INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 6.078
(Volume 6, Issue 4)
Available online at: www.ijariit.com

Parallel A* search on a multi-core CPU

Ishaan Jaffer
ijaffer@andrew.cmu.edu
Carnegie Mellon University, Pittsburgh, Pennsylvania

ABSTRACT

Multi-core central processing units (CPU) and the graphics processing unit (GPU) have become popular parallel computing platforms in recent years [1]. The GPU platform is commonly adopted in the research community as it has been to be superior to the traditional CPU. Straightforward implementations of Parallel algorithms on a GPU can easily achieve speedup of ten times or more over the sequential algorithms. However, achieving significant speedup on a multi-core CPU (over the sequential algorithm) requires intelligently designed and well optimized algorithms [3]. This paper discusses a parallel implementation of A* search which achieved 6.67x Speedup with a search space of 10^6 nodes, 3.14x speedup with 10^7 nodes and 137.67x speedup with 10^8 nodes when run on the eight-core, 3.0 GHz Intel Core i7 processor. This paper also analyses different work partitioning strategies and how the performance of the parallel A* search algorithm scales.

Keywords— *Parallel Computing, A* Search, Search Algorithms, High-Performance Computing, Multi-core, CPU, GPU, Speedup, Threading, Parallel A* Search*

1. INTRODUCTION

Graphs are the fundamental data structure used in fields of computational biology, robotics, understanding social interaction and various forms of relational data [1], [2]. Due to the massive data-set sizes of these applications and the rising popularity of "big data" they have required increasingly long periods of computing time [2].

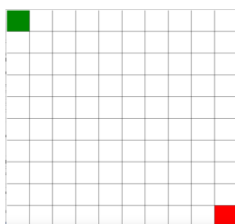

Fig. 1: Setup of the search space (start: green, end: red)

Straightforward implementations of parallelizing the A* algorithm rarely provide any significant speedup due to the random nature of the memory access patterns - a fundamental property of graph algorithms [5]. For the purpose of performance measurement, the search space is a 2-D grid with the start at the top left and end at the bottom right (Chart-1). This configuration was selected because it is the most computationally heavy and takes the longest time to search.

2. SEQUENTIAL A* SEARCH

A* search is guided by a heuristic function and is one of the most widely used best-first search algorithms in artificial intelligence [4]. By design A* search is a sequential algorithm due to which several attempts at parallelizing it efficiently have proved to be challenging [6]. The key data structures in any A* implementation are:

- The closed list - stores all the visited nodes. For this implementation each node struct/structure has a Boolean variable indicating if it has been visited. Hence this implementation does not require a closed list.
- The open list - stores the states that have not been visited. The open list uses a priority queue to store the nodes in a particular order.

The open list priority queue provides us with O(log N) insertion and deletion operations. The nodes in the open list are sorted according to a heuristic function f(node):

$$f(node) = g(node) + h(node)$$

Where *g(node)* is the cost from the starting node to the current node and *h(node)* is the cost from the current node to the end node. For all A* implementations used in this paper, the cost is defined as the Euclidean distance between any two nodes (equations below).

$$g(node) = (start.x - node.x)^2 + (start.y - node.y)^2$$
$$h(node) = (node.x - end.x)^2 + (node.y - end.y)^2$$

The major operations of the sequential A* algorithm are:

- Dequeuing a node from the open list priority queue.
- Checking if the current node is the end or has been visited.
- If the current node is not visited, then marking it as visited.
- For each neighbor of the current node - calculating the heuristic of the neighbor and enqueuing the neighbor to the open list.

