Implementation of Parallel Path Finding in a Shared Memory Architecture

David Cohen and Matthew Dallas

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
Email: {cohend4, dallam} @rpi.edu

ABSTRACT

This paper investigates possible methods of parallelizing two common single source path-finding algorithms. This research could enable faster computation of optimal path planning in complex environments. Both algorithms, A and Bellman-Ford were found to be parallelizable. After testing, A* was shown to lack good scaling potential while Bellman-Ford showed promise as a scalable parallel path finding procedure.*

1 INTRODUCTION

Path finding is an important concept for a variety of real world applications such as robotic movement planning and decision-making. As the size of a graph increases, the computation time required to find the best traversal path increases exponentially. In a realistically modeled environment, this could mean extremely long wait times for shortest path finding. Parallel execution could split this job up between multiple processors and help to combat the inefficiencies arising from complex input data.

Very little previous research could be found about implementing parallel path finding. A team from Digital Equipment Corporation developed an algorithm in the early nineties to evenly split a path-finding problem across multiple processors [1]. Their implementation used the A*, a heuristic search algorithm, to find the best possible path across three variations on this algorithm, using both a message passing architecture and a shared memory architecture. In practice, the message passing system performed more efficiently than shared memory. This was determined to be because of the efficiency lost due to locks.

Though message-passing architecture may be more efficient for this problem, we determined that a multithreading implementation would be more appropriate for our end goals. Because our algorithm is intended for use on mobile robots, it seems far more likely to be run on a small multi-core system with shared memory than a large supercomputer.

Another implementation for parallel path finding was developed and described by Avi Bleiweiss at the NVidia Corporation [2]. Again, the A* algorithm was used but instead of a message passing system, this implementation used a multicore graphics processing unit using CUDA.

2 ALGORITHMS

Choosing an efficient algorithm is paramount to designing a strong scaling parallel pathfinder. In order to find the shortest path, most programs utilize a variant of Dijkstra's algorithm. We found in the limited amount of research done that the most common algorithm to use in parallel path finding is A*. We chose to experiment with this and a modification of the Bellman-Ford shortest path algorithm.

2.1 A*

A* works by iteratively creating a set of reachable vertices. Starting from an initial node, the algorithm will

find the shortest path to a goal node in a graph. The process starts at the initial node and branches to the vertex that it estimates to be closest to the goal vertex. This step is repeated until the goal is found or there are no vertices left to branch from. At each step, the vertex with the lowest estimated distance to the goal is chosen. The neighbors of this vertex are then inspected to see if there is a shorter path to them from it. If the vertex with the shortest distance is the goal then the shortest path to the goal has been found. If branching to new nodes has ceased and the goal was not reached, then there is no path from the start to the goal [3].

The A* algorithm is heuristic in that it uses an estimate of the distance to the goal in its computation. As long as this estimate is less than the actual shortest path, the algorithm will function properly. In real world spatial path finding, the physical distance of a line between the start and goal can be used as an estimate as no path could possibly be shorter. Figure 1 shows pseudocode for the A* algorithm.

```
AStar(graph, start, goal)
    foreach v in graph.vertices
        v.g_score = infinity
        v.h_score = heuristic(v, goal)
        v.previous = null

    openset.add(start)
    start.g_score = 0
    start.f_score = start.h_score

    while openset is not empty
        let x be the node with lowest f_score in openset
        if x is the goal then
        return true

        openset.remove(x)

        foreach y in x.neighbors
            let newg = x.g_score + distance(x, y)

            if newg < y.g_score then
                y.previous = x
                y.g_score = newg
                y.f_score = y.g_score + y.h_score

                if y is not in openset then
                    openset.add(y)

    return false
```

Figure 1. Pseudocode for the A* algorithm

2.2 Bellman-Ford

The Bellman-Ford algorithm is very similar to Dijkstra's algorithm. However, unlike Dijkstra's algorithm, Bellman-Ford does not use greedy selection of a single edge. At each step, every edge is checked to see if it provides a shorter path to the destination vertex. This is repeated |V|-1 times since in graphs without negative cycles, the shortest path can only visit every vertex once. If the distance to the goal is infinite after all the repetitions then there is no path to the goal. Figure 2 shows the pseudocode for the Bellman-Ford algorithm.

