

Pipelining the A* Heuristic Search Algorithm

Kristopher W. Reese

Parallel Programming, Computer Science and Engineering

University of Louisville

Louisville, KY

kwrees02@louisville.edu

I. INTRODUCTION

The A* search is one of the most widely known and implemented search algorithms that has grown out of the early Artificial Intelligence era. It provides a method to find a shortest path in a map or graph by combining $g(n)$, the cost of movement from one state to another, and $h(n)$ the estimated cost to get from one node to the goal. These two portions of the search algorithm make up the heuristic used to minimize the total estimated solution cost in a map.[1]

Though this algorithm provides a good solution and minimizes the total number of nodes visited, there are instances where, in a very large world, the algorithm becomes slow and performs slower than one might want the algorithm to perform at. It was for this reason that we decided to look into pipelining the algorithm. By pipelining the algorithm we are able to maintain the admissibility of a heuristic and the true concept of Best-First Searching while gaining performance boosts by being able to check nodes asynchronously rather than the typical synchronous method of searching that the A* algorithm preserves.

Section 2 of this paper will discuss overcoming the necessity of maintaining a priority queue. Section 3 furthers this discussion by introducing the method for pipelining the algorithm in order for multiple processors to make use of the priority queue. Section 4 will explore the theoretical speedup and efficiency to determine if the Parallelized algorithm is cost-optimal, it will compare this to observed data results on large world maps. We then conclude the paper by discussing the understanding about Parallel Programming that this project has given us, and discussing potential methods for more efficient parallelization.

II. PARALLEL PRIORITY QUEUE

One of the first issues that arises when we discuss parallelizing any sort of heuristic algorithm is the necessity of maintaining Abstract Data Types (ADT) such as stacks, queues, or lists. The A* algorithm is such an algorithm, requiring a Priority Queue, which sorts the data in the queue, moving the best piece of data to the front.

These abstract data types pose challenges across multinode platforms such as the High-Performance Computing Cardinal Research Cluster located at the University of Louisville. Unlike typical home computers, there exists no shared memory between all of the nodes in the cluster. If we wanted to run this algorithm with a large

number of processes, we would be unable to use any sort of shared memory. In order to avoid the potential issues with shared memory, we can use another model that is common in parallel computing, the master-slave model. [2]

In this model, the root node, which we call the master, sends data to the other nodes in the system, the slaves. The slaves do the majority of the work in the system by computing information and sending the data back to the control, master node. The master node simply determines if it wants to store the data and if so, pushes the received information into the ADT. Figure 1 shows a visual description of a one master, one slave model.

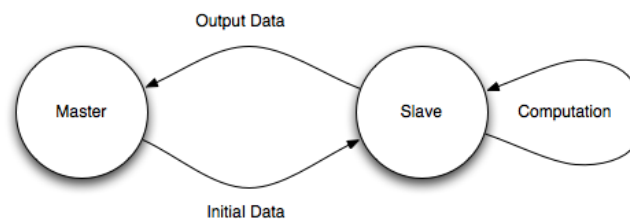


Figure 1. The Master Slave Model. The Master sends information to the slave which does various computations and returns the computed data back to the master control node.

For this project, the master process was used as the collector for all of the potential moves from the best current position in the priority queue. The master process pops a position node off of the priority queue, which starts with the initial starting position on the map. The master node then sends the information to one of the slave processes which computes the surrounding nodes, their weighting, and other necessary information. These computations are stored as a set of position nodes which is then returned to the Master node. Here, the master will determine if the nodes have previously been visited or are currently waiting in the priority queue and pushes the position node into the queue if neither of the tests are true.

Using this model, we can store a potentially infinite number of position nodes in the OPEN queue (Priority queue) or in the visited list. This also allows use to pipeline this algorithm without all of the processes needing the priority queues and avoids unnecessary All-to-All reductions which would create significantly more communication overhead and memory usage overhead. This assertion is further explored in Section 4 of this paper.

III. PIPELINING THE ALGORITHM

By using the Master-Slave model, we run into an unintended benefit when we move to pipeline the algorithm. This model allows us to use any number of potential slave nodes for our algorithm. Which means that for an infinitely large priority queue, the model would be able to reach maximum concurrency using a potentially infinite number of processes without any consequences. Despite the benefit, there is also an unintended consequence for using an infinitely large number of processes on the algorithm.

