Parallel A* algorithm

Yu-Chi Ho
Computer Science
National Yang Ming Chiao Tung University
Taiwan

Yu-Sheng Chen
Computer Science
National Yang Ming Chiao Tung University
Taiwan

Xu-Rong Peng
Computer Science
National Yang Ming Chiao Tung University
Taiwan

ABSTRACT

A* is an informed path-finding algorithm which utilizes heuristic function to find the optimal-cost path in graphs. In order to fully utilize the power of CPU and to boost the performance of the search algorithm, we implemented parallel A* with shared memory and distributed memory model. A detailed analysis of the two models is presented with respect to methodology and performance.

1 INTRODUCTION

Graph and its applications have a significant role in the field of computer science. It is fundamental for applications such as social networks, routing problems, GPS navigation, and more. There are numerous path finding algorithms which have their own specific applications including BFS, DFS, UCS, A* and more. These algorithms can be divided into uninformed and informed algorithms. We decided to dig into the informed algorithms where A* search lies at. The algorithm benefits from the heuristic function which guides the computer with the paths that are more promising.

A* can be seen as the best algorithm for path finding in general. Additionally, with abundant computational resources we can get from modern machines, we decided to further improve the performance of A* by implementing the algorithm in a parallel fashion.

2 PROPOSED SOLUTION

In this section, we will explain how we implemented the parallel A* in shared memory and distributed memory model.

2.1 Centralized Parallel A*

Centralized parallel A* algorithm is a shared memory style algorithm which is easy to implement. Similar to serial A* algorithm, we use OPEN set to store the nodes which have been explored but not yet been expanded. In addition, we use CLOSED set to store the nodes which have been explored and expanded. The two sets are shared among all threads. All threads will try to get the best node in the OPEN set. The node **n** with the smallest **g(n) + h(n)** is what we mean the best, where **g(n)** is the cost from start node to the end node and **h(n)** is the heuristic of node **n**. The OPEN set will be locked when a thread accesses it in order to prevent multiple threads from getting the same node. When a thread chooses the node **n** to explore, **n** will be added to the CLOSED set. If node **n** is the end node, we will check if **g(n)** is less than the current best cost. The current best cost will be updated if **g(n)** is better, otherwise, the thread will be terminated. Thread then expands node n's neighbor as children. This part is identical to serial A*, so we omit it (Line 18-32 in Figure 2).

As figure 3 shows, when two threads expand the same children simultaneously, it may cause the longer path to become the current best path, which is not the best answer, and shorter path will never become the current best path since the node is closed. Thus, we lock the node when it is being expanded to ensure the correctness of the implementation.

Figure 1: Serial A*

Algorithm 1: A*

```
1  Initialize OPEN to {s_0};
2  while OPEN ≠ ∅ do
3      Get and remove from OPEN a node n with a smallest f(n);
4      Add n to CLOSED;
5      if n is a goal node then
6          Return solution path from s_0 to n;
7      for every successor n' of n do
8          g_1 = g(n) + c(n, n');
9          if n' ∈ CLOSED then
10             if g_1 < g(n') then
11                 Remove n' from CLOSED and add it to OPEN;
12             else
13                 Continue;
14         else
15             if n' ∉ OPEN then
16                 Add n' to OPEN;
17             else if g_1 ≥ g(n') then
18                 Continue;
19         Set g(n') = g_1;
20         Set f(n') = g(n') + h(n');
21         Set parent(n') = n;
22 Return failure (no path exists);
```

Figure 2: Centralized Parallel A*

Algorithm 2: Simple Parallel A* (SPA*)

```
1  Initialize OPEN_shared to {s_0};
2  Initialize Lock l_o, l_i;
3  Initialize incumbent.cost = ∞;
4  In parallel, on each thread, execute 5-32;
5  while TerminateDetection() do
6      if OPEN_shared = ∅ or Smallest f(n) value of n ∈ OPEN_shared ≥ incumbent.cost then
7          Continue;
8      AcquireLock(l_o);
9      Get and remove from OPEN_shared a node n with a smallest f(n);
10     ReleaseLock(l_o);
11     Add n to CLOSED_shared ;
12     if n is a goal node then
13         AcquireLock(l_i);
14         if path cost from s_0 to n < incumbent.cost then
15             incumbent = path from s_0 to n;
16             incumbent.cost = path cost from s_0 to n;
17         ReleaseLock(l_i);
18     for every successor n' of n do
19         g_1 = g(n) + c(n, n');
20         if n' ∈ CLOSED_shared then
21             if g_1 < g(n') then
22                 Remove n' from CLOSED_shared and add it to OPEN_shared ;
23             else
24                 Continue;
25         else
26             if n' ∉ OPEN_shared then
27                 Add n' to OPEN_shared ;
28             else if g_1 ≥ g(n') then
29                 Continue;
30         Set g(n') = g_1;
31         Set f(n') = g(n') + h(n');
32         Set parent(n') = n;
33 if incumbent.cost = ∞ then
34     Return failure (no path exists);
35 else
36     Return solution path from s_0 to n;
```

