University of the Witwatersrand

COMS4040: High Performance Computing and Scientific Data Management Group Assignment

Parallel Tree Search In Playing The Game 2048

May 21, 2017

*B*y Mizrahi, M. (708810)
*B*y Chalom, J. (711985)

0.1 Abstract

0.2 Introduction

Games provide an interesting set of problems for artificial intelligence since they provide a level of complexity whilst having clearly defined rules which lends these problems to implementation and simulation. Many games have solution spaces which are intractably large and therefore algorithms and methods employed to solve or approximate any optimal solution to these spaces have to employ smart techniques to deal with the computational constraints of such problems. High performance computing then lends itself to expanding the capabilities of these methods to deal with much larger solution spaces in much shorter amounts of time than a serial approach's performance could achieve. [1]

0.2.1 Problem statement

We wish to enhance a tree search algorithm by using high performance technologies so that we can improve its performance. The tree search algorithm will be applied to a game called 2048.

0.3 Background

0.3.1 The game 2048

This is a single-player non-deterministic sliding block puzzle game. It's a fairly simple game with the aim of producing one tile on the board which is equal to 2048. /citerules

0.3.2 The Rules of the game:

- The game is played on a square grid, i.e. 4x4 or 8x8 grid.
- Every turn the player will choose a direction for the blocks to move, either up, down, left or right.
- Tiles will slide as far as possible until stopped by the edge of the grid or another block.
- If 2 tiles have the same number while colliding then they will combine into a single block and the new value will be the sum of their values.
- After each action, a new tile will appear on the grid with a value of 2.
- The game is won when a tile with value of 2048 appears on the board.

0.3.3 Game Trees

The textbook, Artificial Intelligence: A Modern Approach [1] describes game trees as being directed graphs where each node indicates a game state and each edge indicates an action. A list of edges or connected nodes is a path. A game tree is called complete if it contains all possible actions from each possible state. This means that the size of a game tree can be massive and even exceed the available memory resources of the computing system being used. Each divergent path or sub-tree is mutually exclusive from other sub-trees which branch off either before or at the level of the sub-tree this means that these data structures are highly parallelisable. The main issues with trees is their branching factor and how to assign resources in a smart way to handle the exponentially growing number of nodes.

0.4 Methodology

0.4.1 General Assumptions

All drawn diagrams were drawn using http://draw.io/ and charts were made with Microsoft Excel 2013.
All results were calculated as an average of running the same experiment six times and calculating

the average result.
The system used for empirical analysis, was the wits cluster which is called Hydra. It consists of around 100 nodes where each node has a quad core CPU where three of the cores are exposed to the cluster. Each node also contains two GTX 750Ti GPUS with 2048MB on-board global memory. For our experiments we used 2 cores per node for the serial and the MPI implementations. Three implementations of tree searching was used. These being a serial C++ approach, a MPI approach and a Cuda GPU-based approach. All three approaches uses the C++11 standard. In the Cuda approach we assumed that the sub-trees (computed in parallel) are all completed where dead-nodes are just assigned as 'empty' type nodes and then ignored.

0.4.2 Testing Method

We built a game supervisor which keeps track of and maintains the game states and rules. This allowed us to make the different tree and tree search implementations consistent in their approach to the problem domain. We tested the algorithms and their performance based on the number of nodes in the tree they were able to implement versus the time it took to generate those respective trees for each algorithm. We ran each test several times and took the average result for each configuration. We also tested the algorithms on three board sizes, namely 4x4, 6x6, and 8x8.

0.4.3 Implementation

All the technologies use some form of the C/C++ programming language and specification. All implementations were designed to use a helper class which contains code which is the same across the three implementations. The three implementations also share a class structure for the tree and the nodes of the tree respectively. We made use of a node and pointer tree implementation. The Cuda implementation makes use of many pre-compiler directives to change the shared function calls into Cuda specific function calls – this was done to try and reuse code as much as possible. Certain functions had to be duplicated and parts modified to fit in the correct specification and paradigm depending on the implementation used. For both MPI and Cuda our strategy was to initially generate a tree using a serial method which would go to a depth where the number of cut-off states (the states which are not win states or dead-ends) are equal to the scale of parallelism we want – the number of nodes in MPI and the number of threads in Cuda. This meant that we could then treat these cut-off states as their own local roots of their respective sub-trees which we would then generate in parallel on each respective computing device. This strategy worked very well because the number of parallel tree constructions was far smaller than the final number of nodes we were able to generate. We believed that the overhead of parallelising the tree building in a dynamic load balancing manner [**?**] would be far greater than the cost of a very small initial serial step. This meant that no parallel data communication was needed after each process got their initial node from which to build their respective sub-tree.

Serial Approach

- We used a user controlled stack in order to build the tree.
- From the root node, all possible children are pushed to the stack.
- Then for each node at the top of the stack:
  - Each child is checked to see if it's a dead state or if it is a solution state
  - A dead state is one where there are no possible moves left.
  - If not dead state or solution state then it is pushed to the top of the stack
  - If a solution is found then store it for later processing.
- However, since the game tree is so big for 2048, the tree is limited by the number of nodes.