An advantage of the Bellman-Ford algorithm is its ability to detect and deal with negative length cycles. Neither the A* nor Dijkstra's algorithm are capable of computing shortest path in environments where negative length cycles exist. This ability is unique to Bellman-Ford and Bellman-Ford like algorithms.

```
BellmanFord(graph, start, goal)
    foreach v in graph.vertices
        v.distance = infinity
        v.previous = null

    start.distance = 0

    for i = 1 to (graph.nvertices - 1)
        foreach v in graph.vertices
            foreach u in graph.vertices
            if there is an edge from v to u

                if (v.distance + distance(v, u)) < u.distance
                    u.distance = v.distance + distance(v, u)
                    u.previous = v

    return (goal.distance != infinity)
```

Figure 2. Pseudocode for the Bellman-Ford algorithm

3 PARALLEL IMPLEMENTATION

In order to build the parallel path-finding program, we used pthreads in the C programming language. Pthreads will run on any Unix like system, maximizing the portability of this program across a wider array of different machines. Because this program is potentially meant to find paths for mobile robots, the computer system running it is far more likely to support pthreads than a message passing interface. The serial algorithms above had to have their computational steps divided among multiple threads in order to make them run in parallel.

3.1 Parallel A*

In order to run the A* algorithm in parallel, each thread is a producer and consumer upon the open set of vertices. A thread will consume the vertex that it estimates to be the closest to the goal. The thread will then compute whether going through this node produces a better path for any of its neighbors. Any neighbor value that is recomputed in this way will be reentered into the open set. Then the thread moves on to consuming the vertex that is now the best estimate.

Because this implementation is optimistic the termination conditions had to be modified from the serial A* algorithm. Unlike classic A* a path to the goal can be found that is not the shortest path. The program must not terminate immediately upon finding a path or else it may report a suboptimal solution, instead it should set a global flag to true indicating that a path has been found. If a thread detects no vertices in the open set that it can consume, it sleeps for a short time. If this condition is met repeatedly, the thread terminates. Once all threads have terminated, the computation is complete. If the global flag representing a path found to the goal is set to true, then the shortest path has been found. Otherwise, there does not exist a path from the start node to the goal node.

In order to stop threads from overwriting the work of other concurrent threads, a large number of locks were put in place. The open set was locked in order to avoid non-deterministic behavior of adding and removing vertices from it concurrently. In order to prevent vertices from being updated with partial data, locks were placed on the values associated with each vertex. Originally, the built in pthreads mutexes were used, but these performed poorly with high frequencies of lock contentions. In order to alleviate this slightly, a custom lock was designed using the gcc __sync atomic operations.

3.2 Parallel Bellman-Ford Variant

In the Parallel Bellman-Ford algorithm, the graph was subdivided evenly amongst the threads. An adjacency matrix was used as the graph's data structure for its simplicity. Due to the use of this method, each vertex was readily aware of all edges incident to it. All vertices but the start node begin with a distance value of infinity. Each thread iterates through its set of vertices repeatedly and checks if a shorter path to this vertex has been discovered.

A barrier was added such that after each complete loop through its set of vertices, a thread will wait for all other threads to catch up. This ensures that changes to the distance values of vertices can be disseminated to connected vertices in other threads before the next iteration is begun. In order to avoid memory bottlenecks, a thread will create a copy of every vertex's distance value before each iteration over the node set begins. Another barrier is employed to synchronize the threads after the local copy is made.

