Applying Parallel Programming to Path-Finding With the A* Algorithm

William Miller

Abstract – **The A* algorithm is a popular choice for game developers when implementing artificial intelligence for path-finding purposes. These calculations are typically carried out on a single thread and are often modified for efficiency at the cost of admissibility. Parallel computing can be leveraged to provide both speed and accuracy while scaling to support the trend of increasingly numerous entities in the game setting.**

Index Terms – **A* Algorithm, Parallel Computing, Path-Finding**

I. INTRODUCTION

Artificial intelligence operations in general (and path-finding work in particular) are a costly but necessary feature in electronic games, and are therefore a popular target for optimization [4]. The typical solution is to create a separate thread for these calculations to run in, away from the main game loop [4]. With the increasing availability of cores in personal computers and servers, mirrored by a trend of an increasing numbers of actors on the screen at the same time, allocating a single thread for path-finding may not be an efficient use of resources and could impact the user experience [5].

Path-finding is an high-profile and necessary aspect of next-generation games. The need for a computer agent capable of displaying a competent level of skill in solving the path-finding problem becomes obvious in scenarios where a human player competes with or alongside a computer cohort, such as in the First Person Shooter or Real-Time Strategy genres. The entertainment value of a game is greatly reduced when characters are unable to reliably traverse complex terrain and such failures are often starkly visible to the user. Given the impact of poor path-finding on a user and the inherently expensive nature of path-finding operations in general, game developers are under increasing pressure to deliver intelligent actors utilizing limited system resources [5].

The A* algorithm is one of the more popular solutions used to implement path-finding in games. The algorithm is guaranteed to find the shortest path between two points given a good heuristic, though it can construct a large search tree in the process (consuming time and memory). Implementations generally modify the algorithm to explore a smaller search tree, reducing the time to generate a path by sacrificing the guarantee of the result being the best possible path [2]. These optimizations are reasonable considering the increasing number of entities found in most games, representing an increasing number and frequency of simultaneous path-finding problems. A game environment is extremely sensitive to latency and users generally have a much lower tolerance than in other applications. Modern developments in hardware relax the memory constraints on gaming software however, and parallel programming techniques can be applied to improve the speed of these calculations in respect to sequential processing [1].

II. BACKGROUND

The problem of implementing path-finding is directly related to the increasingly massive number of actors that are active in the player's vicinity. Next-generation games advertise the capability of rendering several thousand units on the screen simultaneously, with each possessing distinct artificial intelligence [6]. Considering that the screen represents only what a player can see in the game world, it is possible that many more actors may exist outside the player's field of vision (requiring no rendering but still necessitating path-finding) [6].

The problem size scales dramatically when multiplayer games are considered. Recent massive multiplayer games have achieved numbers in excess of 50,000 simultaneous users [7]. As games attract more users, these numbers could potentially rise.

[Diagram of A*]

In path-finding, the A* algorithm essentially operates by exploring the nodes in a map, tracking the distance a node lies from the starting point while estimating how distant a node might be from the goal [1]. The sum of these two values forms a node's "cost", which is used by the algorithm to determine in what order nodes should be considered when exploring. It is important to note that the estimate of a node's distance from the goal is produced by a heuristic function, which can underestimate this distance but may never overestimate it for the heuristic to be considered admissible [1][2][8].

[Diagram of Manhattan distance and vector crossproduct]

One such heuristic is known as the Manhattan distance. Used primarily in grid-like environments, it takes the sum of the vertical and horizontal displacement between nodes to be the estimate of the actual distance between the nodes, regardless of obstacles [8]. The Manhattan distance produces a lower bound in grid simulations as two points on a grid are at least the Manhattan distance apart, and each point

on a grid is exactly one unit distant from its neighbors [8]. While guaranteed to be admissible, this heuristic tend to result in a path that mirrors this Manhattan distance, consisting of a single long vertical path and a long horizontal path between two nodes. While this is technically correct, it can be easily modified to take into account the vector cross-product between the start-goal vector and the current-goal vector, resulting in a "straighter" path between two nodes [9].