The primary issue with using a large number of nodes is the ramp-up time for the priority queue to reach a suitable size for all processes that we are using to have work. In the first iteration of the algorithm, we have a total of one item in the queue. This in return may result in a possible total of four suitable movements from this position. We can then use four processes to gather the information on each of the positions in the priority queue. Each of these then have four values, and each of these have four. This results in an exponential ramp up on the number of nodes that are being used at a given time. Figure 2 shows a tree of the ramp-up of the number of processes being used by the program.

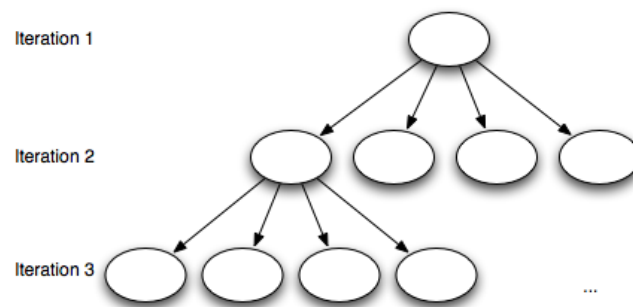


Figure 2. The Exponential Ramp-up of the Master-Slave model. If we assume that there were 16 slave processes across the three iterations, 15 of the processes would sit idle for the first iteration, 12 in the second iteration and 0 in the third iteration. An unintended consequence of using the Master-Slave model on Parallel Heuristic Searching

This poses new issues which need to be overcome such as idling processes until they are needed without causing deadlocks in the program. This can be achieved through polling on each of the processes. The polling will loop infinitely on the process until that process is killed by the master node. During each iteration of the loop, the slave node will check whether the Master node has sent information with a specific tag.

This check can be accomplished through the use of MPI's Iprobe function. MPI Iprobe is a non-blocking test for a message from a specific node. [3] Since we are using Polling for each of the nodes, we should also run each of the sends asynchronously. By running each of the sends from the Master Node asynchronously, we allow the Master Node to continue to do work while waiting on a response from the Slave node in which the data was sent.

The following section will discuss an analysis of the theoretical speed up for the algorithm and discuss the observed times for the serial and the implemented pipelined A* Search Algorithm as it was discussed in this section.

IV. ANALYSIS OF PIPELINED A*

The time complexity of the Serial A* Heuristic Search Algorithm is dependent on the heuristic that is used in the algorithm. In the worst case, the A* algorithm expands exponentially towards the shortest path. However,

the time is polynomial when the search is treated as a tree, we have only a single possible goal state, and when the heuristic meets the condition in Equation 1, where $h'(x)$ is an admissible heuristic.

$$|h(x) - h'(x)| = O(\log h'(x)) \quad (1)$$

The parallel time complexity of the implemented algorithm also becomes dependent on the heuristic that has been chosen for the algorithm. The time complexity of the parallel algorithm turns out to be more complex than the Serial algorithm and consists of three portions that make up our implementation of the pipelined A* algorithm. These include the Runtime of the algorithm overall, the overhead communication costs and the serial searching algorithm. Our equation for the running time is shown in Equation 2 where p is the number of processors and n is the size of the world.

$$T_p = \frac{\log h'(x)}{p} t_w + \log\left(\frac{n}{p}\right) t_c + n t_s \quad (2)$$

With this running time, we can determine the speedup of the algorithm by taking the serial time complexity and dividing the parallel time complexity. Equation 3 shows the equation for the speedup of the parallel algorithm in its most simplified form of the equation.

$$S = \frac{p \log h'(x)}{t_w \log h'(x) + t_c \log(n) + \frac{n}{p} t_s} \quad (3)$$

The efficiency of the algorithm is easy to find as well as we can simply take the Speedup function and divide by the number of processes we are using. Since we have a p in the numerator of the Speedup fraction, we can simply cancel this p out with the one that we would have divided by. Equation 4 is a representation of our efficiency function.

$$E = \frac{\log h'(x)}{t_w \log h'(x) + t_c \log(n) + \frac{n}{p} t_s} \quad (4)$$

