Parallel algorithms for A*

Onur Cankur, Minghui Liu and Siddharth Singh

Group 6

Background

The A* search algorithm was published by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute in 1968, for robotics path finding. The A* search algorithm is an informed search algorithm that finds the lowest cost path from a start vertex to a goal vertex in a weighted graph. It extends Dijkstra's algorithm by using a goal-oriented heuristics function to speed up goal vertex detection.

Heuristics

A heuristic function is a function that estimates the distance between a vertex and the goal vertex. A heuristic function h is said to be admissible iff it never overestimates the distance to the goal, $h(v) \leq c(v, d)$ for all vertex v. A heuristic function h is consistent iff it follows the triangle inequality, $h(u) \leq w(u,v) + h(v)$, where $w(u, v)$ is the weight of the edge from vertex u to v.

Sequential A*

In the sequential A*, we maintain an open set of vertices and a closed set of vertices. Initially only the starting vertex is in the open set. In each iteration of the algorithm, the vertex n with the smallest $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach node n, and $h(n)$ is the heuristics of n (the estimated cost to go from n to the goal) is popped from the open set and expanded. Expansion is the process of looking at each neighbour m of node n, if m is never visited, or if the $g(m)$ is lower than what we have seen before, m is added into the open set.
Once the goal is popped from the open set, the algorithm terminates. By the admissible and consistent property of the heuristic function, we are guaranteed that the cost to the goal is the lowest.

The pseudo code for sequential A* is shown in figure 1 below.

The goal node is denoted by node_goal and the source node is denoted by node_start

We maintain two lists: **OPEN** and **CLOSE**:

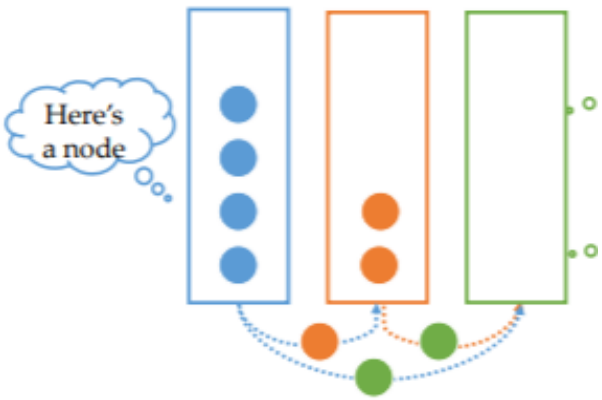
OPEN consists on nodes that have been visited but not expanded (meaning that sucessors have not been explored yet). This is the list of pending tasks.

CLOSE consists on nodes that have been visited *and* expanded (sucessors have been explored already and included in the open list, if this was the case).

```
1  Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
2  while the OPEN list is not empty {
3    Take from the open list the node node_current with the lowest
4         f(node_current) = g(node_current) + h(node_current)
5    if node_current is node_goal we have found the solution; break
6    Generate each state node_successor that come after node_current
7    for each node_successor of node_current {
8      Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9      if node_successor is in the OPEN list {
10       if g(node_successor) ≤ successor_current_cost continue (to line 20)
11     } else if node_successor is in the CLOSED list {
12       if g(node_successor) ≤ successor_current_cost continue (to line 20)
13       Move node_successor from the CLOSED list to the OPEN list
14     } else {
15       Add node_successor to the OPEN list
16       Set h(node_successor) to be the heuristic distance to node_goal
17    }
18     Set g(node_successor) = successor_current_cost
19     Set the parent of node_successor to node_current
20   }
21   Add node_current to the CLOSED list
22 }
23 if(node_current != node_goal) exit with error (the OPEN list is empty)
```

Hash Distributed A*

One method of parallelizing A* search is to use a hash function to statically distribute vertices to multiple processes. Each process maintains its own open set. When expanding a vertex n, a process uses the hash function to determine the owner of a neighbour vertex m, and send it, along with g(m), to the owner using a message. The receiver then determines whether to add m to its open set.



However using HDA* we lose the nice property that once a goal vertex is seen then the algorithm can terminate. Even if we use an admissible and consistent heuristic function, we are not guaranteed that the first time we see the goal vertex we have found the lowest cost path.

In HDA* algorithm, termination is complicated and is its own research topic. In this project we implemented two termination conditions, the barrier termination method outlined in [1] and a modified barrier termination method outlined by combining [1] and [2].