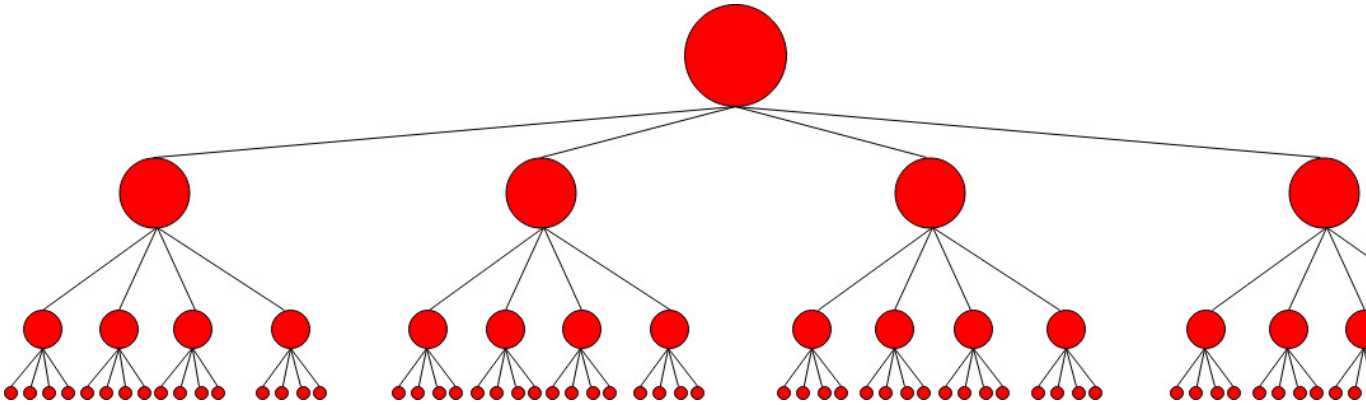

Figure 1: Tree structure of 2048 in serial approach

MPI Approach

- The MPI approach is very similar to the serial approach.
- The root process creates a small sub-tree until the number of leaves is equal to number of processes used.
- Each process is sent its initial state
- Each process then creates a serial search from the initial state.
- Then each process sends back its local optimal solution depth to the root process.
- The root process compares them and chooses the minimum.

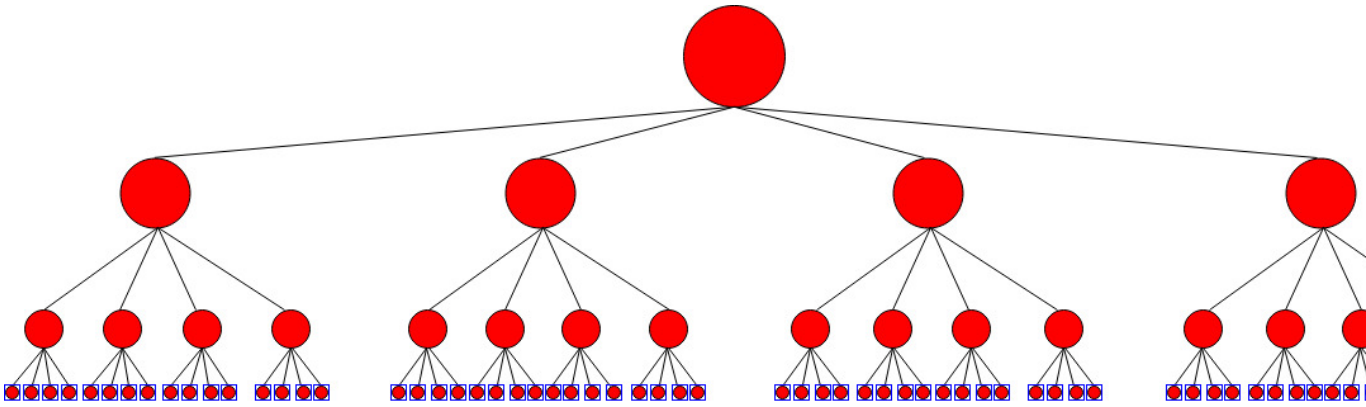

Figure 2: Tree structure of 2048 in MPI approach. The leaf nodes are distributed to each process where a serial tree search is run using the leaf nodes as the new root nodes.

The leaf nodes are distributed to each process where a serial tree search is run using the leaf nodes as the new root nodes.

Cuda Approach

- The CUDA approach starts in a similar manner to the MPI approach.
- The host generates a small sub-tree and stores each initial state to be used in the first column of a matrix.
- Each row in this matrix is a sub-tree to be explored by a thread.

- For each entry in its row, each thread creates the nodes children and places them in the row of the matrix.
- For this approach to work, we must assume the tree is complete and symmetric.

