CS4204-02-Ant Colony Optimisation

120006720

April 2016

1 Travelling Salesperson

Assumptions made for the travelling salesperson graph were as follows:

1. Connected graph - having edges between all nodes
2. Edge cost between two any nodes is always greater than 0
3. Directed edges with two connections between each node

2 Design

Much of the code has been designed in a self-descriptive way, making use of variable naming and method naming to describe what elements of the code are doing. The host code is separated into three files `ants.c`, `BorrowedFunc.h` and `Structure.h`. `ant.c` is where the `main` is located and deals with the ant colony data structure and the iterations of ant execution. `Structure.h` deals with OpenCL data structures and parsing of kernel files. Finally, `BorrowedFunc.h` contains code that was not written by the author of this report and was functions created by others for this use.

The main kernel is designed to simulate each ant and their decision making process, taking in the cost and pheromone array and generating a single solution from them. To avoid passing of dynamically-sized blank memory for the data structures required for each individual kernel a string replacement was performed before compile time replacing placeholders such as `NODESIZE` with the value that will be passed in by the user at run-time. This is as data transfer to the device can be very slow. It made use of `__private` memory to avoid performing repetitive searching tasks (such using a boolean array to remember which nodes had been visited) and to avoid having to read and write from `__global` memory (instead using a private array to store the solution, only writing to the global solution array at the end).

2.1 Initial Values

Based off of command line arguments (number of nodes, minimum and maximum edge weights) a random connected graph representation is generated as

a single dimensional array, which itself is representing a two-dimensional array (as two connections for each edge means that travelling from node A to node B may not be the same as B to A). The randomness of the graph is ensured by using the PCG algorithm (see subsection *Randomness*).

By default the initial value for all pheromones is calculated as the inverse of the length random walk multiplied by the number of nodes present, influenced by the research performed by Sonigoswami et al.[4] though they use a heuristic-based walk instead. However, the user may specify a initial pheromone value, though this is not advised since if a value is applied the system which is much higher than the pheromone value that any ant can put down, then the change in pheromone values will not be enough for an ant to be significantly attracted by it until all the paths have evaporated to a certain degree.

2.2 Pheremonal Updating

The update value applied to pheromones can be performed in one of two ways in this system allowing for both parallel and sequential editing of this key data. This can be set by the user using the `-pPh` flag to use the GPUs for updating of the pheromone array. The function decided to represent g (the amount added to each edge) was set as:

$$g(S_k) = \frac{1}{\text{length}(S_k)} \tag{1}$$

Where S_k is the solution from the k^{th} ant.

The parallel update thus had a much more inefficient calculation, as since OpenCL does not allow for lock free editing of data values, instead each work item must have to work on a separate section of the pheromone array. This is fine for evaporation, but for finding which solutions contain the edge which it is currently updating the pheromone for is more complex than sequentially going through each solution and adding the g on for every edge travelled.

2.3 Randomness

In order to make the ants have truly random probable path following a combination of functions were used. First random seeds were generated on the host using the PCG algorithm[3] seeded following protocol (using the current time and the number of rounds required). The number of seeds generated is equal to the number of nodes, as this is how many decisions any one ant will need to make. Finally, during kernal execution each kernel will add its global ID to the seed and perform an xorshift on this new unique seed, and generates a decimal value from 0 to 1 (inclusive). This gives widely varying probabilities as seen in figure 1 which plots the frequency of occurrence of probabilities between set values, and also displays the normal for these taking a sample of three million values. The kernel density of the results can also be found in the appendix under figure 5.

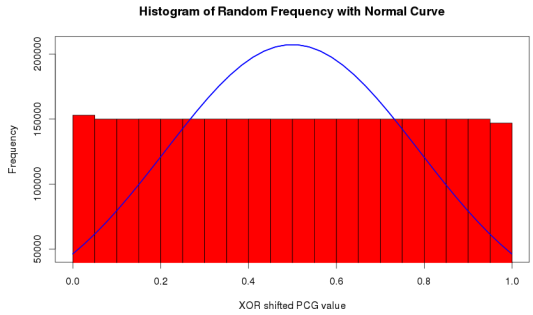

Figure 1: Frequency and normal of the resultant probabilities for ant movement

3 Results

An interesting note about both datasets is not only how non-optimal the solutions are, but also in the disregarding of strong results. With the implemented pheromone update only 19 of 70 tests from the `resultsNodes.csv` dataset settling the best solution by the final iteration, which an ant had walked along at some point during all the iterations. Only 37 of the 188 from `compiledParaVNorm.csv` were