We start with describing the Barrier Method. In the barrier method, a process broadcasts the current path length (approx_path_to_dest) to all other processes as soon as the goal vertex is reached for the first time. The receiving processes mark a boolean flag (dst_found) as true and enters into an asynchronous barrier if its open set is empty. When in barrier mode, a process can continue to receive messages but stops processing the open set. When all processes have entered the barrier mode, the program terminates.

In the modified barrier method, the processes receiving the goal-found broadcast can also enter the barrier if the cost to reach the vertex on top of the open set is larger than the received cost (f(openSet.top()) > approx_path_to_dest). The algorithm terminates when all processes enter the barrier and either their open list is empty or their open list satisfies the aforementioned property.

Implementation

Charm++

In the Charm++ implementation we created a Main chare and an array of Worker chares. Each Worker chare has four entry methods:

1. hdastar() - perform 1 iteration of astar
2. receiveNode(float, float, int, int) - receive a node and add to open list
3. setDstFound(int) - receive the first detected path length to goal vertex
4. resumeFromBarrier() - come out of barrier and continue working

The Main chare has two entry methods:

1. done() - receive information that a worker has stopped
2. dstFound() - receive the first detected path length to goal and broadcast to all workers
3. reportIn(CkReductionMsg *msg) - receive a sum of all open set sizes and terminate if the sum zero

All synchronizations required for termination are performed using Main chare. If local conditions for checking termination are met at a worker chare, it notifies the main chare and stops computing. Once all workers have notified theMain chare, a reduce operation is initiated to check if all chares meet the local termination conditions.

MPI

The MPI implementation was more complicated compared to the Charm++ version. We had to implement a dynamic asynchronous messaging backend similar to that of Charm++, using

`MPI_IProbe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD)` and `MPI_Isend`.

This was inspired from the MPI based implementation of Charm++. We use `MPI_Ibcast` for broadcasting the first detection of goal vertex. The owner of destination issues broadcast only when it detects goal vertex. Other processes issue the same method call right at the beginning of an iteration to potentially receive the broadcast. We use `MPI_Allreduce` for calculating the termination condition. If the local condition for termination is met at a process, it contributes 0 to the all reduce. If the all reduce result is 0 then the program terminates, otherwise the program continues.

Experiment Setup

We performed all the experiments on the CPU nodes of the DeepThought2 cluster at University of Maryland. To speed up the process of submitting multiple jobs with different parameters and of repeating the experiments and calculate average times, we wrote custom shell scripts to aid us.

Graph Generation

For our experiments we created a graph generation tool that places vertices with randomly generated coordinates on a 2D square canvas. The tool can generate two types of graphs depending on how edges are generated.

1. K-Nearest Neighbour graph (kNN)
2. Unit Disk graph (rDisk)

A kNN graph connects each vertex with its K nearest neighbours. A lower K helps the performance of A* search as fewer neighbours need to be expanded. However if K is too low we risk breaking the connectivity of the graph and there might not be a path from the start vertex to the goal vertex. We empirically determined a suitable value of K to be 4.

A rDisk graph connects each vertex with all neighbours with a radius of r. To guarantee the connectedness of the graph, r needs to be set depending on the number of vertices.

$$\gamma^* = 2\left[\left(1+\frac{1}{d}\right)\left(\frac{\lambda_d(c)}{\xi_d}\right)\right]^{\frac{1}{d}}$$

$$r(n) = \gamma^*\left(\frac{log(n)}{n}\right)^{\frac{1}{d}}$$

d is the dimension of the space, d = 2 for our case, λd(c) is the volume, ξd is the unit ball volume in d dimensions, and n is the number of vertices in the graph [1].