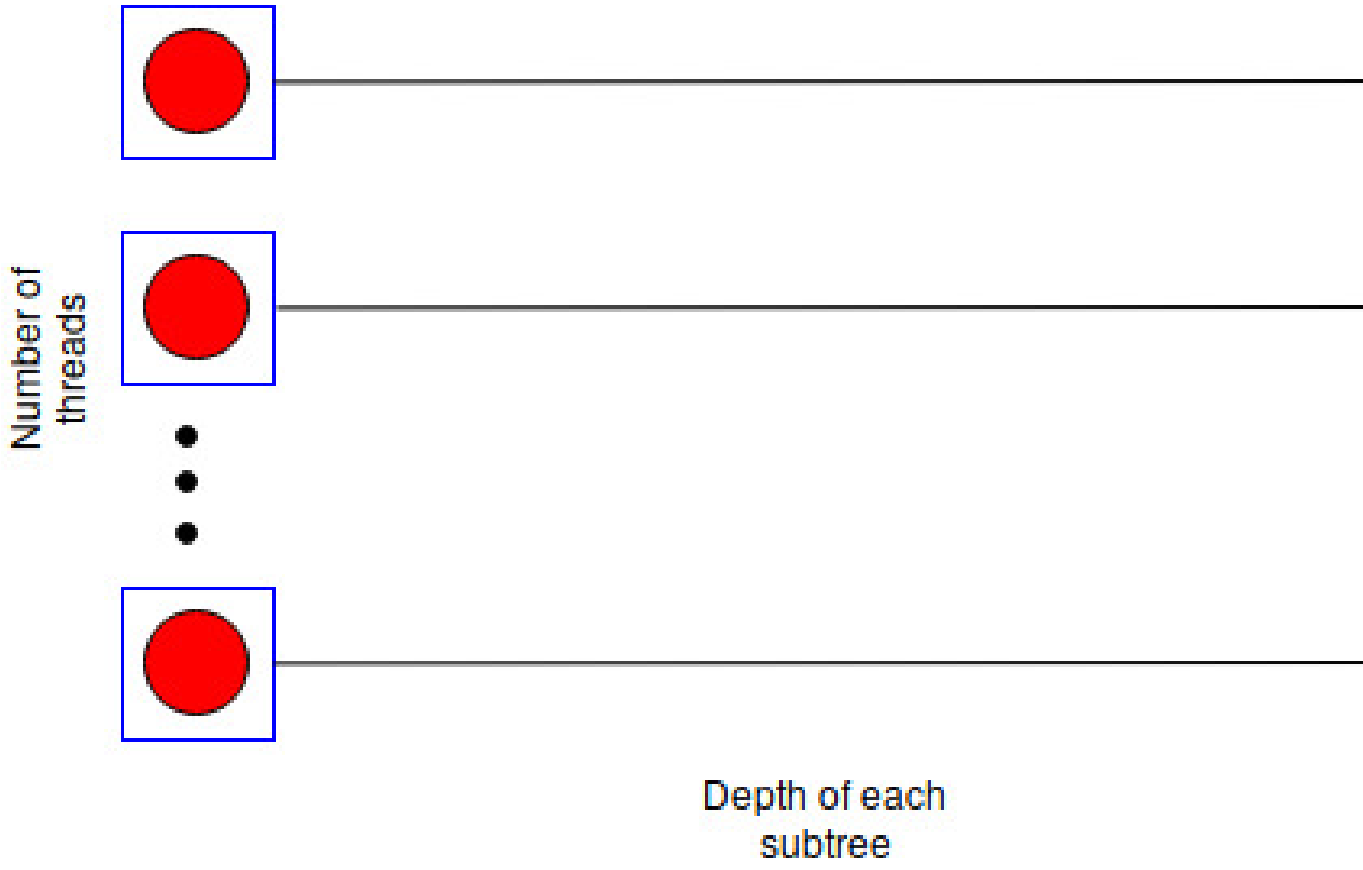

Figure 3: Tree structure of 2048 in Cuda approach. The leaf nodes are stored in an array where the first column has the initial states for each thread and each row represents a subtree to be explored.

A down side with the Cuda implementation is that we were unable to perform a warm-up step – beyond initialising the cuRand library - as that would mean building the entire tree twice.

0.5 Experimental setup

Since the game 2048 is a single player game we were able to independently run each AI and then analyse the returned results. Each AI kept track of certain statistics such as number of nodes and the time taken to reach a tree of each node size. We also built many command line parameters into the AI programs so that we could change the parameters and dynamics of the domain and AI at will. We used these parameters to test a variety of node sizes from 10000 nodes up to 5000000 nodes and to change the board size as needed. For the MPI implementation we also changed the number of processor nodes from the cluster that we used so that we could analyse the results to determine how the different node configurations effected performance.

We would start each AI with an initial state which would be generated using a pseudo-random number generator and scheme. We could choose to seed the PRNG with a static value to force the same initial state across our implementations.

We used the PBS scripts to allow us to repeat each experiment (with a precise configuration) many times to get a more accurate result. Each AI was capable (via the helper functions we made)

to save their results as csv files which we were then able to open in Excel to process and generate graphs for each implementation's performance.