3. PARALLELIZING A* SEARCH

Language and Libraries Used: C, C++, OpenMP
Machine: Eight-core, 3.0 GHz Intel Core i7 processor

© 2020, www.IJARIIT.com All Rights Reserved

3.1 Parallelizing the calculation of the heuristic

For each node in the graph there are 8 neighbors and for each neighbor the heuristic needs to be calculated. In this approach the 8 heuristics were calculated in parallel and the nodes were then sequentially added to the priority queue. With the given setup it was found that this approach provided limited speedup and scaled poorly, furthermore the speedup depends on what heuristic is being used. For computationally heavy heuristics there was a significant speedup while a slowdown was observed for common heuristics like Manhattan distance and Euclidean Distance. The challenge with parallelizing A* is creating a mapping that works independently of what heuristic is used. This approach did not take much advantage of the 8 cores available on the machine and led to a **speedup of only 1.1x** on a 100x100 (10^4 nodes) graph.

3.2 Providing each thread with its own open list priority queue

For a machine with T cores this approach would initialize T priority queues, with each thread exploring and execute A* search in its own assigned region of the search space. The data flow for this approach is shown in fig.2.

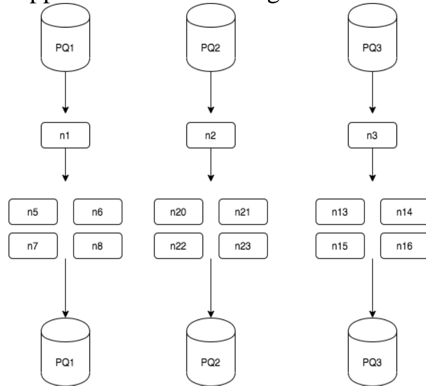

Fig. 2: Parallel A* with independent priority queues

Since each of the T threads can run an A* search independent of the other T-1 queues, this eliminates any time that would have been lost due to locking and unlocking a shared Priority Queue or any time spent on communication between the threads.

In order to use this technique of parallelization, the search space needs to be appropriately partitioned and the following conditions need to be held True to ensure the correctness and efficiency of A* (when run with T threads):

- During the initialization of each priority queue, any node added to PQ1, PQ2, PQ3…PQT needs to have a path from the start to the node.
- When the nodes are distributed amongst PQ1, PQ2, PQ3...PQT each PQ should have a fair distribution of nodes. If one thread was only given all the lowest priority elements, then there would be a large amount of work imbalance and idle time for the rest of the cores.

To ensure the conditions above are held true, a combination of depth limited A* search with parallel A* search is implemented.

3.3 Limited A* Search with Parallel A* Search

Limited A* search does not visit the entire search space but instead visits the nodes in the bounded search space. The limited A* search is run sequentially and visits a subset of the search space, adding nodes to the shared open list priority queue [7]. The steps in this proposed parallel implementation are the following:

- Depth limited A* to initialize T open list priority queues.
- Each thread runs an A* search using its own open list and explores a subset of the search space.

Chart-3 illustrates the main steps in this final approach.

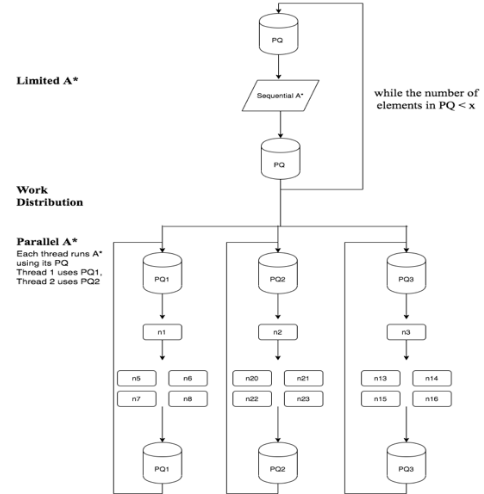

Fig. 3: Depth limited A* with Parallel A* (n1, n2…n16 are nodes in the graph)

The limit or constraint placed on the limited search can heavily influence the overall performance. For the purpose of this paper, the size of the shared open list priority queue is used to decide when the limited A* should terminate. This is discussed in more detail in the Work Partitioning Section.

3.4 Work Partitioning

In order to minimize idle time for each core the following aspects of the current parallel algorithm are optimized:

1. When the Limited A* should terminate. This is controlled by the size of the Priority Queue.
2. After the Limited A* terminates the manner in which nodes are distributed between the Priority Queues of Individual Threads.

