

Parallel Computing/Programming Assignment #4 and #5: Massively Parallel Game of Life with MPI, Pthreads and Parallel I/O

Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
110 8th Street
Troy, New York U.S.A. 12180-3590
Email: `chrisc@cs.rpi.edu`

April 24, 2018

DUE DATE(no late days allowed): Wednesday, April 25th, 2018 at 11:59 p.m.

Note this version fixes various typos discussed in class. If you already have slightly different data points than described below, that is fine. Go with those and highlight them in the write-up with your explanation for the differences.

1 Assignment Description

This is a “double” assignment and so your grade will be counted twice (for assignment #4 and #5). You can have a team of up to 3 students.

For this assignment, you are to create an MPI implementation of *Conway’s Game of Life* to leverage Pthreads within compute nodes and conduct a strong scaling performance study on the IBM Blue Gene/Q. In particular, you will determine how much performance improvement is obtained by using Pthreads for intra-node shared-memory parallel vs. intra-node message passing using MPI.

1.1 Review: Basic Rules

The Game of Life is an example of a Cellular Automata where universe is a two-dimensional orthogonal grid of square cells (with WRAP AROUND FOR THIS ASSIGNMENT), each of which is in one of two possible states, *ALIVE* or *DEAD*. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur at each and every cell:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by over-population.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system using a Random Number Generator (RNG) at time 0. Use the RNG provided in the assignment4-5 code template (yes, a template this time!). The first generation is created by applying the above rules to every cell in the seed; births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a “tick”. The rules continue to be applied repeatedly to create further generations.

Note, a “tick” starts with $Cell(0, 0)$ and ends with $Cell(N - 1, N - 1)$ in the serial case. When we get to parallel things will get much more interesting! (see below).

1.2 Adding Additional Randomness

Using the uniform distribution provided in RNG (see the template on how to use this new RNG), if value return by the RNG is greater than a set THRESHOLD, then perform the above described basic rules for each cell. Otherwise, randomly pick state of *LIVE* or *DEAD*.

1.3 Parallelization Approach

To ensure that there is more true “randomness” among the cells, each row will have it’s own RNG seed set. According to the template, every MPI rank will initialize 16,384 RNG streams. These streams are about 2^{70} calls apart. **SO, EACH ROW WILL HAVE IT’S OWN RNG STREAM! This is important for your Pthreads implementation.**

For the parallelization approach, a Blue Gene/Q compute node for example will allocate 1 MPI rank only per node and then create 63 additional threads per nodes. All threads on each node (including the MPI rank-thread) will perform an even “chunk” of rows for the Cellular Automata universe. For example, suppose you have a 1024x1024 universe using 64 threads using a single Blue Gene/Q node. If you divide the universe equally, each thread will have 16 rows or 16x1024 cells to compute. Thus, thread 0 (on node 0), will compute rows 0 to 15, thread 1 computes rows 16 to 31 and thread 2 will compute rows 32 to 47 and so on.

Now when running using more than one Blue Gene/Q node, each MPI Rank only needs to allocate it’s specific chunk (which is operated on by all threads within that compute node) plus space for “ghost” rows at the MPI Rank boundaries (not thread boundaries). The “ghost” rows can be held outside of the main MPI Rank’s Cellular Automata universe.

Now, you’ll notice that rows at the boundaries of MPI Ranks need to be updated / exchanged prior to the start of computing each tick. For example, using the above 1024x1024 universe and 16 MPI Ranks as an example, MPI Rank 0 will need to send the state of row 0 (all 1024 entries) to MPI Rank 15. Additionally, Rank 15 will need to send row 1023 to Rank 0. Also, Rank 0 and Rank 1 will do a similar exchange.

Note - the above example is only a 1024 x 1024 universe and below you will be using the larger 4096 x 4096 cell universe.

For these row exchanges, you will use the `MPI_Isend` and `MPI_Irecv` messages. You are free to design your own approach to “tags” and how you use these routines except your design should not deadlock.

1.4 Parallel I/O

When indicated for a particular experiment, you are to write out the cell universe into shared single file. All MPI ranks will participate using the `MPI_file.write_at` operation. Make sure your implementation performs the `MPI_file.open` using the `MPI_COMM_WORLD` communicator to ensure the file pointer is properly shared across all MPI ranks. With 128 ranks, each rank will be responsible for 2,097,152 cells. You will want to make sure you compute each MPI rank’s offset correctly using the cell data size, number of cells in the universe per rank and the rank’s value.