0.6 Results and discussions

0.6.1 Charts

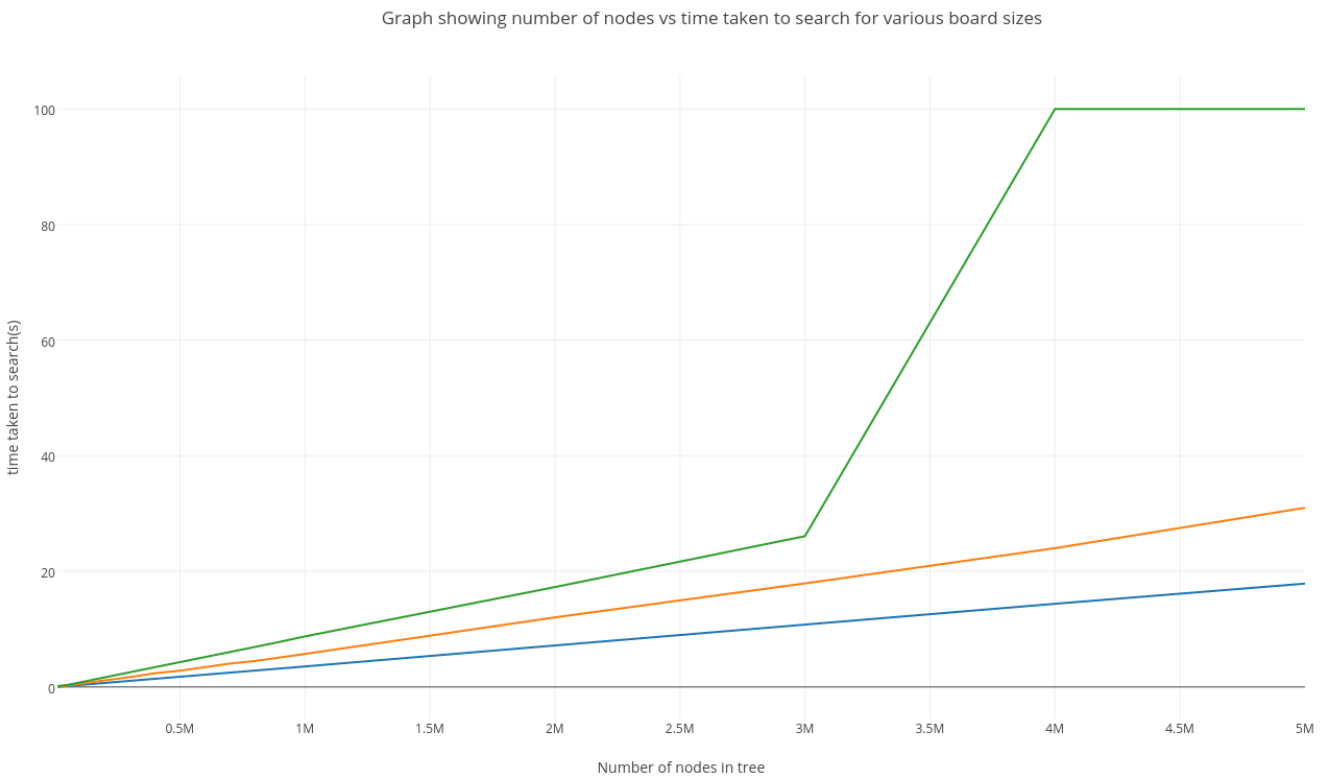

Figure 4: Serial execution time

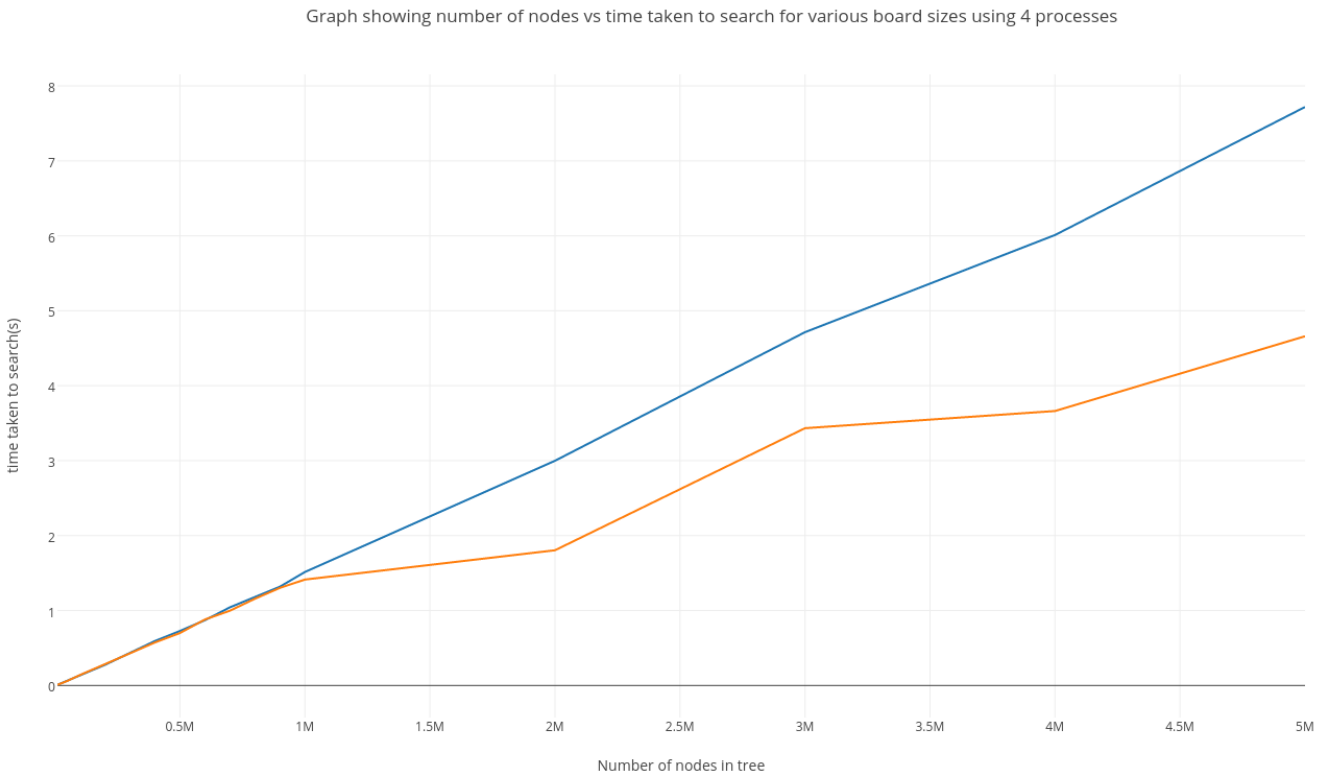

Figure 5: MPI execution time where number of nodes is divided by 4 processes evenly