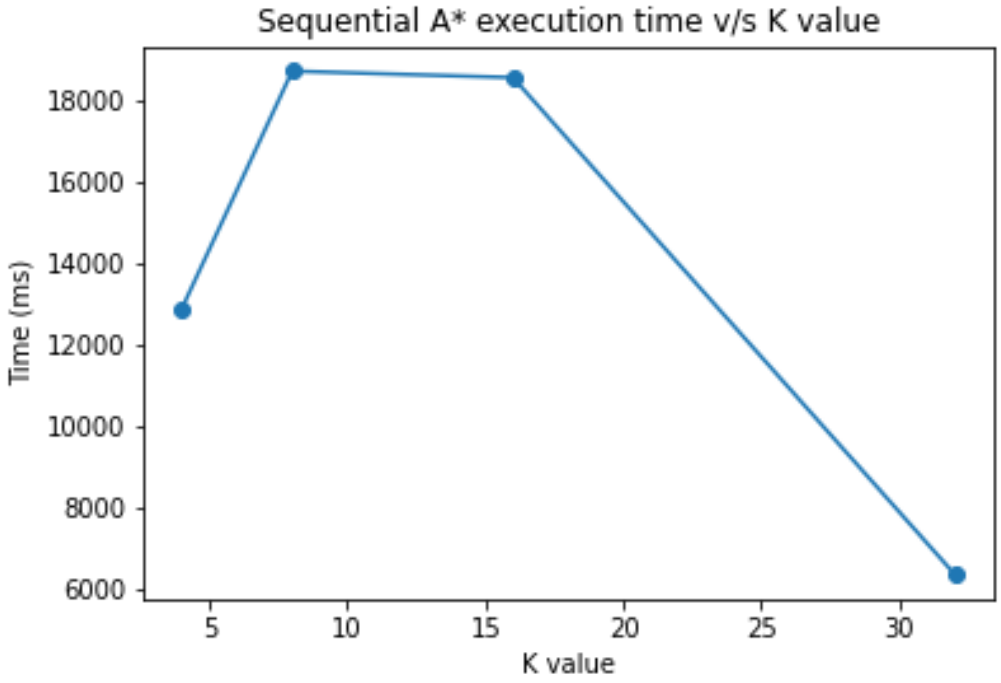
We set all edge weights to the Euclidean distance between two vertices. We also use the Euclidean distance between a vertex to the goal as the heuristic function, and it is both admissible and consistent.

While we wrote scripts to generate both types of graphs, we ended up using just the knn graphs for our experiments.

Experiments

I. Determining a suitable value of K

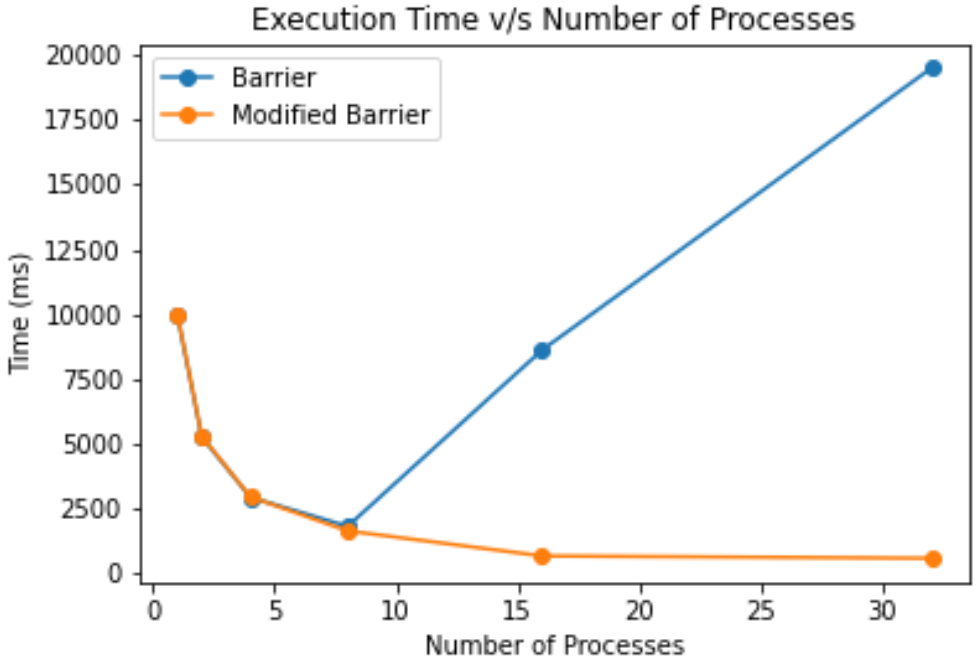
In this experiment we tried to see the effect of different K on sequential A* performance. We profiled the sequential A* implementation using KNN graphs of 1 million vertices but with varying K values: 4, 8, 16, 32.



The result shows that a K between 8 and 16 is suitable for parallelization.

II. Comparison of termination algorithms

We profiled our MPI implementation using a kNN graph of 1 million vertices and set K=4 and vary the number of processes.