III. LITERATURE REVIEW

Several other works provide insight into the area of path-finding in gaming. One such work was a paper written by Jad Nohra and Alex J. Champandard this year on recent developments in parallel path-finding using modern computer hardware [4]. This paper briefly covers the focus of the game industry on specific aspects of path-finding and explores in-depth the impact of parallel computing on performance, especially as it is effected by a variety of map conditions. Other research done by Adi Botea, Martin Mûller and Johnathan Schaeffer discusses optimizations common in the game industry to work around speed and memory constraints and proposes a new version of A* to reduce problem complexity [2]. Variations such as this one hold off on calculating the complete path from start to end, instead finding a route to the general area of the goal before continuing the calculation. While not considered for the purposes of this paper, alterations such as these can spread out the load of path-finding calculations, though they potentially lose accuracy as a result.

Another work researched by Pierre Pontevia describes path-finding as the basis for nearly all games that involve moving characters. Pontevia describes the current state of path-finding in games as being imperfect, as the use of A* to find a path between two locations is the easy portion of path-finding. The challenge comes with implementing allowances for things such as a dynamic world with a need to calculate a new route when conditions change, other characters in proximity affecting the character or its choice of a path, and many other conditions that make path-finding more complex than simply implementing the A* algorithm. Pontevia also explains that traditional tricks to work around fully developing path-finding (such as using a simplified version of the terrain, contriving to make it unnecessary for characters to move over difficult terrain, or manually creating checkpoints during development for characters to follow instead of dynamically calculating a route) will no longer work convincingly in next-generation games [5].

Han Cao, et al. wrote a paper on applying parallel programming techniques to several algorithms for the purposes of applying them to the optimal path problems handled by the Intelligent Transportation System in China. They specifically considered the Dijkstra Algorithm, Nearest E-nodes Algorithm and the Hierarchical A* Algorithm, analyzing them in Xi'an road networks with thousands of nodes and paths. An interesting feature of the paper is a section discussing the challenge of efficiently balancing the load across the nodes in a parallel system, and the policies available in OpenMP for accomplishing this [1].

Heuristics are an integral part of the A* algorithm, and Richard Korf provides invaluable information in his paper, "Recent Progress in the Design and Analysis of Admissible Heuristic Functions". He explores application search algorithms and heuristics to solving puzzles including the Fifteen Puzzle and the Rubik's Cube, using algorithms A* and Iterative-Deepening-A*. The heuristics that Korf examines for these applications are Pattern databases and Rubik's Cube Pattern databases, both specifically tailored to the problems mentioned above. He proposes the use of Disjoint Pattern databases, which is far more accurate and efficient in solving these puzzles.

The final source considered was a paper by Ko-Hsin Cindy Wang and Adi Botea which examines multi-agent path planning. They propose an algorithm called Flow Annotation Replanning to combat the technical problems involved in global searches by multiple entities. FAR endeavors to implement flow restrictions to prevent head-on collisions as well as to reduce the search space an entity must consider when planning a route. This method does implement the possibility of deadlock, which the algorithm is capable of resolving. The testing methodology described in this source is the basis for the methodology used in the experiments of the current paper [3].

IV. APPLYING PARALLEL COMPUTING

The problem of path-finding lends itself to being easily broken down in two main ways. The first approach is to break each pathing problem in two, starting from both the start and end nodes simultaneously [10]. This variation of A* (also known as Bidirectional Search) allows a problem to be split between two cores, but it does not easily scale upward with a greater number of cores or a larger problem size [10]. This approach clearly requires two free cores per pathing problem and so would be unable to run a particular problem given less than that amount [10].

[Diagram of Bidirectional Search]

The second intuitive approach is to take each pathing problem as a single unit, handing a new unit to each core as it becomes available. This approach scales easily with any number of cores and is possibly the most efficient means of distributing large data sets [4]. This second approach is the technique used in the experiment described later in the current paper.

V. EXPERIMENT

To demonstrate the impact of parallel computing on path-finding, the A* Algorithm (utilizing the Manhattan

distance heuristic with the vector cross-product tie-breaker described in the Background section) is used by 150,000,000 independent actors operating in a large 50 x 50 grid world. This number of actors was arrived at by multiplying an estimated 3,000 actors by the roughly 50,000 players described in section 2. A start and end node are chosen for each entity at random, and the experiment runs until all entities have resolved a path to their respective destinations. This simulation is then be re-run for a number of cores ranging from 1 to 4.