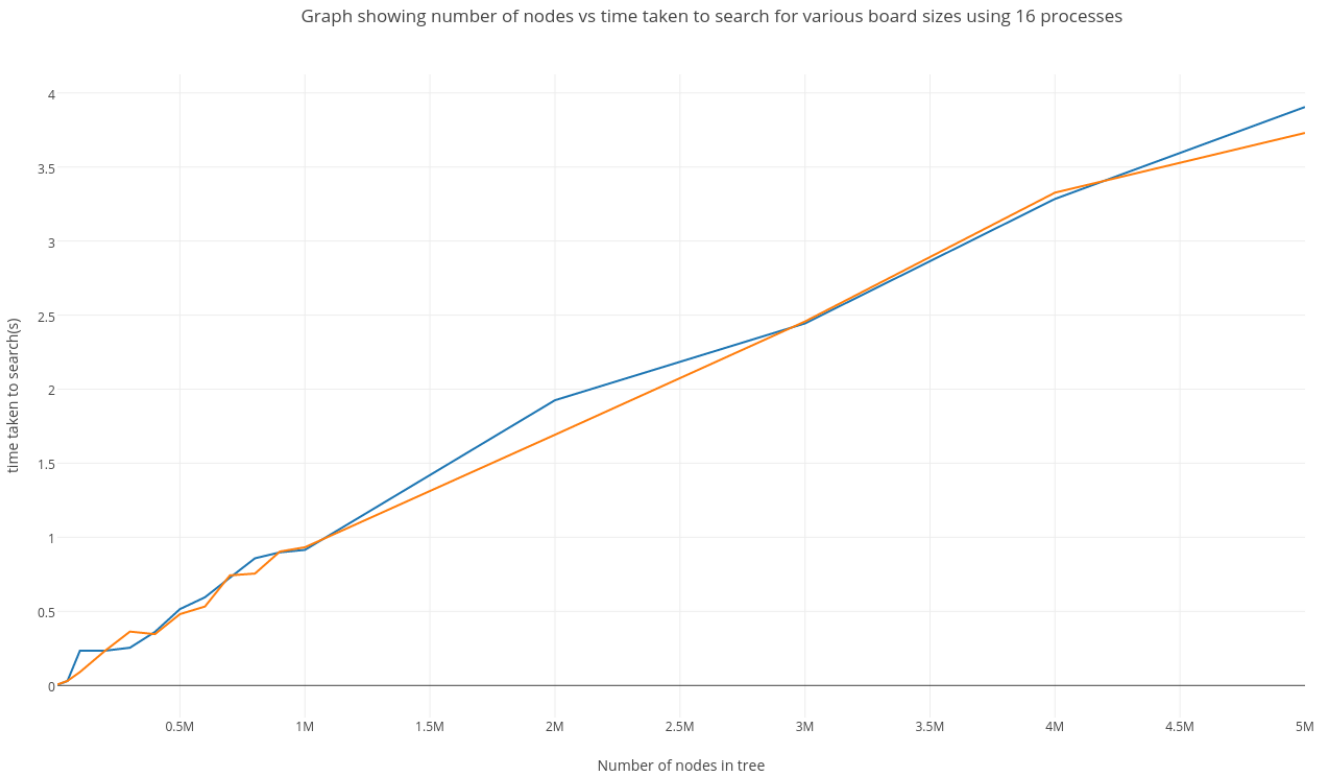

Figure 6: MPI execution time where number of nodes is divided by 16 processes evenly

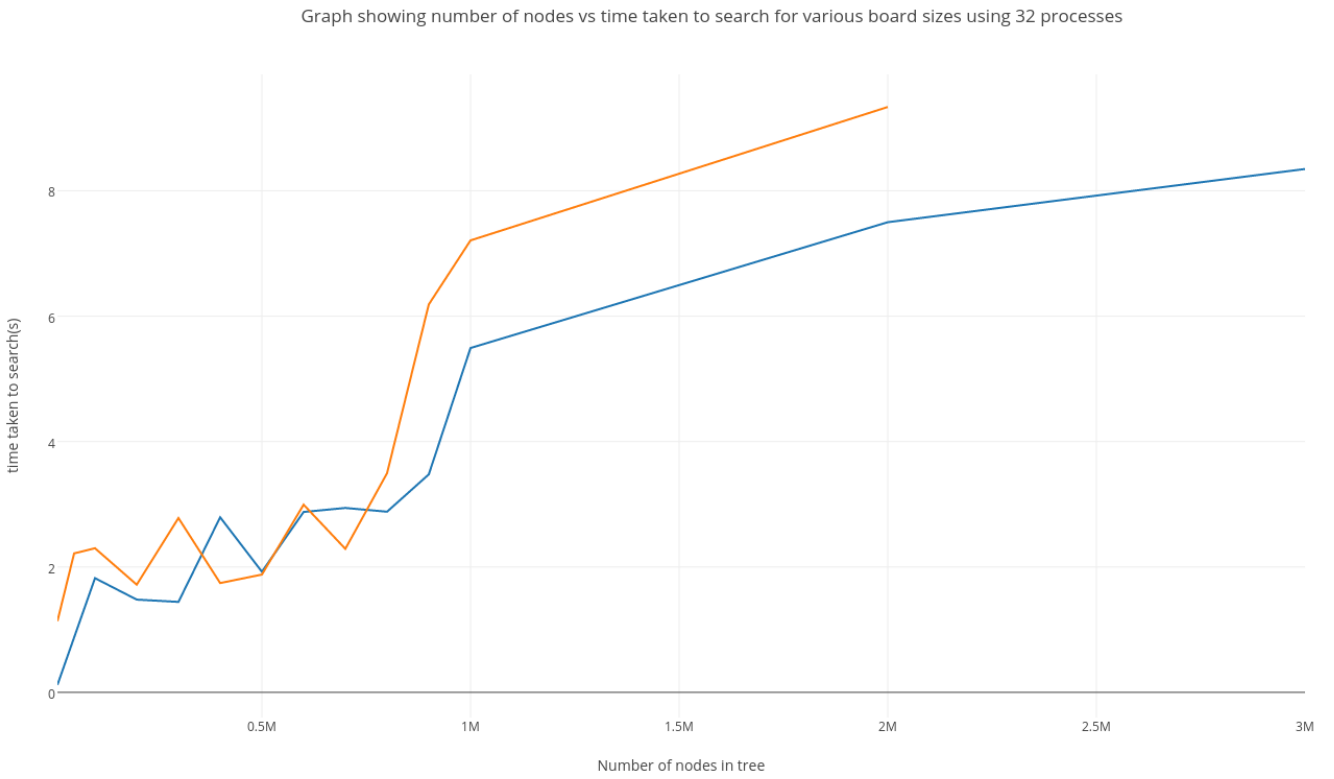

Figure 7: MPI execution time where number of nodes is divided by 32 processes evenly