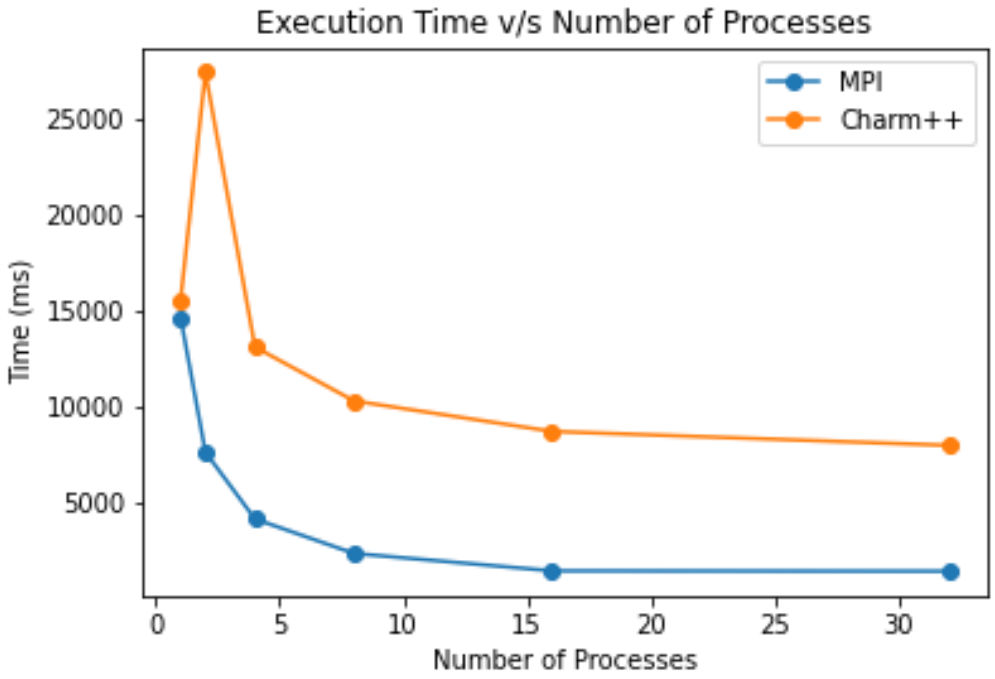


At low process counts (<= 8) the Barrier method and the Modified Barrier method performed similarly. But at high process counts, the Modified Barrier performed significantly better than the Barrier method and continues to scale. We conclude that the performance of HDA* is very sensitive to termination algorithms at high process counts.

III. Comparison of two implementations

In this experiment we compare our Charm++ implementation with our MPI implementation. We used kNNs graphs of 1 million vertices, and set K=8. We vary the number of processes used by the MPI implementation, and use the same numbers for the number of Worker chares of the Charm++ version.

The result shows that the MPI version performed consistently better than the Charm++ version. The Charm++ implementation's execution time shot up at 2 chares and then dropped. Just to make sure this was not the result of system noise, we repeated the 2 chare experiment multiple times and still found a similar result. We remember that students of CMSC818x also pointed out a similar problem in Fall 2020.