To best decide at what size of the priority queue the limited A* should terminate the maximum allowed size of the PQ was varied and speedup on a 1000x1000 (10^6 nodes) graph was measured. All sizes are multiples of 8, since this was run on an 8-core machine. Having multiples of 8 in the priority queue ensures that each thread can gets an equal number of nodes to work on.

Table 1: Percentage of nodes explored by limited A* and time taken

% of Total Nodes	Size of PQ (rounded to closest multiple of 8)	Time Taken (seconds)
0.1%	1000	0.0120
0.05%	504	0.0070
0.025%	248	0.0040
0.01%	104	0.0023
0.005%	48	0.0020
0.0025%	24	0.0017
0.001%	8	0.0014

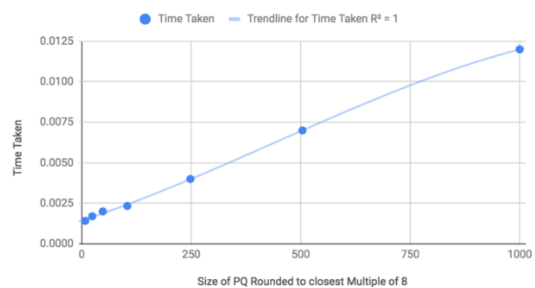

Fig. 4: Total Time Taken vs Size of PQ

© 2020, www.IJARIIT.com All Rights Reserved

From the measurements it was found that minimizing total time spent running sequential limited A*, maximized total speedup. Specifically, terminating sequential A* search when the shared open list PQ consisted of 0.001% of the total search space nodes led to the best overall performance of the parallel A*.

3.5 Work distribution between individual threads

After 0.001% of the total search space nodes are in the Priority Queue the nodes need to be distributed amongst individual threads. To find the best partitioning scheme Round Robin Partitioning, Interval Partitioning and Greedy Partitioning were tested. Round Robin Partitioning had the lowest standard deviation in total priority of each priority queue (STD DEV = 2211.21), this led to the best distribution of the most promising nodes amongst threads (See Appendix A for detailed results).
To summarize, the final parallel implementation consisted of the following:

- A limited sequential A* to explore 0.001% of the total search space.
- On a machine with T cores, T priority Queues are initialized.
- The explored nodes are then distributed to the Priority Queues of T threads using the Round Robin partitioning strategy.
- Each thread then conducts A* search using its individual priority queue and the entire operation terminates either when one of the threads finds the end or when all threads have explored the entire search space.

4. RESULTS

The following measurements were recorded for the final parallel A* implementation:

Table 2: Time Taken for the parallel A* implementation

Number of Threads	Time Taken (seconds) for 10^6 nodes	Time Taken (seconds) for 10^7 nodes	Time Taken (seconds) for 10^8 nodes
Sequential	0.0060	5.0900	0.8260
2	0.0009	3.5000	0.0060
3	0.0009	2.6800	0.0060
4	0.0011	2.1000	0.0062
5	0.0012	1.9800	0.0070
6	0.0013	1.8000	0.0070
7	0.0014	1.7500	0.0072
8	0.0012	1.6200	0.0072
9	0.0014	2.0500	0.0257
10	0.0015	2.3000	0.0680

Table 3: Speedup for the parallel A* implementation

Number of Threads	Speedup for 10^6 nodes	Speedup for 10^7 nodes	Speedup for 10^8 nodes
Sequential	1	1	1
2	6.6667	1.4542	137.6667
3	6.6667	1.8992	137.6667
4	5.4545	2.4238	133.2258
5	5.1326	2.5707	118.0000
6	4.6511	2.8277	118.0000
7	4.4444	2.9085	116.3380
8	5.0977	3.1420	114.7222
9	4.2105	2.4829	32.1400
10	4.0000	2.2130	12.1470

