Parallel A* Search

Yu Wu
yuwu3@andrew.cmu.edu

Yuqiao Hu
yuqiaohu@andrew.cmu.edu

***Abstract*—A* is a best-first graph search algorithm that computes the optimal paths in a graph. As the graph to search can be often extremely large in modern applications, it is important to have an efficient way to parallelize A* to leverage the multi-processing power of modern hardware. In this work, we study and implement two representative variants of the parallel A* algorithm: HDA* and GA*. HDA* uses hash function to distribute node expansion work across CPU processors in s distributed message passing model, while GA* employs the massive number of parallel cores on GPU. We evaluate our implementations on the path-finding problem, which is an important application of A* in robotics planning. We investigate the performance of our parallel A* and provide a detailed analysis and comparison of the work and communication overhead in experiments. Our HDA* implementation achieves a speedup as high as 25x on the 2D grid path-finding benchmark with 128 processes.**

1. Introduction

Graph search algorithms serve as fundamental tools across a broad spectrum of applications, providing solutions to complex problems in diverse fields such as artificial intelligence, computer science, and operations research. These algorithms navigate through interconnected nodes, representing relationships or dependencies within a system, to discover optimal paths or solutions. One such influential algorithm is the A*, which has proven its significance in a myriad of domains. Graph search algorithms, including A*, play a pivotal role in pathfinding, routing, and optimization problems, where the exploration of a graph's nodes is essential for determining the most efficient routes or solutions. A* stands out due to its guarantee of solution optimality and its ability to achieve high efficiency given an informed heuristic. To utilize the massively parallel computation ability of modern hardware and further improve the efficiency of A*, this project investigates and implements scalable algorithms that parallel A* on both CPU and GPU.

A* computes paths with minimum cost given start and goal nodes on weighted graphs. It is a greedy best-first search that always explores the most promising node it has discovered based on informed heuristics. Similar to the Dijkstra search, A* maintains an OPEN queue that contains nodes it has discovered so far. The nodes are prioritized by the sum of its cost and the heuristic, which represents an informed estimation of the cost from nodes to the goal. Like Dijkstra, it iterates over the queue until it reaches the goal state.

Since the OPEN queue is an inherently sequential structure and components in a general graph are often highly dependent on each other, parallelizing A* has been a challenging problem. However, there has been abundant work parallelizing A* with various methods and settings. Early attempts in parallel A* often adopt a multi-threaded approach on a single machine, and usually employ a shared memory model. However, in these methods, the number of cores parallel is often limited and the memory bottleneck can also prevent these methods from scaling to a larger scale. Hash-Distributed A* (HDA*) is a parallel A* variant that employs a distributed memory model and scales well on a distributed cluster with a relatively large number of cores. More recently, algorithms like GPU A* (GA*) leverage the massive number of cores on GPU to accelerate the node expansion.

In this project, we implemented HDA* in a messsage-passing model with OPEN_MPI platform and GA* on the CUDA platform. To investigate and compare the performance of these algorithms, we focus on the path-finding problem in robotics planning. We evaluate our implementations on the benchmark by ref and measure the performance speedup over the regular sequential A* baseline. We further provide a detailed analysis on the overhead and investigate the how various factors like the testing instance affect the performance speedup.

2. Literature Review

Parallelizing A* search has been an active research area among the AI community and there have been works focusing on various way to leverage multi-core hardware to speed up A*. Most parallel A* algorithms expand the nodes in the graph in parallel, and the common challenge to these algorithms is the large number of work overhead due to state re-expansion. For background information, A* is a best-first search algorithm, implying that it will only visit each state once, and when it visits or expands a state, it is guaranteed that A* find the optimal cost to that state once reaching it, given the condition that the heuristics is both consistent and admissible.

One of the earliest parallel A* search algorithms is Parallel Retraction A* (PRA*) [3]. In PRA*, a retraction technique is used to allow nodes with poor heuristic values

to be removed from the open list to maximize the utilization of SIMD Connection Machine (CM-2) for massive parallelism. Another approach that Parallel A* takes is PBNF [2], which uses abstraction to partition the state space and detect duplicate states with speculative expansions when necessary to prevent frequent locking.

The parallelism of A* search is further explored in Hash-Distributed A* [1]. In this approach, the algorithm distributes and schedules work among processors based on a hash function of the search state to effectively utilize the large amount of distributed memory for solving problems that require huge RAM to solve. However, the standard Zobrist Hashing used by HDA* incurs significant communication overhead despite its good load balance. In order to address this issue and improve the performance of HDA*, Abstract Zobrist hashing [4] is proposed, which reduces node transfers and mitigates communication overhead by using feature projection functions.