1.5 Algorithm

So the algorithm becomes:

```
main(...)
{
    Setup MPI (template has this already!)
    if Rank 0, start time with MPI_Wtime.
    Allocate My Rank's chunk of the universe +
        space for "ghost" rows.
    Randomly initialize my universe using the
        right RNG stream for each row.
    Note: perform this serial within each MPI rank/thread group.

    Create Pthreads here. All threads should go into the for-loop.

    for( i = 0; i < number of ticks; i++)
    {
        Exchange row data with MPI Ranks
            using MPI_Isend/Irecv from thread 0 w/i each MPI rank.

        [Note: have only 1 MPI rank/pthread perform ALL MPI
        operations per Rank/Thread group.]

        Do rest of universe update making sure to use the
            correct row RNG stream &&
            use the right "ghost" row data at Rank boundaries

        HERE each PTHREAD can process a row
        Need to add pthread_mutex_trylock around shared access
        variables if needed.
    }
```

```

MPI_Barrier();

if Rank 0, end time with MPI_Wtime.

Perform an output of Cell Universe using MPI_file_write_at
    if needed by experiment
    Collect I/O performance stats using MPI_Wtime.

Print Compute (and I/O if needed) Performance Stats!!

MPI_Finalize();
}

```

A final note: your program is NOT parallel deterministic. That means that your Cellular Automata universe will differ at each step/tick in the computation depending on how many MPI Ranks you use. This is because the exchange of rows is using data computed in a prior tick and does not get the benefit of the update dependencies that happen within a “tick”.

Also, to link with the Pthread lib, make sure you add -lpthread to your compile command.

2 Experiments

You will conduct the following series of performance experiments.

2.1 Strong Scaling with Parallel I/O

a strong scaling experiment using “AMOS”, the CCI Blue Gene/Q system. Here, you will have a mix of MPI ranks and threads per compute node.

- Run a 16384x16384 cell universe using 256 MPI ranks, 0 Pthreads per Rank (4 BG/Q nodes) for 128 ticks with a threshold of 25%.
- Run a 16384x16384 cell universe using 64 MPI ranks, 4 Pthreads per Rank (4 BG/Q nodes) for 128 ticks with a threshold of 25%.
- Run a 16384x16384 cell universe using 16 MPI ranks, 16 Pthreads per Rank (4 BG/Q nodes) for 128 ticks with a threshold of 25%.
- Run a 16384x16384 cell universe using 8 MPI ranks, 32 Pthreads per Rank (4 BG/Q nodes) for 128 ticks with a threshold of 25%.
- Run a 16384x16384 cell universe using 4 MPI ranks, 64 Pthreads per Rank (4 BG/Q nodes) for 128 ticks with a threshold of 25%.

Note, the MPI rank also counts as a Pthread within a compute node.

Perform the above experiments using 16, 64 and 128 compute nodes with the same mix of MPI ranks and Pthreads per compute node. In total, you will have 15 experiment runs.

For the 128 compute node configuration, turn on the parallel I/O and output the cell universe and record the time spent performing I/O as well.

Now, plot **compute** execution time (Y-axis) as a function of the total number of MPI ranks * Pthreads per rank (X-axis) grouping each threads per MPI rank as a graph line. Do you observe any performance differences as the threads per MPI rank changes? Also, in your report, discuss the max speedup relative to the execution time for the 256 MPI rank on 4 compute node case you obtain as well as at what point is your parallel efficiency the greatest (i.e., speedup / MPI ranks for each case and which case yields the highest value).

Next, plot the **parallel I/O** execution time (Y-axis) as a function of the total number of MPI ranks used in the 128 compute node series tests. Discuss what write I/O performance differences you observe as the number of MPI ranks increases.

2.2 Parallel I/O and Heatmap of Final Universe State

For the parallel I/O and heatmap of final universe state conduct the following series of experiments.

- Run a 16384x16384 cell universe with a threshold of 0% for 128 ticks using the best performance 128 compute node configuration.
- Run a 16384x16384 cell universe with a threshold of 50% for 128 ticks using the best performance 128 compute node configuration.
- Run a 16384x16384 cell universe with a threshold of 75% for 128 ticks using the best performance 128 compute node configuration.

Including the previously run 25% threshold experiments, construct a *heatmap* of 1024x1024 size for each configuration (4, 1Kx1K heatmaps in total) where a single element of the heatmap is the sum of a 16x16 grid of the live/dead state within the cell universe. You are allowed to use any post process tools available or modify your C-code to construct the heatmap for you using the MPI gather routines and parallel processing techniques you've learned in class. Make sure you describe your approach for heatmap construction in your report.

Additionally, include each 1Kx1K heatmap graphic where higher numbers mean more cells alive within each 4x4 grid. What do you observe about the patterns in each heatmap as the randomization threshold is increased?

For details on how to create a heatmap using GNUPlot, see: <http://www.kleerekoper.co.uk/2014/05/how-to-create-heatmap-in-gnuplot.html>.

3 HAND-IN INSTRUCTIONS

Submit both your PDF report and C-code to the class *Submittity* server, `submittity.cs.rpi.edu`.