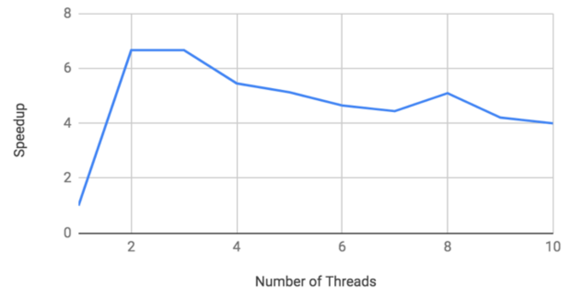

Fig. 5: Speedup vs Number of Threads (10^6 nodes search space)

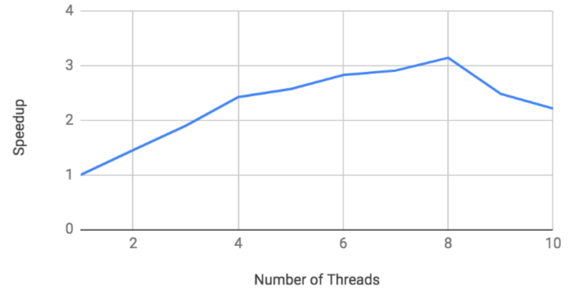

Fig. 6: Speedup vs Number of Threads (10^7 nodes search space)

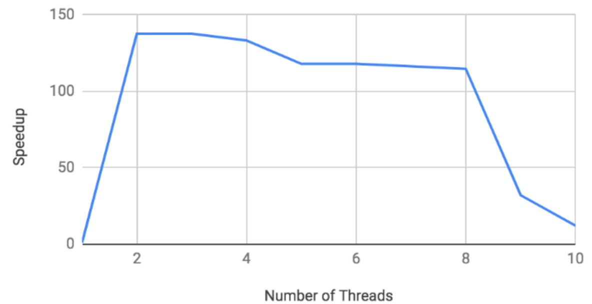

Fig. 7: Speedup vs Number of Threads (10^7 nodes search space)

4.1 Visualization of Searches

To analyze both searches a visualization heatmap was created. The legend indicates time in seconds.

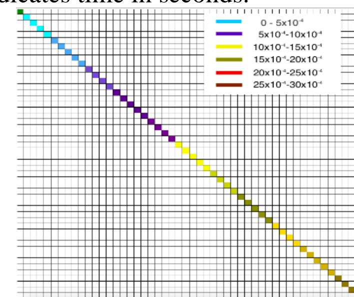

Fig. 8: A* Sequential on a 50x50 graph

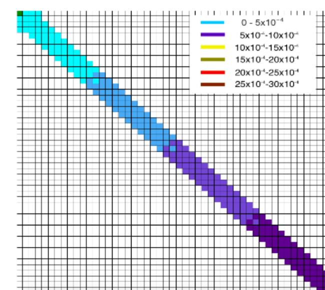

Fig. 9: A* Parallel on a 50x50 graph

4.2 Analysis

Speedup scaled as the size of the graph increased. Introducing parallel search for 10^8 nodes led to a speedup of 138x over the

© 2020, www.IJARIIT.com All Rights Reserved

sequential version while parallel A* for 10^6 nodes led to a 6x speedup. In the sequential version for each node visited, 8 of its neighbors need to be enqueued to the open list PQ and the heuristic needed to be calculated for each of them. Due to this, in the parallel search the search space for an individual thread is reduced by $1/8T$ (with T cores). The parallel version provides each thread with an assigned search space which leads to more overall nodes being visited (as seen in the visualizations in section 4.1). This decreases the number of neighbors that need to be enqueued per thread and thus less heuristics calculated per thread.

For 10^6 nodes and 10^8 nodes the best performance was observed with 2 threads and then linearly decreased as we added more threads (It is important note that there was still a significant amount of speedup over the sequential version when using 8 threads, it was just not the best performing). Since the limit of sequential A* depends on the number of threads, using more threads increases the number of nodes visited. In the case of search spaces with 10^6 nodes and 10^8 nodes, 2 threads provided the best balance between splitting up the search space while ensuring that the total number of nodes visited remained close to the size of the graph. Chart-10 illustrates how an increase in the number of threads impacted the total number of nodes visited for a graph with 10^6 nodes.