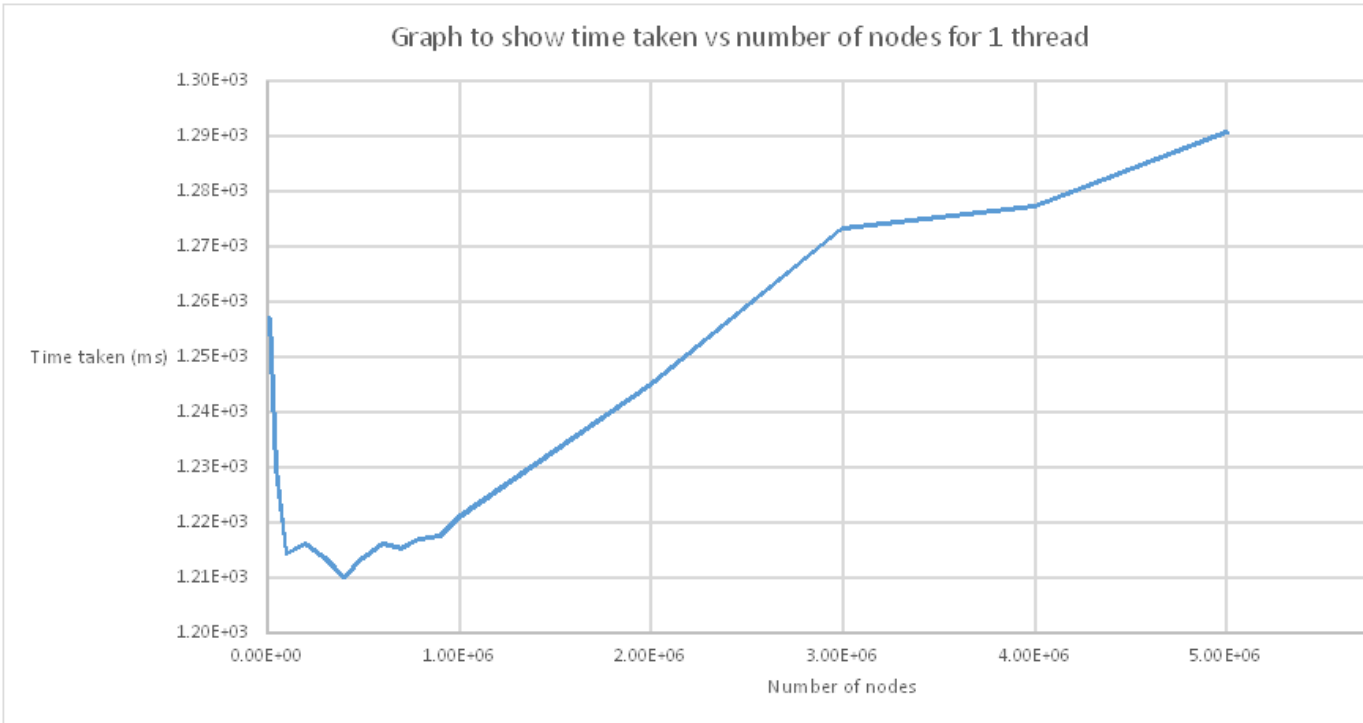

Figure 8: Cuda execution time for 1 thread

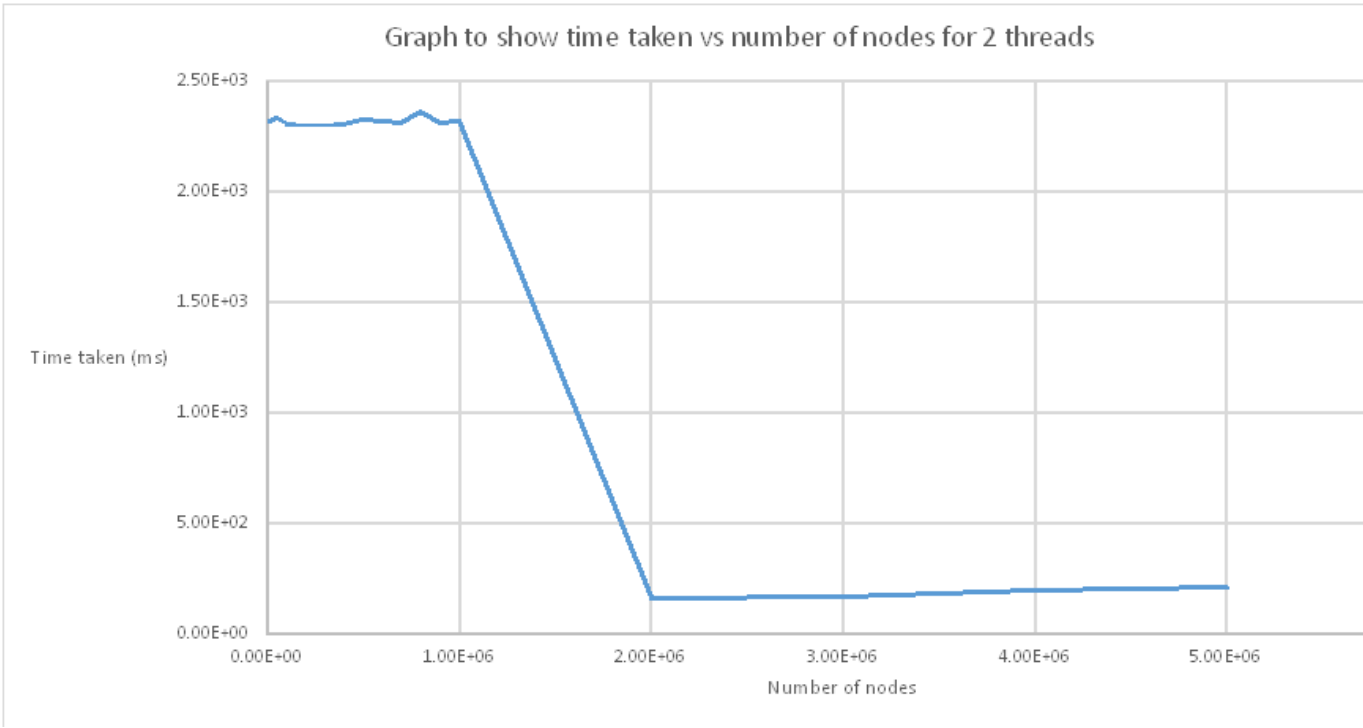

Figure 9: Cuda execution time for 2 threads

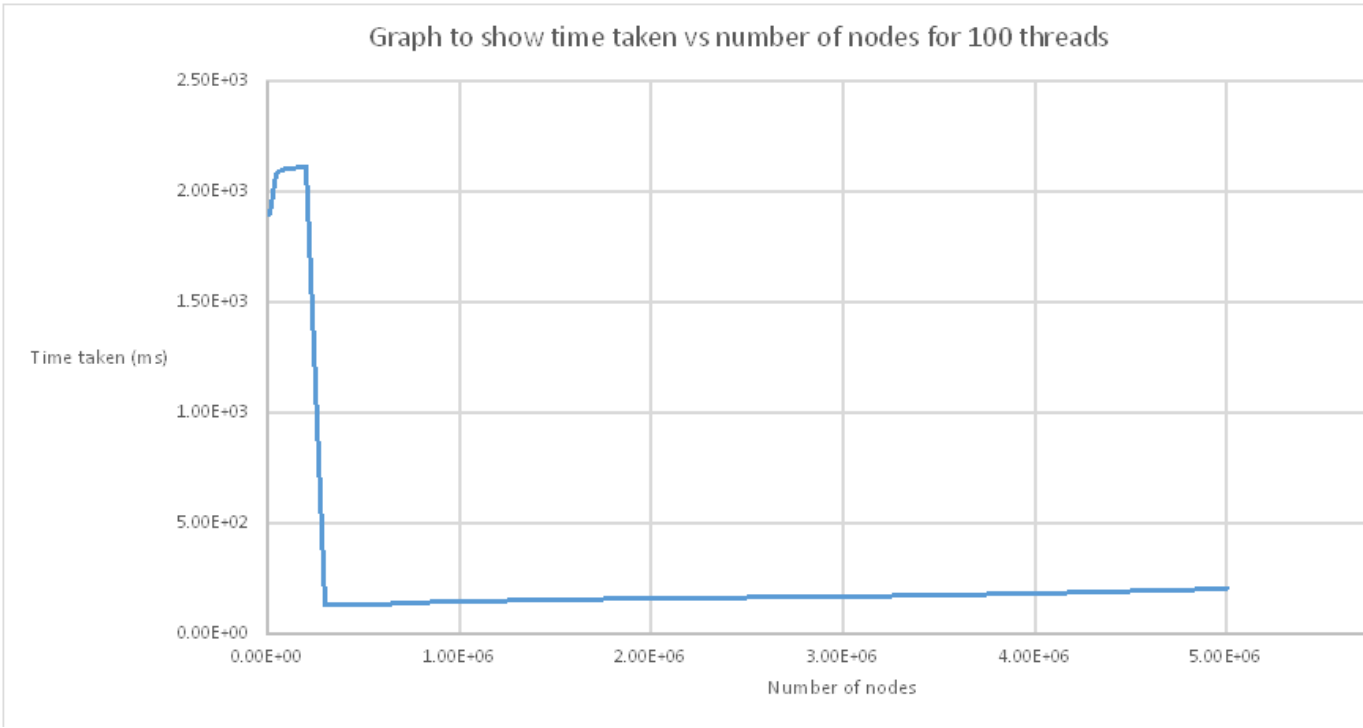

Figure 10: Cuda execution time for 100 threads

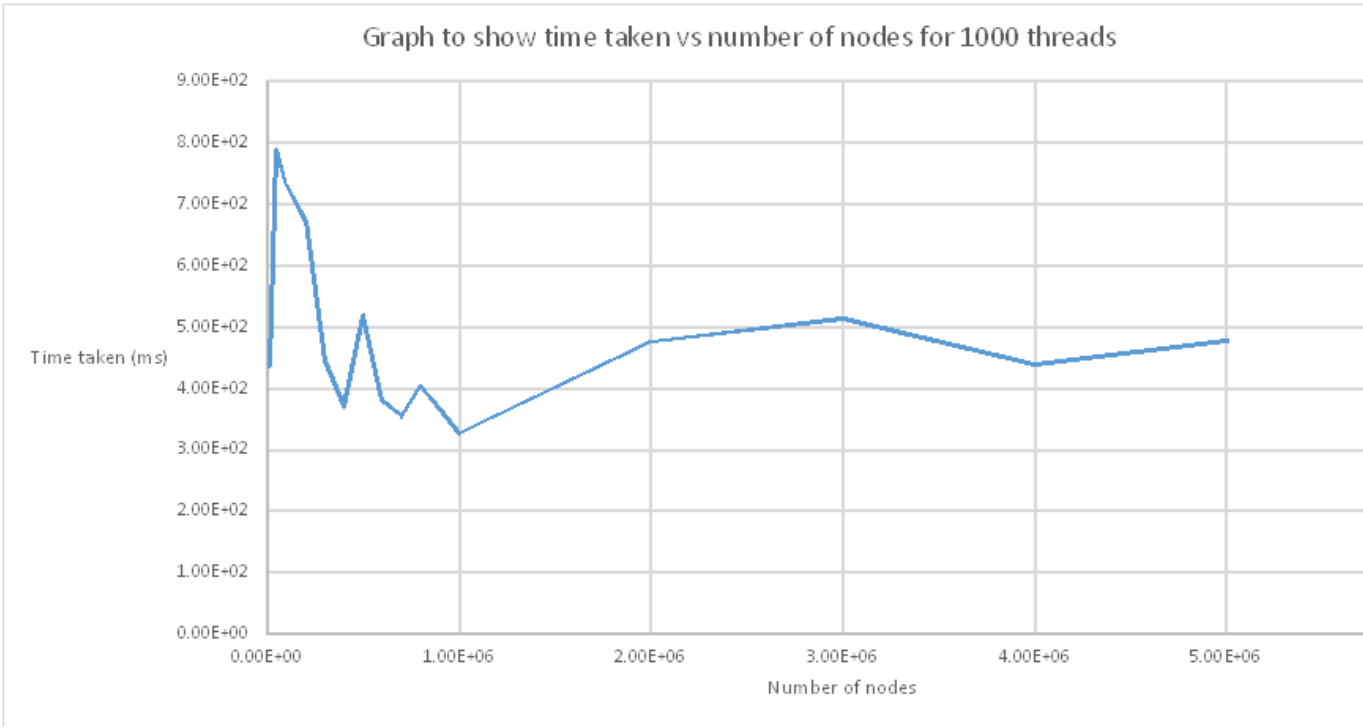

Figure 11: Cuda execution time for 1000 threads

0.6.2 Discussion on performance of results

Using the serial algorithm on the cluster and using anything over 3 million nodes resulted in the program being killed as there was too much computation for one process to do. From our results we found that increasing the board size produces a linear increase in time. In the serial implementation, 5 million nodes on a 4x4 board took 17.882812 seconds, while using MPI with 4 processes resulted 4.659657 seconds. This means there is a speedup of 3.838. Using MPI with 16 processes and the same setup resulted in 3.904706 seconds. This means there is a speedup of 4.580. The results for using 32 processes resulted in haphazard timing for a low number of nodes. This could be due to each process doing a very little amount of work and as such the problem size should be increased to keep the same efficiency as using 4 processes and 16 processes.

By running the serial algorithm using increasing board sizes, we found that as the board size is increased the percentage of leaf nodes that are solutions also increases. When usig a 4x4 board, only 12% of the leaves are solutions.However, using a 16x16 board, 99% of the leaves are solutions. This could be due to the increased space which allows for more correcting of mistakes and hence less states to be trapped in.

Complexity Analysis

