plot-modules.s
make plots/mpi-scaling.pdf
```

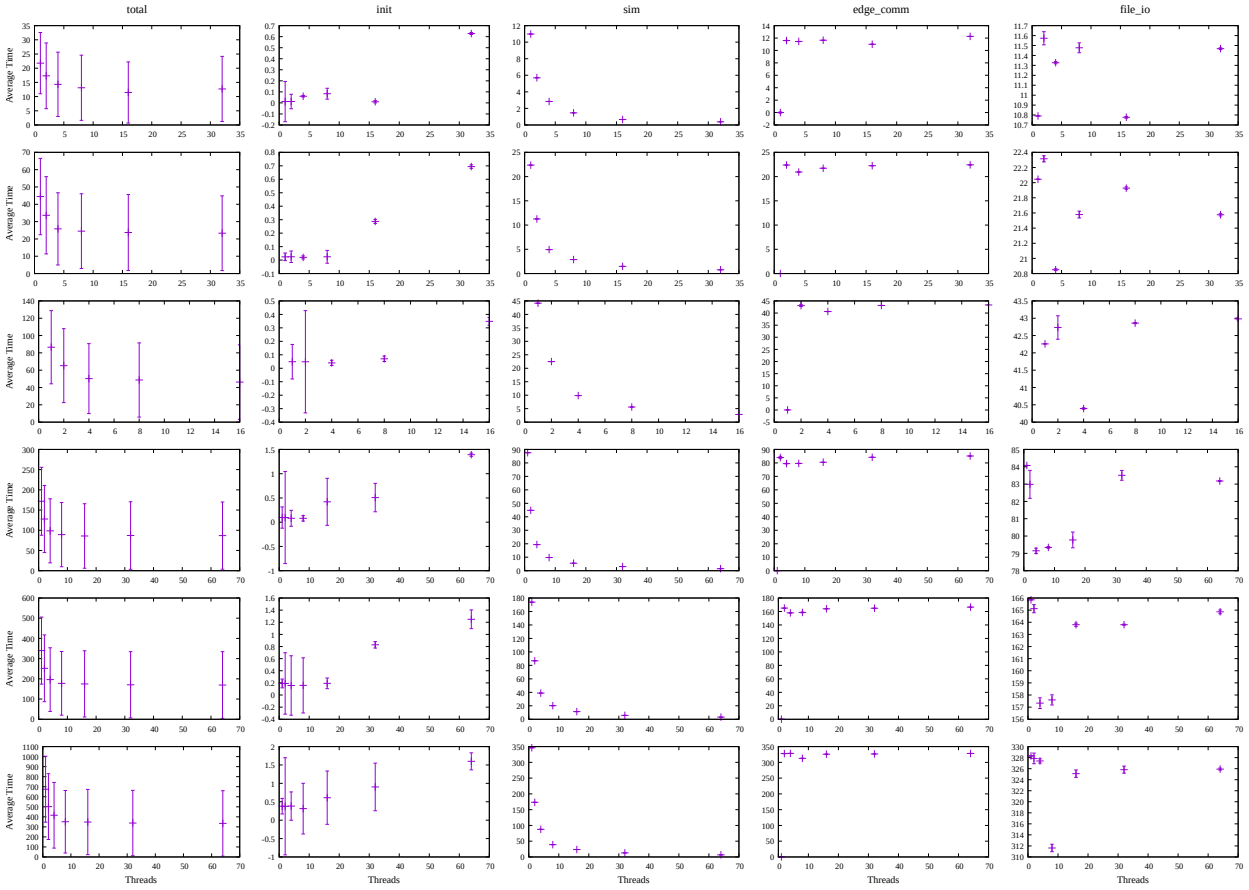
Note that this creates a slew of SLURM jobs.

My trials were performed on the HPCC `dev-intel14` and are provided in `timings/compiled/mpi.dat`, with averages extracted into `timings/compiled/mpi-avg.dat`. I had some weird errors crop up when doing these timings; specifically, *the configurations* `threads=64` *and* `y_size=500,2000,1000` *always fail. Occasionally* other configurations will fail in the same way, and I believe I had more consistent trouble when I was trying to do 128 threads (but I stopped doing that, so I don't know). This was across hours and hours of time, since I ended up redoing timings several times. The error is always (for example):

```
*** An error occurred in MPI_Recv
*** reported by process [1533739017,63]
*** on communicator MPI_COMM_WORLD
*** MPI_ERR_COUNT: invalid count argument
*** MPI_ERRORS_ARE_FATAL (processes in this communicator will now abort,
***    and potentially your MPI job)
```

This would potentially make sense if it was only consistent for a given configuration... but it's not, so I really have no clue.

The data is visualized with `plots/mpi-scaling.pdf`, which gives a matrix of scatter plots of average time vs. number of threads; columns correspond to timing categories, and rows correspond to simulation size.




We see what appears to be an exponential decay in the total amount of time, with the maximum of 64 threads always giving the best time on average but significant gains only up to about 8 threads. Note the high variance across trials and that, while the thread-scaling is independent of size, the absolute amount of time appears to scale linearly in the array size.

Initialization is pratically 0 until about 16 threads are involved, at which point it increases in what appears to be a linear fashion.

The simulation time is evidently the main source of the patterns seen in the total time, but it consistently only makes up about a third of the total time.

Edge communication is basically constant across number of threads, with the exception of 1 thread where it vanishes (since we are guaranteed only one node). It also scales linearly in the simulation size, and makes up another third of the total time.

Output is the most erratic, but I believe this to be dumb luck: we can sort of see two levels in the data, which can be explained as some sets of trials only being allocated one node (and so have a lower time) and some being allocated multiple. We again see linear scaling in the simulation size, and it makes up the last third of the total time.

# Potential Improvements

The code could be furthered improved mostly through improving the interthread communication. In particular, boundaries only need to communicate what changes. This could be accomplished by allowing process to launch into a receive loop, receiving changed points until they get a message telling them to stop. This would create more overhead when the boundaries are changing a lot, but I reckon that is the less common than

when the boundaries are static. The chunks that each thread works on could also be changed to span both rows and columns, which would give better scaling for larger simulation sizes.

We could also imagine explicitly adding "local" parallelization, through either say CUDA or OpenMP, which would allow each thread to accomplish more work.