[Image of example grid map, generated in grayscale]

The actual implementation of this experiment is be done in Java, utilizing the language's multithreading support. The map grid is implemented as a simple 2D array of Squares, where each Square represents a node with its own heuristic value and terrain cost. The heuristic value is calculated on creation using the modified Manhattan distance heuristic described in earlier sections, and the terrain cost is assigned from a list of reasonable constants representing terrain types of varying difficulty.

Each actor entity is a class containing data identifying its random start and end nodes, the final path between the two as well as a copy of the map grid described above, implemented in a similar fashion but with support for remembering the path cost of reaching each node. This is accomplished through the use of a parent node pointer, which represents the Square previously traversed to reach the current Square. These pointers (commonly used in A* [1][2]) are set as the actor explores the grid using A*, and allow the actor to backtrack once the goal node is reached to establish the path that was traversed. For memory efficiency, each actor's map copy is only instantiated when the actor is actively calculating a path and is destroyed after the path is found.

[Diagram of overall architecture showing threads]

After all of the entities are created and given their start and end node assignments, a number of threads are spawned (ideally equal to the number of available cores) which begins completing the path assignments for the list of entities. The threads pull actors as needed from a shared list using a Synchronized method to avoid race conditions. This is preferable to assigning 1/N actors to each thread (where N is the number of threads spawned) because with such a large data set of randomly generated start and end points, it is conceivable that some threads may complete more quickly than others and become idle.

VI. RESULTS

[This section cannot be written until the experiment is complete. It will contain data on the time the experiment takes to complete for different numbers of cores, a comprehensive scalability analysis, and charts visually describing all relevant data. I estimate it will fill roughly a full page.]

VII. CONCLUSION

It has been shown in previous works that path-finding in gaming can benefit from parallel computing [3][4]. As advances in technology give commercial users access to cheaper and more powerful hardware, this type of optimization will become far more accessible to game developers. The stop-gaps used by developers in the past to sidestep the problem of path-finding and convincing artificial intelligence in general will no longer be acceptable as games demand more interaction with acting, moving characters. Even assuming developers embrace the potential of parallel programming to ease this burden, the art of path-finding is far from perfect and much work remains to be done [5].

References

[1] H. Cao et al. “OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis”, *Proceedings of the 2009 WRI World Congress on Software Engineering – Volume 01*, pp. 61-66, 2009.

[2] A. Botea et al. “Near Optimal Hierarchical Path-Finding”, *Journal of Game Development*, Vol. 1, pp.7-28, 2004.

[3] K. Wang and A. Botea, “Fast and Memory-Efficient Multi-Agent Pathfinding”, *ICAPS*, pp. 380-387, 2008.

[4] J. Nohra and A. J. Champandard, “The Secrets of Parallel Pathfinding on Modern Computer Hardware”, *Intel Software Network*, Intel, 2010.

[5] P. Pontevia, “Pathfinding is Not A Star”. Autodesk, white paper. pp. 1-6, 2008.

[6] J Rodzik, “99 nights E3 Preview,” *www.seriouszone.com*, May 12, 2006. [Online].
Available: http://www.seriouszone.com/cms/articles/42_1.php .
[Accessed: June 26, 2010].

[7] S. Brennan, “Eve Online celebrates 54,446 simultaneous users, a new record,” *www.massively.com*, January 4, 2010. [Online].
Available: http://www.massively.com/2010/01/04/eve-online-celebrates-54-446-simultaneous-users-a-new-record/ .
[Accessed: June 26, 2010].

[8] R. Korf, “Recent Progress in the Design and Analysis of Admissible Heuristic Functions,” *Lecture Notes in Computer Science*, pp. 45-55, 2000.

[9] A. Patel, “Game Programming,” *theory.stanford.edu*, February 16, 2010. [Online].
Available: view-source:http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html .
[Accessed: June 26, 2010].

[10] J. Chang, “Making the Shortest Path Even Quicker,” *research.microsoft.com*, July 7, 2009. [Online].
Available: http://research.microsoft.com/en-us/news/features/shortestpath-070709.aspx .
[Accessed: June 26, 2010].