CGL Hints

A good page with references is available at <http://www.bitstorm.org/gameoflife/>

# Part 1 - Software Development

* You should use a single code base for all three partitioning strategies (single-processor, block, and checkerboard). Most people use a scheme where each processor keeps variables for the ranks for its 8 neighbors (N, S, E, W, NE, NW, SE, SW), and exchange ghost rows with each neighbor only if it exists. That way, the same code works for all partitions and the only difference between the three is just the calculation of the neighbors. Below is an example from my solution where I utilize the MPI\_PROC\_NULL parameter to pass to MPI for a null op, as well as printing out the neighbor calculations & an example assert call (which should be used often in your code to verify your assumptions):

|  |
| --- |
| void print\_header(void) { printf("[%3i:%6i] : ", rank, output\_count++); }  …  dimension = sqrt(size);  …  if (rank < dimension) //we are in first row  {  North\_rank = MPI\_PROC\_NULL;  NE\_rank = MPI\_PROC\_NULL;  NW\_rank = MPI\_PROC\_NULL;  }  …  if (!quiet)  {  print\_header(),printf("DEBUG: NW:(%d), N:(%d) NE:(%d)\n", NW\_rank, North\_rank, NE\_rank);  print\_header(),printf("DEBUG: W:(%d), Me:(%d) E:(%d)\n", Westy\_rank, rank, Easty\_rank);  print\_header(),printf("DEBUG: SW:(%d), S:(%d) SE:(%d)\n", SW\_rank, South\_rank, SE\_rank);  fflush(NULL);  }  …  assert( my\_row < num\_rows-1 && my\_column < num\_columns-1 && my\_row >= 1 && my\_column >= 1 );  … |

* Use two dynamically allocated arrays for your world: one that represents iteration *i* and one for *i+1*. Use i to update i+1, then **swap the pointers** at the end of the iteration.
* When implementing the Game of Life rules, **inline** the cell update code instead of calling a subroutine.
* The playing field dimensions should be allocated dynamically after reading the configuration file, not using static 2-D arrays. Use malloc() or calloc() instead.
* When allocating arrays dynamically, remember to use a single malloc() call to get a single block of memory. Thus, the 2D array will really be a contiguous 1D array, and you can index into the array using A[ y \* width + x ] or \***(**A + y\*width + x). (You can use a double pointer array if you want after allocating it so that you can use A[y][x] notation instead). If you call malloc() more than once for the same playing field, the blocks will not be contiguous and will cause problems when sending using derived datatypes.
* For the checkerboard implementation, you (may want to or may **need** to, read the Midterm part 2 assignment carefully) use [two-stage forwarding](http://www-unix.mcs.anl.gov/mpi/tutorial/perf/mpiperf/sld081.htm) instead of sending four small diagonals.
* When debugging your program, consider making small game fields manually and printing out the playing field on each processor.

# Input File Format and MPI-IO

* If you want to test your data file ingest code easily, consider using a PGM file containing ASCII printable characters so you can see that the data got to the right processor. We’ve provided [one here (ascii-24x24.pgm)](http://www.google.com/url?q=https%3A%2F%2Fwww.dropbox.com%2Fs%2Fdayh3xdet2d59o0%2Fascii-24x24.pgm&sa=D&sntz=1&usg=AFQjCNHA9uBRXSTrkFS7YT6Cn99HRvC41w). This file is in PGM format and contains only ASCII text for a 3x3 checkerboard partitioning where each processor has an 8x8 grid. If the file is loaded right, you should see numbers bounded by letters on each processor, and your ghost rows SHOULD BE EMPTY! On a block partitioning with np=3, you should see three blocks (A,B,C) next to each other.

Of course, this file is really for testing only. If you apply the rules and run the game, it's just a fully-populated grid. It's convenient for testing because the "binary" data is coincidentally printable ASCII, so you can see what's happening using printf() easily.

* You may use any MPI-IO technique you prefer to save the field to a disk file, but it must work for all parallelizations. You should be able to do this with one derived data type, one derived file type from MPI\_Type\_create\_darray, and a single call to MPI\_File\_write. (You can write the pgm header on one process, and then use that as the offset for the call that writes the data.) For parallel output to work, you must be writing the data to a parallel file system. Check the documentation for the system you’re using to determine which file system is parallel.
* [This tutorial](http://people.cs.uchicago.edu/~asiegel/courses/cspp51085/lesson10/mpi-io.ppt) might be helpful for understanding MPI-IO distributed arrays. Associated code examples are [available here](http://www.mcs.anl.gov/research/projects/mpi/usingmpi2/examples/main.htm) (see the Parallel I/O section).

# Viewing PGM image files

The [Netpbm command-line tools](http://netpbm.sourceforge.net/) can be used to convert PGM files to a more easily viewable image format.

On Debian, try installing the netpbm package.

On OSX, try installing netpbm through fink, ports, homebrew, etc. For homebrew, it can be installed by running ‘brew install netpbm --devel’

Once installed, use the ppmto\* commands to convert .pgm files into something else.

For example, converting to a bitmap (.bmp) on the command-line:

|  |
| --- |
| ppmtobmp conway-900x900.pgm > conway-900x900.bmp |

# Creating an animation from image files

Example ImageMagick command to create animated .gif from a series of .pgm files:

|  |
| --- |
| convert -delay 20 -loop 0 file1.pgm file2.pgm file3.pgm animate.gif |