Because the tree is asymmetric, the exact size of the tree cannot be determined. If we assume that the tree is complete then there would be 4^n nodes, where n is depth of the search. We found that the minimum depth is always around 1000. This means that is the tree were complete then there would be 4^1000 nodes.

0.6.3 Problems we encountered with this project

We had many problems in this project and found solutions or workarounds to a lot of these problems.

We initially attempted to implement an in-place depth-first-search because we believed that it would be more memory efficient than either a recursive or user-managed stack approach. We

found that the implementation was difficult and ended up changing our implementation to a user-managed stack.

We thought that we would be able to search the entire game tree for 2048 but it turns out that it is a very large game tree. This game tree is also asymmetric which makes it unlikely for the size of any given game tree to be accurately determined. We had to change our project slightly so that our AIs would only search a sub-tree from any initial state and then return a local minimum path from that search.

We initially used strings to represent our game state. Cuda does not support strings and we had issues managing states. We fixed this by moving to an array of ints. Attempting to code in a way which reduced redundant code for three different paradigms was difficult and sometimes impossible. We ended up having some forced redundancy when certain functions or methods were incompatible with one of our implementations, namely Cuda.

The Cuda implementation was the most challenging part of the project. Cuda is not very well documented and every new generation of GPU introduces new features and quirks. We worked on the 700 series which has now become deprecated. This means that ways of doing things which are now supported were not for us. A big issue is that Cuda does not like nested data in classes and will not copy such data from the host to the device or vice versa. Random number generation was also difficult because the cuRand library does not follow the same conventions as the main parts of the Cuda run-time, such as returning errors. The final issue we had was that indexing was very difficult because we had to flatten our class structure and use primitive 1D arrays where possible. We spent lots of time figuring out how our indexes worked. What we had to do in the end is flatten our data structures and only give the device each initial game board we had computed in serial. The GPU would then generate in device global memory each sub-tree (per thread) and return only the most optimal path and its respective thread. We know how many nodes were generated because we had to assume that the tree was complete and any node which was a dead-end became an empty node.

The cluster's restrictions killed the serial AI when it was busy executing 4000000 and 5000000 nodes.

0.7 Conclusion

Acknowledgements

We made use of the CUDA 8.0 SDK and its associated images and sample programs. We also used the lab examples from class as a way to figure out how to implement CUDA. We would like to thank Dr Richard Klein for his help in the use of Cuda.

Bibliography

[1] Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach (3rd Edition). chapter 5, pages "161–189". Pearson, https://www.amazon.com/gp/product/0136042597, 2009.