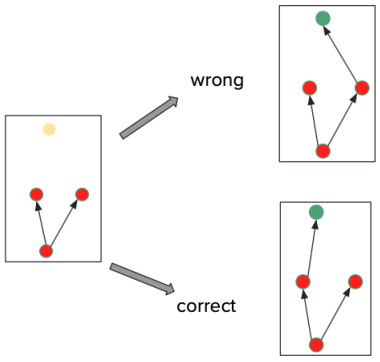

Figure 3: Case that could result in a wrong path

2.2 Decentralized Parallel A*

Decentralized parallel A* has a similar strategy as centralized parallel A* except the OPEN set and CLOSED set are distributed to each of the threads. They are private members to their own threads. In the distributed memory model, we can eliminate the lock when threads are choosing the specific node to explore. Because each thread has their own private OPEN set, they can explore the set simultaneously. During the exploration, we need to distribute the explored node to one specific thread. We use a random number generator to help us decide which thread should be responsible for the state. All threads do the above jobs iteratively until the end node is found.

However, the program is correct when thread number is one. This is due to the reason that the real world map is bidirectional. Other threads may modify the correct paths previously stored, resulting in an oscillation from the final result.

3 EXPERIMENTAL METHODOLOGY

3.1 Graph Construction

The dataset is from OpenStreetMap, where we obtain the map information of Hsinchu City. The raw data is then processed with OSMnx, a package from github that can retrieve the data of nodes and edges.

3.2 Heuristic

Heuristic is the soul of A*. We chose the straight line distance to target node as the heuristic of each node, which is admissible and straightforward.

3.3 Environment

Table 1 shows the computation environment. We use pthread to realize the parallel A*.

CPU	i5- 8257U 4 Core 8 Thread
OS	OSX 12.0.1
Compiler	clang 11.0.0
Parallel method	pthread

Table 1: Environment

4 EXPERIMENTAL RESULTS

The correctness of the program should be our first consideration. Therefore, before considering the performance of the program, we compare the result we get from the parallel A* search to the serial version, so that we can make sure the result and our implementation of the parallel version is correct. (Notice that the number of visited node may be different due to multi-threading, but the result should be the same.)

Figure 4 and 5 shows the experimental result of centralized parallel A* search without indexing. As shown in the figures, when the thread number increases, the execution time significantly decreases. Since we run our program on a machine with 4 cores 8 threads, when the thread number is greater than 8, we cannot get better performance. Moreover, the performance may be even worse.

Centralized parallel A*

Number of thread	Total execution time	A* algorithm time	visited node	Speed up	Amdahl's law Parallelizable proportion
1	11.433	10.347	7074	1.0	
2	5.896	5.252	7080	1.970	0.985
3	4.228	3.516	7093	2.943	0.990
4	3.399	2.747	7104	3.767	0.979
5	2.928	2.496	7118	4.145	0.948
6	2.804	2.179	7130	4.749	0.947
7	2.579	2.103	7135	4.920	0.930
8	2.472	2.077	7137	4.982	0.913
9	2.601	2.088	7140		
10	2.484	1.966	7147		
11	2.555	1.987	7151		

Figure 4: Result without indexing

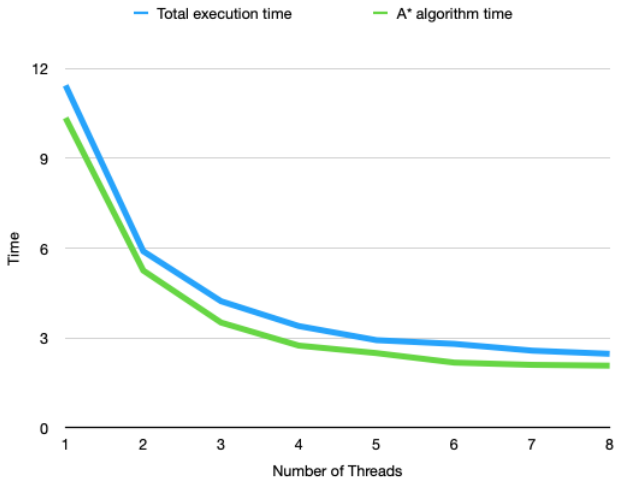

Figure 5: Time without indexing

When we did more detailed measurement on our program, we found that expanding new nodes costs most of the time since we have to do a linear search on both edge table and node table. So we came up with a new version which implemented the two tables by