There are also parallel A* search algorithms that take advantage of GPU hardware. The GPU-based A* (GA*) [5] accelerates the search process of an agent with the help of GPU processors, which makes it a good fit for solving complex problems such as combinatorial optimization problems, pathfinding, and game solving. Another GPU-based search is DA* [6], which is based on the multi-GPU architecture. DA* also adopts effective graph partitioning and data communication strategies from GA* to make the most of this multi-GPU architecture.

There are also some latest parallel A* search approaches that exploit parallelism from a novel perspective. The Parallel A* for Slow Expansion (PA*SE) [7] parallelizes state expansions to get close to a linear speedup while preserving the completeness and optimality of A*. Another related approach is Edge-Based Parallel A* for Slow Evaluation (ePA*SE) [9], which improves on PA*SE by parallelizing edge evaluations instead of state expansions. This makes ePA*SE more efficient in domains where edge evaluations are expensive and need varying amounts of computational effort, which is often the case in robotics.

3. A* Algorithms and Parallelism

In this section, we will first describe the traditional A* search, which utilizes a sequential approach. Then we will focus on how we exploit parallelism with HDA* on OPEN_MPI platform and GA* on CUDA platform.

3.1. Traditional A* Search

Traditional implementations of A* search usually use two lists to store the states during its expansion, i.e., the open list and the closed list. The closed list stores all the visited states, and is used to prevent unnecessary repeated expansion of the same state. This list is often implemented by a linked hash table to detect the duplicated nodes. The open list normally stores the states whose successors have not been fully explored yet. The open list employs a priority queue, which is typically implemented by a binary heap. States in the open list are sorted according to a heuristic function f. For each node n, $f(n)$ is given by

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

where the function $g(n)$ calculates the distance or cost from the start node to the current node n, and the function $h(n)$ estimates the distance or cost from the current node n to the target node. We refer to the value of function f as f value and compare the f values of different paths to determine the optimal path.

In each round of A* search, we extract the node with the minimum f value from the open list, expand its outer neighbors and check for duplication. After that, we calculate the heuristic functions of the resulting nodes and then push them back to the open list. If there are nodes that have already been stored in the open list, we only update their f values for the open list.

In A* search, we require the heuristic function to be *admissible* for optimality, i.e., h(x) is never greater than the actual cost or distance to the end node. In our sequential codebase, we calculate the Manhattan distance from the current node n to the target node as our function $h(n)$.

3.2. HDA* Search

The Hash-Distributed A* (HDA*) search combines the hash-based work distribution strategy of PRA* and the asynchronous communications of Transposition-table driven work scheduling (TDS)[10] . In HDA* the closed and open lists are implemented as a distributed data structure, where each processor is assigned a partition of the entire search space.

Different processors communicate with each other using *Message Passing Model*. In particular, the closed and open lists of separate MPI processes share data with each other via non-blocking send and receive operations. Our implementation of HDA* takes advantage of MPI_Isend, MPI_IProbe and MPI_Ibcast for non-blocking communication. This enables HDA* to pack multiple states together to reduce the overall communication overhead.

In contrast to traditional A* search implementations, where we terminate immediately when the target node is reached, there is no guarantee that the MPI process that first reaches the target node has the optimal path. Therefore, when a processor discovers a valid path, it will broadcast the cost of this path to all other processors. Then it will wait until all processors have proven that there is no solution with a cost better than this path. In order to terminate correctly, it is not sufficient to check the local open list in every processor. We also need to ensure that there is no work currently on the way to arrive at a processor.

3.3. GA* Search

In order to address the limitation of parallelism due to the outer degree of each node in the search graph,

Algorithm 1 HDA*(start, goal)

```
1: if HASH(start) = pid then
2:     open_list.PUSH(start)
3:     hash_table.PUSH(start)
4: while True do
5:     if not open_list.empty() then
6:         curr ← open_list.POP()
7:         if curr = goal then
8:             found ← true
9:             continue
10:        neighbors ← EXPAND(curr)
11:        for node in neighbors do
12:            if HASH(node) = pid then
13:                add_to_local_open_list()
14:            else
15:                message_set[HASH(node)].PUSH(Msg(node))
16:    else
17:        if found = false then
18:            MPI_Test(req, flag, status)
19:            if flag = true then found ← true
20:        else
21:            MPI_Allreduce(&to_send, &to_recv, 1, ...)
22:            if (to_recv = 0) then return path found
23:    send_message_set()
24:    recv_message_set()
25:    add_to_remote_open_list()
26: clear_open_list()
27: clear_hash_table()
```

we attempt GA* to further exploit parallelism with GPU hardware. In GA*, instead of using one single priority queue for the open list, we allocate a large number of priority queues during A* search. Each time we extract multiple nodes from each priority queue, which thus parallelizes the sequential part in the original algorithm. Meanwhile, the GA* algorithm also increases the number of expanding nodes at each step, which further improves the degree of parallelism for the computation of heuristic functions.

