Conway’s Game of life Assignment 1 Report

Cocurrent System 159.355

Jonathon Kearney

09005935

This report outlines the experiments, observations and strategies undertaken while constructing a parallel implementation of Conway's Game of Life. It will cover the strategies successfully and unsuccessfully used and the observations I made while going through that process. Both MPI and OpenCL proved to be very different in their implementation, which made the process of coding them both interesting and at times frustrating.

Index

1. Parallelisation Strategy

1.1. MPI

My strategy with the MPI implementation was based around the idea of limiting the amount of MPI transfers between processes and making sure no process receives any superfluous data.

I went about this by splitting the array into four 25 column sections. I then copied each side column of each section and amend it to the opposite side. This would result in four 27 column sections that I amended into one 1D array. My strategy from there was to scatter that array to each rank and have each rank calculate the middle 25 columns of their sections. Following this, I used gather to amend them back together. From there I unflattened the array, copied the middle 25 columns into update, made current=update, and repeated this process indefinitely.

1.2. OpenCL

I implemented a more basic strategy for the OpenCl implementation to avoid any issues with concurrency. I created three separate kernels: one that adds the outside rows; one that adds the outside columns; and one that calculate the inside cells. Each subsequent kernel requires the previous one to complete to produce the right result, therefore each kernel is enqueued separately.

In the MPI implementation, the infinite loop that runs the iterations is called in the GameofLife class. I called this implementation in the runIterations method, which contains two arrays. The first has the data read out of it, and the second has the new cells written into it. After each iteration the resulting data is read back into update, printed, and the arrays then switch roles for the subsequent iteration.

2. Early experiments and issues

2.1 MPI

My first speedbump in this assignment was a trivial one. I tried initializing MPI in the game of life class but had problems when it came to the argument required to do so. I soon realized that I should initialize everything in a main method. In retrospect this seemed obvious, which so often is the case when looking back.

Initially I tried to create a test class so I could have a controlled environment to test the MPI methods. I started by creating one dimensional arrays and sending them via the MPI methods. I came across some difficulty in debugging the MPI code so to debug i mainly used print statements scattered around to see which processes got how far. After some success, I transferred that code into two methods in the LifeEnv class: “runOneIteration” and “sendArrayMaker”. I left the Test class in the package to show my progression.

Once I got my code to a point where it was outputting a new display, I added some “1” cells that would produce a simple and predictable outcome after one iteration. In this instance and in the instance of OpenCL later, I used 3 vertical points next to each other. Eg [10][10], [10][11], [10][12]. I knew these would produce 3 horizontal points next to each other if the calculation was successful, so i tampered with my code until i got that result.

2.2. OpenCL

After obtaining the correct drivers and downgrading to a JOCL library version that worked, I went to work on creating a controlled environment Test class. I soon realised that I needed to find a way for error messages to appear because by default they were not on. I did this by setting exceptions enabled as true and by testing the return values of some of the methods.

The difficulty I found in the OpenCl implementation was actually getting a return result when reading the buffer after the kernels were called. The issue in the end was one of properly setting the dimensions of the local and global work size arrays and understanding that each global ID dimension requires its own value to set its parameters.

3.Observations/Comparisons

I created a simple test on each implementation in order to measure the time it takes to complete an iteration. This test basically involved creating two variables that stored the system time at different points, then subtracting the first from the last. Each iteration involves gathering the current grid, calculating it, and sending it back. Below is the results in milliseconds of ten such tests. These tests were run on an Generation 3 i5 Intel processor and HD 4000 integrated graphics, and as such these results will likely be different compared to other more capable machines.

											AVG
MPI	153	159	134	135	224	190	173	165	183	161	**167.7**
OpenCL	142	143	142	144	143	143	143	141	143	141	**142.5**

(Note: There was a roughly 600ms setup time for the context, commandQueue, Program Loading etc in OpenCl. This is a one off event though as it happens outside of the loop that updates the cells)

Firstly I saw that the OpenCl implementation runs faster, but as mentioned above it has a setup cost of around 600ms. That means that after roughly 24 iterations it overtakes MPI in efficiency. If I had a dedicated graphics card I would assume my OpenCl results to be even better.

Secondly I saw that the OpenCl implementation performs more consistently than the MPI version. This could possibly be due to my CPU having to simultaneously run other tasks in the background, whereas the GPU can be dedicated to the task.

My conclusion after these and other observations is that the GPU and OpenCl are more suited to this problem than MPI. When a problem can be broken down into small non stressful computations, OpenCL should be used. When the problem involves fewer parts but larger computations then MPI may be more suitable. I predict that as the dimensions of the Game of Life grid get larger, the gap between the MPI and OpenCl speed results will get exponentially larger too.