map from C++ standard library. The experiment result is shown in figure 6 and 7. We expect that when the thread number is less than 8, we can get better performance when we increase thread number. However, the performance get worse when the number of thread is greater than 6. We think the reason is that the time cost by A* algorithm is really little , so the overhead of pthread operations dominate the performance. Also, we can observe the decreasing rate of the execution time is much more less than the previous version.

Centralized parallel A* with index

Number of thread	Total execution time	A* algorithm time	Visited node	Speed up	Amdahl's law Parallelizable proportion
1	1.123	0.458	7074	1.000	
2	1.079	0.394	7075	1.162	0.279
3	1.002	0.347	7075	1.320	0.364
4	1.002	0.325	7074	1.409	0.387
5	0.989	0.315	7075	1.454	0.390
6	1.043	0.366	7075	1.251	0.241
7	1.025	0.359	7076	1.276	0.252
8	1.105	0.428	7075	1.070	0.075
9	2.601	2.088	7077		
10	2.484	1.966	7077		
11	2.555	1.987	7078		

Figure 6: Result with indexing

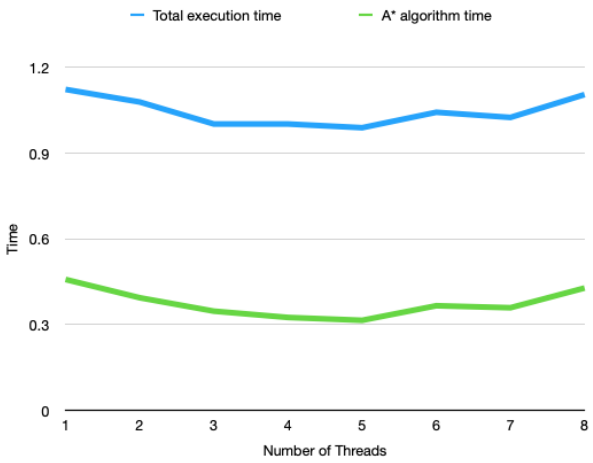

Figure 7: Time with indexing

We can also apply Amdahl's law to the above two implementation, and try to estimate the proportion of parallelizable region of both version. Our calculation shows that the parallelizable proportion of centralized A* search without indexing is much more than that of the version with indexing. This is mainly because of the difference of the time cost by expanding nodes. The result also perfectly explains why the decreasing rate of the execution time of centralized parallel A* search with indexing is much more less than the version without indexing.

5 RELATED WORK

There are lots of research projects about parallel A* search.

For the applications which cost most time at computing heuristic functions, there would be more focus on parallel computation of heuristic functions. The computation of heuristic functions of every node is independent and can be done in the same time, which is highly suitable for parallelism.

Maintaining explored set also plays an important role in such search algorithms, so there are also a lot of works [1] focusing on node duplication detection and explored set operations.

For the applications which cost most time at priority queue operations, there would be more focus on parallel priority queues. From this point of view, many methods [2] [3] are based on hardware support (such as GPU) to perform extraction of multiple nodes from the priority queue at the same time.

6 CONCLUSIONS

A* search algorithm is highly used in many fields. Although we are not the first one to do the topic about parallel A* search, we successfully realize parallel A* search which can really have better performance than the serial version in our project. Also, the comparison between the two version shows us that the more expensive the node expansion cost, the more efficiency the program can obtain.

As for future work, one possible topic is to apply decentralized parallel A* search on bidirectional graph. Another interesting topic is visualization. If we can do some great visualization of the process of parallel A* search, we can observe the process more clearly and easily. By doing so, we may get some new ideas about what may go wrong or which part can be improved.

7 REFERENCES

[1] Ariana Weinstock and Rachel Holladay. Parallel A* graph search.
[2] Yichao Zhou and Jianyang Zeng. Massively Parallel A* Search on a GPU.
[3] Nickson Joram. Parallel A* Search on GPU. https://medium.com/analytics-vidhya/parallel-a-search-on-gpu-ceb3bfe2cf51