We can use all of this information to then calculate the isoefficiency equation so that we can look at the scalability of the algorithm. To do this we want to find the K value and multiply the communication overhead into this to find the isoefficiency. Equation 5 shows the K value for the isoefficiency equation which is simply $K = \frac{E}{1-E}$. The K value has been put in its simplified form. The isoefficiency function of the algorithm is shown in theta notation in Equation 6.

$$K = t_c \log h'(x) + \left(t_w t_s \left(\frac{n}{p}\right) \log(n)\right)^2 + \left(t_s \left(\frac{n}{p}\right)\right)^2 \quad (5)$$

$$\Theta\left(\left(\frac{n}{p}\right)^2 \log^2(n)\right) \quad (6)$$

Even with the isoefficiency, it is hard to determine whether the algorithm is more scalable than the serial algorithm. There may be instances where this algorithm is more scalable if the heuristic that is being used is not the most optimal or where n and p result in a lower rate than the serial heuristic time complexity. The same holds true for the cost optimality in the equation. Equation 7 shows the test that we do for finding cost optimality for an algorithm and Equation 8 shows the test.

$$T_0(W, p) \leq O(W) \quad (7)$$

$$\log \frac{n}{p} \leq \log h'(x) \quad (8)$$

We can see that the algorithm is cost optimal where $\frac{n}{p} \leq h'(x)$. Knowing this we can determine that the parallel algorithm will be better in some special cases but may not always be the case. Running this algorithm on a large world shows that in some cases the pipelined algorithm runs faster than the serial algorithm. For this experiment we ran both algorithms on the same 300 by 300 map using the Manhattan distance Heuristic for searching. The Pipelined algorithm was run using 8 processes on a single node on the Cardinal Research Cluster.

We found that the serial algorithm ran about 21 seconds faster (at around 787 seconds) than the parallel algorithm, which took a total of just over 808 seconds. This experiment proves that in some instances the serial algorithm may be better than the pipelined algorithm. The experiments below support this hypothesis by showing that the pipelined algorithm may prove to be a faster algorithm for some instances.

We ran another experiment using the same map but with a much less optimal heuristic, a heuristic based on energy levels in conjunction with a euclidean distance measure. This was run on the same 300 by 300 map as the previous experiment using 8 processes on a single node for the parallel algorithm. During this experiment, we found that the parallel algorithm was much quicker at finding the most optimal path than the serial algorithm by about 30 seconds. This finding validates the math that was referenced in this section.

V. CONCLUSION & FUTURE WORK

We see that according to the experiments run and the results that were attained that pipelining this algorithm is yet another example of what Artificial Intelligence calls "No Free Lunch". This simply means that there is no singular solution to a set of problems and that a heuristic that might work well in one case may not work for other instances. These results show that even in the case of pipelining heuristic searching algorithms, we run into the same issues as serial algorithms where the serial may work best for one problem and the parallel may run best for a separate, but similar, problem.

This document also discussed potential ways to pipeline algorithms. It gave methods for polling on processes, allowing certain processes to remain idle until they are needed. We also discussed ways in which Data Structures might be used by multiple nodes without needing to store multiple copies of the structure or sending All-to-All broadcasts to keep the information in the data structure updated.

Despite this, there are still possibilities for further speeding up the parallel version of the algorithm. One place where we see a heavy workload on the Master node is in searching data structures for nodes in the system. It would be worthwhile to determine a parallelized algorithm that can use some of the idle nodes to help search the data structure to determine if a node has been visited or is in the queue waiting to be visited. We might also consider coming up with a way to allow only the master node to contain the entire world map. Currently each process requires a copy of the map be stored in their private memory. This is an ineffective use of memory and would be something to experiment with in the future.

REFERENCES

- [1] S. Russell, and P. Norvig *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River: Prentice Hall, 2010.
- [2] A. Grama, G. Karypis, V. Kumar, and A. Gupta *Introduction to Parallel Computing*, 2nd ed. Harlow, England: Addison-Wesley, 2003.
- [3] Argonne National Laboratories, "MPI_Iprobe", April 25, 2010. [Online] Available: http://www.mcs.anl.gov/research/projects/mpi/www/www3/MPI_Iprobe.html