For complex problems that have large search spaces, node duplication detection becomes a key to performance improvement. In our implementation, we utilize the parallel hashing with replacement algorithm. In this approach, we do not need to guarantee that all the duplicated nodes must be detected. In particular, when a new node need to occupy the position of an old node, instead of pushing the old node to a different position, parallel hashing with replacement simply drops the old node. This algorithm is simple, fast, and easy to be parallelized.

4. Experimental Evaluation

4.1. Implementation and Benchmark

We implement HDA* algorithm as presented in Section 3.2 using the OPEN MPI platform. We use the Manhattan

Algorithm 2 GA*(start, goal, k)

```
   Let {Q_i}_{i=1}^k be the priority queues of the open list
2: Let H be the closed list
   PUSH((Q_1, start)
4: bestNode ← nil
   while Q not empty do
6:     S ← [ ]
       for i ← 1 to k in parallel do
8:         if Q_i is empty then
               continue
10:        q_i ←EXTRACT(Q_i)
           if q_i = goal then
12:            if bestNode = nil or f(q_i) < f(bestNode)
   then
                   bestNode ← q_i
14:                continue
           S ← S+ EXPAND(q_i)
16:    if bestNode ≠ nil and f(bestNode) < min_{q∈Q} f(q)
   then return the path generated from bestNode
       T ← S
18:    for s ∈ S in parallel do
           if s ∈ H and H[s].g_val < s.g_val then
20:            remove s from T
       for t ∈ T in parallel do
22:        t.f_val ← f(t)
           Push t to one of the priority queues
24:        H[t] ← t
```

Algorithm 3 Hash-with-Replacement-Deduplicate(H, T)

```
   T' ← T
   for i ← 0 to |T| in parallel do
3:     z ← 0
       for j ← 0 to d − 1 do
           if H[h_j(T[i]) ∈ {T[i],nil} then
6:             z ← j
               break
       t ← T[i]
9:     atomic-swap(t, H[h_z(T[i])])
       if t = T[i] then
           remove T[i] from T'
12:        continue
       for j ← 0 to d − 1 do
           if j ≠ z and H[h_j(T[i])] = T[i] then
15:            remove T[i] from T'
               break
   return T'
```

distance as our heuristic. As it is closely studied in 4, the hash function determines the work distribution of HDA* and is critical to the performance. It directly influences the total number of nodes expanded (work overhead) and the total number of nodes that need to be communicated across the process (communication overhead). With respect to the choice of hash function, there is a tradeoff between work balance and less communication overhead. Intuitively, a purely random hash function has a perfect work balance, while having more locality implies neighbors of a node are more likely to be hashed to the same process, thus less communication overhead but potentially more unbalanced work. To investigate how hash function and locality affect performance, we test with 3 hash functions. In the context of pathfinding in grid world, we can easily represent each state with its location, which is a single integer. The random hash function is purely random without any locality, while the grid hash hashes each 8 by 8 grid of cells to the same random process id, showing locality in each grid. The modulo hash function is a naive and simple way and simply take modulo of the total number of process.