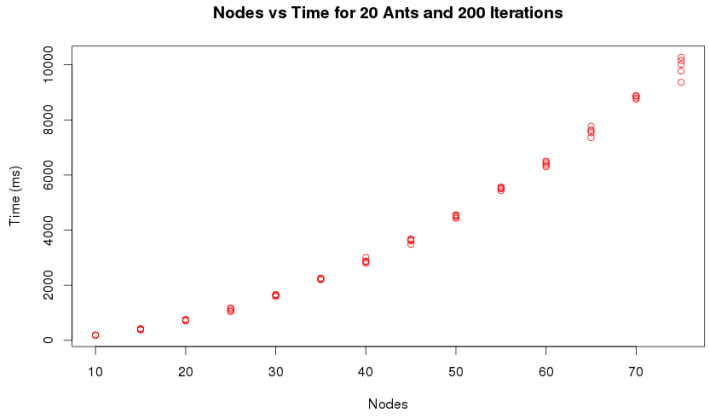

Figure 2: Time taken with sequential pheromone update, from `resultsNodes` dataset

With larger numbers of ants and iterations the results show prominently the escalation of time taken to simulate the ants in these scenarios. Figures 3 and 4 are taken from the `compiledParaVNorm` dataset and show the difference from parallel and sequential pheromone update. Although sequential is much faster for small values of nodes (figure 3) this starts to change at around 40

nodes. This indicates at around this level of simulation that the complexity of the parallel update is negligible compared to the speedup gained by breaking the task into data parallel sections.

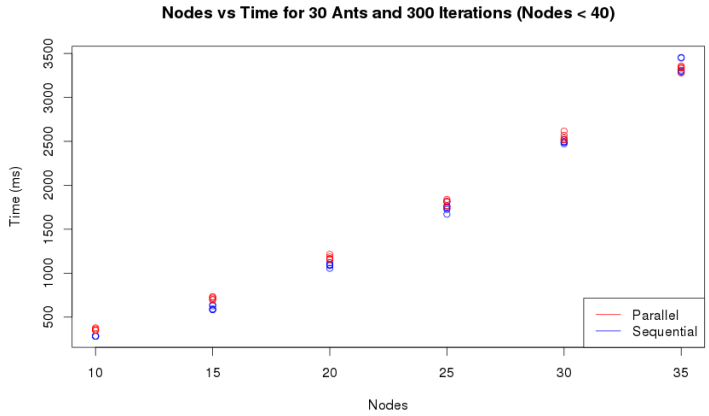

Figure 3: Time taken with varying pheromone updating and small number of nodes

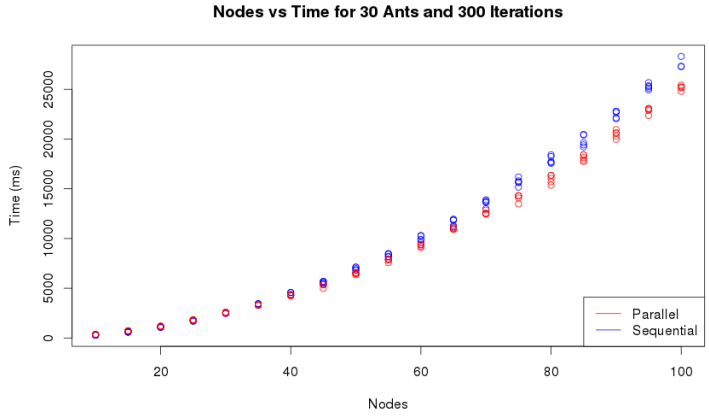

Figure 4: Time taken with varying pheromone updating across larger node samples

4 Conclusion

This project aimed to create a working parallel version of an ant system, which this project completed. It also compared varying levels of parallelism, looking

at breaking down the task at varying levels and comparing the effectiveness of each. This project shows how OpenCL can be used in a scientific setting to perform complex arithmetic which can be executed simultaneously, where if the same code is ran on the CPU it can take much longer. The complexity of OpenCL structure and syntax is overshadowed by its potential uses across many scientific fields, as demonstrated due to the success of this project in a short time frame where the designer started with no experience.