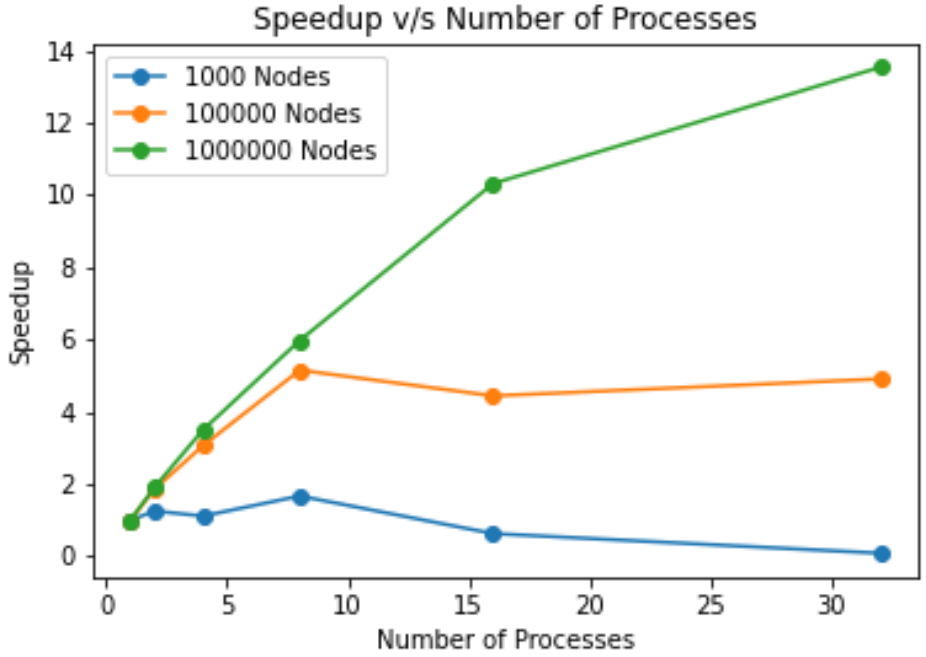


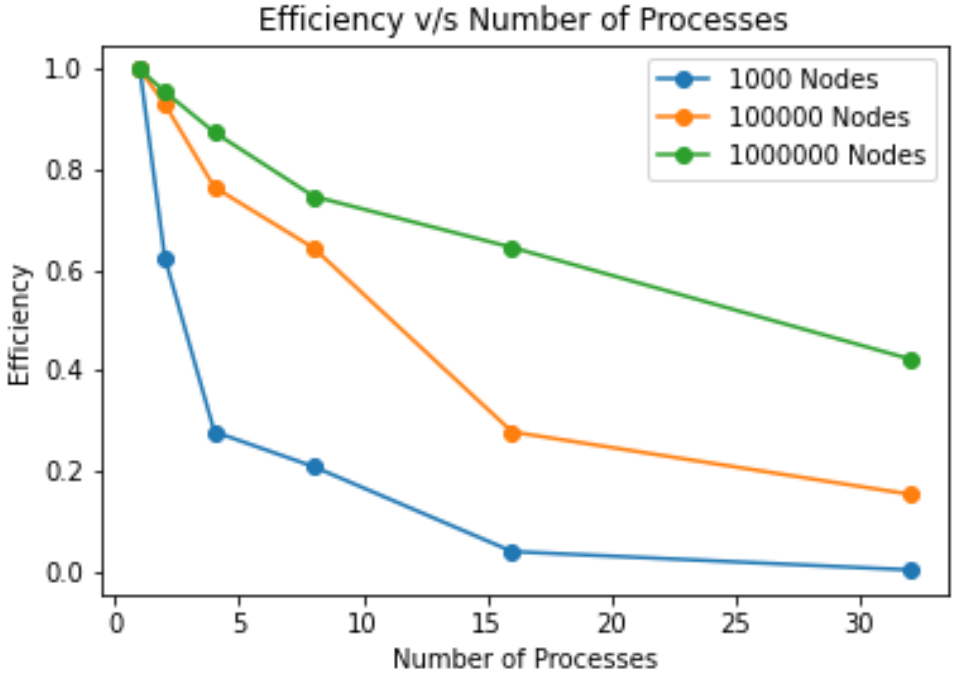
IV. Performance of MPI on various vertex counts

In this experiment we benchmarked the performance of the MPI implementation on various vertex counts. We used kNN graphs and set K=8 but used three different numbers of vertices: 1000, 100,000 and 1,000,000.

The higher vertice count achieved better speedup, efficiency and better scalability. Speedup for 1M nodes on 32 processes is 13.54 times.

It thus only makes sense to use parallel A* on really large graphs with millions of nodes.


Speedup v/s Number of Processes
1000 Nodes
100000 Nodes
1000000 Nodes
Speedup
Number of Processes
0
2
4
6
8
10
12
14
0
5
10
15
20
25
30


Efficiency v/s Number of Processes
1000 Nodes
100000 Nodes
1000000 Nodes
Efficiency
Number of Processes
0.0
0.2
0.4
0.6
0.8
1.0
0
5
10
15
20
25
30

Artifacts

The code of this project is available on Github at https://github.com/minghui-liu/parallel_astar. See the readme about how to run.

Bibliography

[1] Parallel A* Graph Search, Ariana Weinstock and Rachel Holladay, link
[2] Multi-Agent A* for Parallel and Distributed Systems, Raz Nissim and Ronen Brafman link

Contributions of Individual Group Members

1. Minghui
    - Writing graph generation algorithms (KNN and RDisc) using networkx
    - Contributed to the Charm++ implementation
    - Wrote major portions of the report
    - Implemented the sequential A*algorithm in Python
2. Siddharth
    - Wrote code for the MPI implementation - developed the dynamic async communication backend
    - Implemented the correct termination conditions for the algorithm in both MPI and Charm++
    - Implemented the sequential A*algorithm in C++
3. Onur
    - Contributed to the Charm++ implementation
    - Generated all visualizations and graphs for the report
    - Wrote bash scripts for conducting all experiments on deepthought2
    - Ran all experiments on deepthought2