We evaluate our implementation of HDA* on the Bridges-2 cluster at Pittsburgh Supercomputing Center. We run our experiments on 2 AMD EPYC 7742 CPU, with a total of 128 cores and 256GB RAM. Due to time limitations, our GA* implementation is still in the debugging phase and is not evaluated experimentally. We focus on the path-finding problem under the settings of a 4-connected unit-length grid graph. The problem instances and start-goal location scenarios are from the 2D Pathfinding Benchmarks [8], which is a standard benchmark for path finding tasks. We select 13 maps with various structures and for each map, we test on 10 start-goal scenarios. The scenarios in [8]are sorted by difficulties, i.e. the cost of the optimal possible solution. To fully show the capability of HDA* leveraging parallelism, we use the 10 most difficult scenarios for each map. However, most of the maps from the benchmark have sizes up to 512 by 512 or 1024 by 1024. As we found in 4.2, such sizes are too small for the parallel search algorithm to fully utilize the parallel resources available. Therefore, we magnify both sides of each rectangle map by a factor of 4, obtaining new maps with sides of length 2048 or 4096. The total number of states in the magnified maps ranges from a few million to more than 10 million. The start-goal locations from the scenarios are equally interpolated to ensure correctness. We run our HDA* with $P = 2, 4, 8, 16, 32, 64, 128$ and compare their runtime speedup against the sequential A* baseline. We also keep track of the number of nodes expanded N and the number of nodes that are received from another process N_{comm}. We measure the work overhead as $\frac{N_{parallel}}{N_{seq}}$. Comparing these two metrics, we will hopefully learn about the effectiveness of each hash function in reducing work overhead and communication overhead.

4.2. Speedup in Runtime

We present the performance speedup of our HDA* over sequential A* on 6 out of 13 maps we tested in Figure 1. The full results are available in the project Git repository. We observe that both the grid and naive modulo hash function scale reasonably well and achieve a 15 to 25 times speedup at 128 cores. Surprisingly, while grid hash performs the best when the number of cores $\leq$ 64, the naive modulo hash function scales the best at 128 cores and outperforms grid hash in most cases. Random hash performs poorly and only achieves a 6 to 10 times speedup at 128 cores. This comparison shows that compared to the random hash, both grid and modulo hash effectively gain a significant performance boost from the locality, by trading some work distribution balancing. This is likely due to locality reducing the communication overhead and work overhead, and as we will see in subsequent sections, this is indeed the case. One thing to note is that since all numbers of process P and the size of the sides of the maps are exponents of 2, the naive modulo hash actually hashes all cells in the same column to the same processor, showing a clear pattern of locality.

The speedup performance of HDA* greatly depends on the size and the structure of the map instance. In 2, we show the performance of HDA* on the same Paris map that is scaled to different sizes. We observe that at the original 1024 by 1024 sizes, the speedup is capped at 10 and the algorithm struggles to scale its performance as P increases. This shows that HDA* can only fully leverage the parallel resources when the scale of the problem instance is reasonably large. We also observe that there is a non-trivial constant overhead in the MPI platform when initializing and finalizing the process. This constant overhead also explains the performance scales at larger problem instances, as the constant overhead contributes less to the overall runtime.

The speedup also depends on the structure of the graph instance. More specifically, HDA* performs better compared to the sequential baseline when there are more obstacles in the instance and the heuristics does *not* provide a good estimation for the actual cost. Take an extreme example, when our algorithm is run on an empty grid map with no obstacles, HDA* has a negative speedup over sequential A* of about 10 to 20 times. This is because, in an empty map, the Manhattan heuristic accurately predicts the cost from any node to the goal state, therefore A* will only expand nodes on the optimal path to the destination. Since HDA* requires expanding many more nodes than those on the optimal path, it is inevitable that HDA* incurs a huge work overhead in this case. In contrast, in instances like 64room_005 and maze2048-1-9, the optimal path is often many times longer than what the Manhattan heuristic predicts as the agent is required to circumvent many obstacles in the map to reach the destination. Thus, both A* and HDA* have to explore a large number of states that are not on the optimal path before reaching the goal. In this case, HDA* can fully leverage its extra computational resources to explore a large number of states.

4.3. Work and Communication Overhead

Work and Communication overheads are the main bottlenecks that prevent HDA* from achieving linear speed-up.