There are a number of possible future comparative tests to be conducted on this implementation. A simple one would be to reduce the number of arguments passed into the kernel. Reading to and from the GPU can take large amounts of time and any reduction in this could lead to significant improvements, and since unchanging parameters are passed in each time there is a lot of unneeded overhead. This could be removed by instead before compilation of the kernel program a string replace could be enacted to input all of these values. Furthermore, there could be greater experimentation into the use of blocking and non-blocking writes to the buffer to see the difference in time.

A larger extension to this project would be creating what is commonly referred to as an *ant colony system* (ACS)[1][4][2], as this implementation is instead described as a *ant system*(AS)[1]. An ACS instead employs a elite ant strategy whereby for any one iteration, only the ant with the lowest cost solution will deposit any pheromones along the paths (though this could be extended to n best ant solutions). ACS also uses a different pheromone trail evaporation, where not only will the trail evaporate over time, but a local update is done any time any ant takes a path; this means that immediately, within the same iteration, ants are less likely to take this path and instead will be pushed to greater exploration. This could be employed in OpenCL using `__local` memory across an entire work group, though could lead to large slow down if using blocking memory, or widely varying behaviour if non-blocking.

5 Appendix

5.1 Running Instructions

If the code is run with no arguments all the possible argument values and meaning are outputted to explain how to run the program.

It is currently set up to run on linux lab machines as in `Structure.h` there is a variable named `platformToUse` which is set to 1, which makes it take the second platform of the machine, namely the GPU.

5.2 Restrictions

1. **Maximum number of nodes must be less than 15 digits in length** - this is a hard-coded limit due to string replacement in the kernel compilation. This could be fixed by passing the value into the kernel arguments, though this would less efficient.

5.3 Graphs

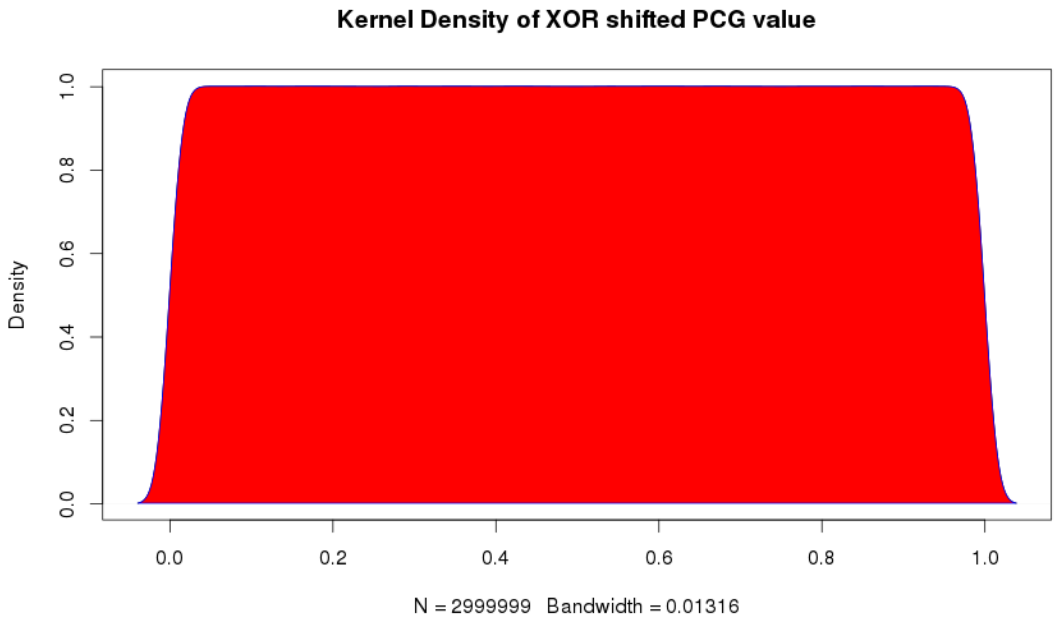

Figure 5: Kernel density of the resultant probabilities for ant movement

References

[1] Rohit Chaturvedi and Haider Banka. Modified ant colony optimization algorithm for travelling salesman problem. *nternational Journal of Computer Applications*, 97(10), July 2014.

[2] Marco Dorigo and Luca Maria Gambardella. Ant colonies for the traveling salesman problem, 1997.

[3] Melissa E. O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation, 2015.

[4] Gaurav Singh, Rashi Mehta, and Sapna Katiya Sonigoswami. Implementation of travelling salesman problem using ant colony optimization. *International Journal of Engineering Research and Applications*, 4(6):63–67, June 2014.