4 EXPERIMENTS

For testing, graphs were randomly generated based off of four tunable parameters; the number of vertices, the degree of each vertex, the maximum weight of each edge and an initial seed for the random number generator. The RPI Computer Science Cluster was used to test the programs. This system consisted of four dual core AMD processors. Tests were performed multiple times to ensure better accuracy. Numerous test cases with varying graph parameters were run to examine the effects on the run times of the algorithms. Figure 3 demonstrates the various configurations of graphs used to test the algorithms.

Graph	# of Vertices	Degree
G3	2000	3
G4	4000	3
G5	2500	1000
G8	3500	1000
A1	2000	1
A2	2000	10
A3	2000	100
A4	2000	1000

Figure 3. Table of graph parameters used in testing

1 RESULTS

The initial implementation of A* did not show any significant speedup. In fact in the two test cases that were run for this algorithm, it showed negative speedup when running on additional processors. Slight speedup was gained when the pthreads locks were replaced by our custom locks, but this scaling was still insignificant. Figure 4 shows the small speedup for parallel A*. Due to our inability to find a better method for parallelizing the A* procedure, investigation into this algorithm was discontinued.

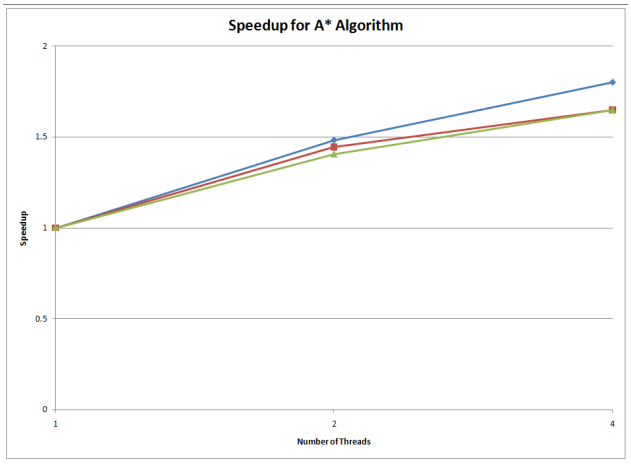

Figure 4. Graph of speedup for the A* algorithm

The Bellman-Ford algorithm showed significantly better scaling on up to four processors. The scaling, while not

quite linear, was far better than A*. However, when the program was run on eight processors the scaling either remained constant with that of four processors or decreased. Figure 5 shows the scaling for the parallel Bellman-Ford algorithm for 1, 2, 4 and 8 processors.

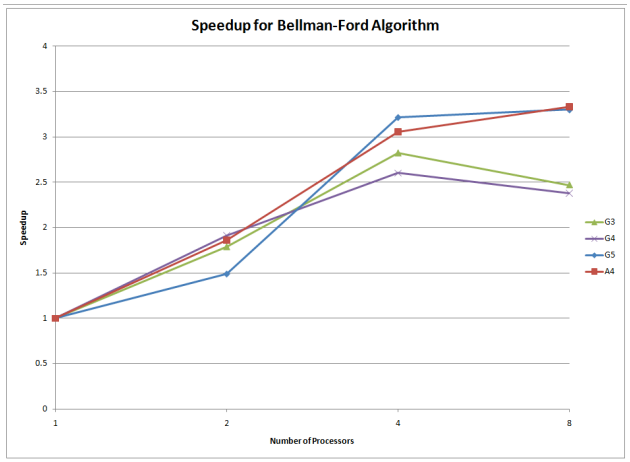

Figure 5. Graph of speedup for the Bellman-Ford algorithm

The Bellman-Ford algorithm is extremely sensitive to the number of vertices that make up the graph. Runtime takes much longer as the vertex count increases. Figure 6 shows these effects.

