Game of Life

Aaron Morgenegg

10/26/18

OpenMPI Game of Life

This program runs Conway's Game of Life using multiple processes. The assignment description called for 1024x1024 pixels for the game size, but this took more processing power than I had available. Instead I ran the game with 256x256 resolution. At this resolution, it took 1211.80 seconds to run 100 days of the game, using randomly generated starting cells. To demonstrate the program, I also ran the program with a simple glider in the center, which is shown in the attached images.

```
// mpic++ main.cpp
// mpiexec -np 2 a.out

#include <iostream>
#include <fstream>
#include <time.h>
#include <stdlib.h>
#include <mpi.h>
#include <unistd.h>
#include <algorithm>
#include <math.h>
#include <string>
#include <vector>

const int RESOLUTION = 256; // size of array, 2d so it's squared
const int PROPORTION = 20; // % of squares initially alive
const int DAYS = 100;
// If true, set each cell randomly according to proportion
// If false, start with an empty world, with a glider in the center.
const bool RANDOM_MODE = true;
const std::string FILE_ROOT = "output/day_"; // Root filename for image output
const bool SEPARATE_FILE_BY_DAY = true; // output to same file, or seperate files

std::vector<int> getEmptyArray(int size){
// 2d array represented by 1d array
std::vector<int> plot(RESOLUTION*size,0);
        return plot;
}

std::vector<int> initArray(int size, int proportion){
// Return an array of size x size, with a proportion% chance of being alive
std::vector<int> plot = getEmptyArray(size);
for(int i = 0; i < size*size; i++){
if(rand()% 100 < proportion){
plot[i] = 1;
}
}
return plot;
```

```
}

std::vector <int> initGlider(int size){
// Return an empty array, with a glider in the center.
std::vector<int> plot = getEmptyArray(size);
int midpoint = size/2;
plot[midpoint*RESOLUTION + midpoint-1] = 1;
plot[midpoint*RESOLUTION + midpoint] = 1;
plot[midpoint*RESOLUTION + midpoint+1] = 1;
plot[(midpoint-1)*RESOLUTION + midpoint+1] = 1;
plot[(midpoint-2)*RESOLUTION + midpoint] = 1;
return plot;
}

void sendPlot(std::vector<int> plot, int world_size){
for(int i = 1; i < world_size; i++){
MPI_Send(&plot[0],RESOLUTION*RESOLUTION,MPI_INT,i,0,MPI_COMM_WORLD);
}
}

std::vector<int> gatherArray(int world_rank, int world_size, std::vector<int> plot
int offset = RESOLUTION/world_size;
if(world_rank==0){
std::vector<int> recv_data = getEmptyArray(RESOLUTION);
MPI_Gather(&sub_plot[0],RESOLUTION*offset,MPI_INT,&recv_data.front(),RESOLUTION*of
sendPlot(recv_data, world_size);
return recv_data;
} else{
MPI_Gather(&sub_plot[0],RESOLUTION*offset,MPI_INT,NULL,RESOLUTION*offset,MPI_INT,0
MPI_Recv(&plot[0],RESOLUTION*RESOLUTION,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNO
}
return plot;
}

int getRelativeCellValue(int index, std::vector<int> plot, int i_diff, int j_diff)
try{
return plot.at(index+(i_diff*RESOLUTION)+j_diff);
}
```

```
catch (...){
return 0;
}
}

int countNeighbours(int index, std::vector<int> plot){
int count = 0;
count += getRelativeCellValue(index, plot, -1, -1);
count += getRelativeCellValue(index, plot, -1, 0);
count += getRelativeCellValue(index, plot, -1, 1);
count += getRelativeCellValue(index, plot, 0, -1);
count += getRelativeCellValue(index, plot, 0, 1);
count += getRelativeCellValue(index, plot, 1, -1);
count += getRelativeCellValue(index, plot, 1, 0);
count += getRelativeCellValue(index, plot, 1, 1);
return count;
}

int updateCell(int index, std::vector<int> plot){
int value = plot[index];
int neighbours = countNeighbours(index, plot);
if(neighbours == 3){
return 1;
}
else if(neighbours == 2){
return value;
}
else if(neighbours < 2){
return 0;
}
else if(neighbours > 3){
return 0;
}
}

std::vector<int> updateSubPlot(std::vector<int> plot, int start_index, int chunk_s
std::vector<int> sub_plot = getEmptyArray(chunk_size);
for(int i = 0; i < chunk_size; i++){
```

```
sub_plot[i] = updateCell(i+start_index, plot);
}
return sub_plot;
}

std::vector<int> setupPlot(int world_rank, int world_size){
if(world_rank == 0){
std::vector<int> plot = initArray(RESOLUTION, PROPORTION);
if(!RANDOM_MODE) plot = initGlider(RESOLUTION);
sendPlot(plot, world_size);
return plot;
} else{
std::vector<int> plot = getEmptyArray(RESOLUTION);
MPI_Recv(&plot[0],RESOLUTION*RESOLUTION,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNO
return plot;
}
}

std::string getFilename(int day){
if(SEPARATE_FILE_BY_DAY){
return FILE_ROOT + std::to_string(day) + ".ppm";
} else {
return FILE_ROOT + ".ppm";
}
}

void saveWorld(std::vector<int> world, int day){
std::string filename = getFilename(day);
std::ofstream my_file(filename, std::ofstream::app);
if(SEPARATE_FILE_BY_DAY) std::ofstream my_file(filename);
        my_file << "P1" << std::endl;
        my_file << RESOLUTION << " " << RESOLUTION << std::endl;
for(int i = 0; i < RESOLUTION; i++){
                for(int j = 0; j < RESOLUTION; j++){
                        my_file << world[i*RESOLUTION+j] << " ";
                }
                my_file << "\n";
        }
```

```
        my_file.close();
}

void gameOfLife(int world_rank, int world_size){
std::vector<int> plot = setupPlot(world_rank, world_size);
for(int i = 0; i < DAYS; i++){
if(world_rank==0){
std::cout << "Day " << i << std::endl;
        saveWorld(plot, i);
}
// divide work between processes
int chunk_size = RESOLUTION*RESOLUTION/world_size;
int start_index = world_rank * chunk_size;
// update your portion of the array
std::vector<int> sub_plot = updateSubPlot(plot, start_index, chunk_size);
// merge portions back into array for next day
plot = gatherArray(world_rank, world_size, plot, sub_plot);
}
}

int main(int argc, char** argv) {
MPI_Init(&argc, &argv);
srand(time(NULL));
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
// MPI_Send(&sendData,1,MPI_INT,dest,0,MPI_COMM_WORLD);
        // MPI_Recv(&recvData,1,MPI_INT,dest,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

gameOfLife(world_rank, world_size);
MPI_Finalize();
return 0;
}
```