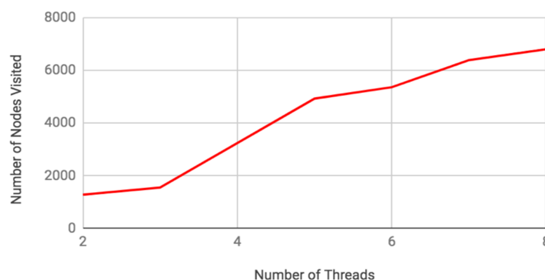

Fig. 10: Number of Nodes Visited vs. Number of Threads

5. CONCLUSION

This paper proposes a parallel A* search algorithm implementation that can be run on a multi-core CPU. The experiments demonstrated that CPU based parallel algorithm can have a considerable amount of speedup without using the computing resources of a GPU. This algorithm can significantly accelerate various search and computation tasks on a wide range of machines. An analysis of the current parallel implementation with 3 main phases - depth limited sequential A*, distributing work between each thread and each thread executing A* in parallel – found the following data.

Phase	% of total time spent
Limited A*	1.27%
Work Distribution	0.03%
Parallel A* by each thread	98.7%

On further investigation, the process of marking a node as visited within parallel A* search took the largest amount of time in parallel searches. This was due to contention amongst the threads to mark the node as visited. In the future, this phase can further be optimized and improved to get better speedup.

6. ACKNOWLEDGEMENT

This research paper is made possible through help and guidance from the School of Computer Science at Carnegie Mellon University and the professors for "Parallel Computer Architecture and Programming" – Prof. Bryant and Prof. Beckman. Two anonymous reviewers also provided insightful comments and feedback.

6. REFERENCES

[1]. S. Hong, T. Oguntebi and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," 2011 International Conference on Parallel Architectures and Compilation Techniques, Galveston, TX, 2011, pp. 78-88.

[2]. Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11). Association for Computing Machinery, New York, NY, USA, Article 65, 1–12. DOI:https://doi.org/10.1145/2063384.2063471

[3]. Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An effective GPU implementation of breadth-first search. In Proceedings of the 47th Design Automation Conference (DAC '10). Association for Computing Machinery, New York, NY, USA, 52–55. DOI:https://doi.org/10.1145/1837274.1837289

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

[5]. A. Lumsdaine, D. Gregor, B. Hendrickson, J. Berry, and J. Guest Editors, "Challenges in parallel graph processing," Parallel Processing Letters, vol. 17, no. 1, pp. 5–20, 2007.

[6]. Yichao Zhou and Jianyang Zeng. 2015. Massively parallel a* search on a GPU. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI'15). AAAI Press, 1248–1254.

[7]. Korf, R.E. Depth-limited search for real-time problem solving. Real-Time Syst 2, 7–24 (1990). https://doi.org/10.1007/BF01840464

Appendix A

Round Robin										
Time Taken (Min of 3 runs)= 0.00132										
PQ Number	0	1	2	3	4	5	6	7	Mean	Standard Deviation
Total Priority	25487430	25491286	25491294	25489321	25493257	25493281	25493288	25493344	25491562.63	2211.208069
Num Elements	13	13	13	13	13	13	13	13	13	0
Greedy										
Time Taken (Min of 3 runs) = 0.00159										
PQ Number	0	1	2	3	4	5	6	7	Mean	Standard Deviation
Total Priority	25487430	25491303	25491301	25487346	25493278	25497286	25493288	25491269	25491562.63	3246.031332
Num Elements	13	13	13	13	13	13	13	13	13	0
Interval										
Time Taken (Min of 3 runs) = 0.0019										
PQ Number	0	1	2	3	4	5	6	7	Mean	Standard Deviation
Total Priority	25108049	25243814	25370057	25512467	25641375	25776574	25874213	25405952	25491562.63	262158.1888
Num Elements	13	13	13	13	13	13	13	13	13	0

© 2020, www.IJARIIT.com All Rights Reserved