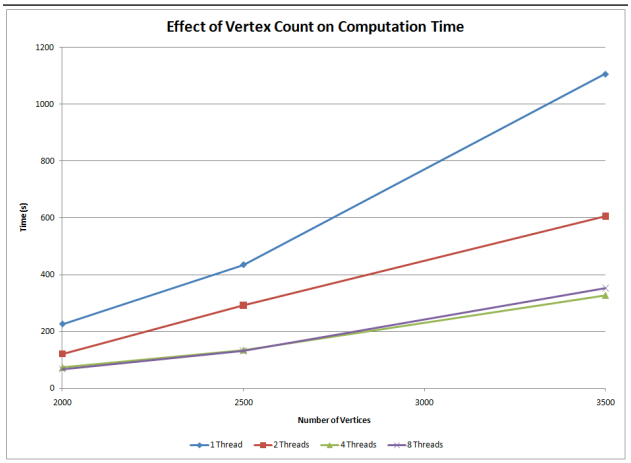

Figure 6. The runtime of the Bellman-Ford algorithm on graphs of increasing vertex counts

Unlike the number of vertices, the number of edges contained in the graph does not have a significant effect on computation time. Figure 7 shows the relationship between runtime and the total number of edges in a graph. Not until the graph becomes extremely dense does the runtime increase noticeably. The number of vertices in a graph affects the computation time far more than the number of edges when using a Bellman-Ford algorithm.

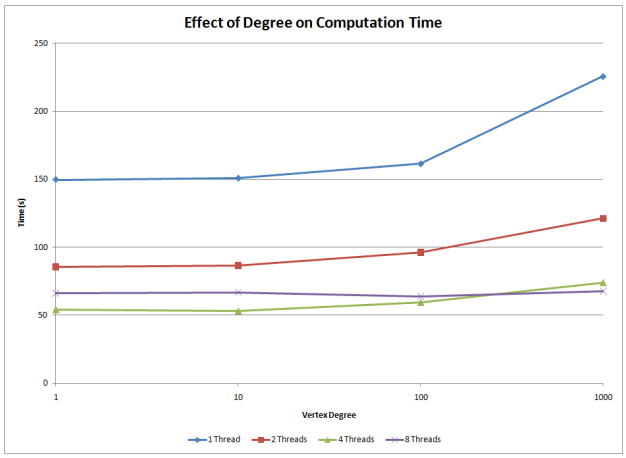

Figure 7. The runtime of the Bellman-Ford algorithm on graphs of increasing number of edges

1 CONCLUSION

The A* algorithm showed very minimal scaling. This is likely due to the large number of locks employed. These locks were necessary to ensure the correctness and optimality of the results but prevent it from achieving good scaling.

The Bellman-Ford algorithm dropped off in scaling after four threads. This is most probably due to the effects cache coherency between the separate processors. It can also be easily shown that our algorithm is order $O(N^3)$ where N is the number of vertices in the graph. This is purely a downside of using an adjacency matrix as a data type for the graph as there is no simple way to loop over the edges. The overhead of the number of edges is very minimal and therefore does not reflect in the overall complexity of the problem.

1 FUTURE WORK

Futures studies could look into developing a method of A* that requires fewer locks. Instead of an array to implement the open set, a priority queue could be utilized. This could potentially alleviate some of the overhead associated with adding and removing vertices.

More research should be done to determine the exact cause of the decline in scaling after four threads. Once this is found, a solution might present itself. In order to cut down on memory usage the graph could be implemented as an array of lists instead of an adjacency matrix.

REFERENCES

[1] CVETANOVIC Z., NOFSINGER C.: Parallel AStar Search on Message-Passing Architectures. *System Sciences, Proceedings of the Twenty-Third Annual Hawaii Conference*, 1 (1990), 82–90. 97–104.

[2] BLEIWEISS, A. 2008. GPU Accelerated Pathfinding. In Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 66–73.

[3] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics SSC4* **4** (2): 100–107.

[4] Richard Bellman: *On a Routing Problem*, in Quarterly of Applied Mathematics, 16(1), pp.87-90, 1958.