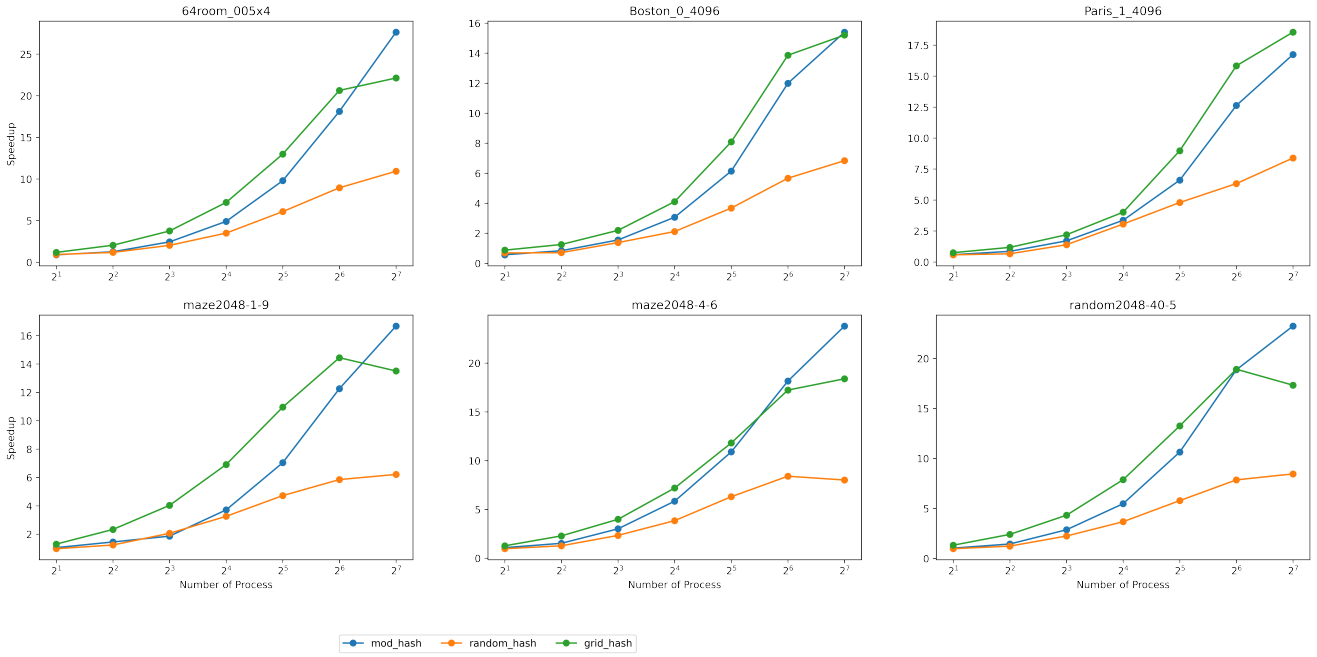

Figure 1. The speedup of HDA* across different graphs versus different hash functions

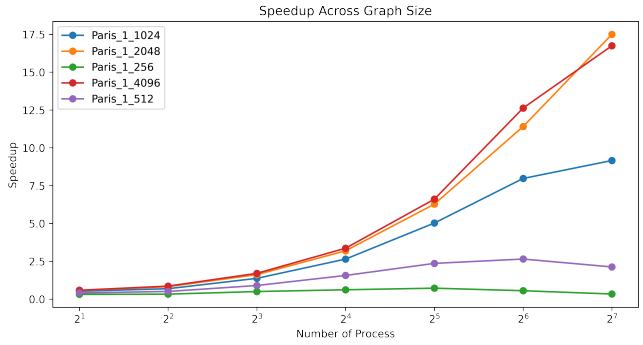

Figure 2. The speedup of HDA* across different size of graphs

We present the work and communication overheads in Figure 3, 4, and 5 on the random2048-40-5 map. similar results and analysis apply to all other maps we tested on. Since we have multiple OPEN queues and expand multiple nodes in parallel, we don't have the nice guarantees as in sequential A* that we only need to expand a node once. It is often required to re-expand nodes before we find the optimal cost to them. In addition, there are often cases where the number of useful node expansions is limited at the moment and some processors have to expand nodes that do not contribute to forwarding the search toward the goal. These two factors together constitute the extra nodes expanded by HDA*, which can be represented by the work overhead. We define work overhead $= \frac{N_{parallel}}{N_{seq}}$, as explained in 4.1. In Figure 3, we show that all three hash functions control the work overhead fairly well (within 1.4) when $P \leq 32$. However, when P goes beyond 32, the random hash's overhead rapidly increases and surpass 2.0, while the other two hash function still keeps the overhead within 1.5. How the hash function exactly influences the work overhead is not immediately clear, as the node expansion is a combined result of many unpredictable factors such as process scheduling. However, we hypothesize that locality also helps reduce work overhead, as when there is less communication, there would be less contention in sending and receiving nodes, and each node that will eventually be part of the optimal path can be sent faster to its owner processor. This hypothesis would require further experiments and research to be tested.

Now we turn our focus to the communication overhead C, which we simply measured as the total number of nodes that have been received during the search. In Figure 4, we clearly observe that $C_{random} > C_{module} > C_{grid}$. This observation agrees with our expectation that more locality implies less communication overhead since grid hash has locality over a 8×8 grid, modulo hash has locality over each column, and random hash has no locality. While the C keeps constant as the number of processes increases for grid and modulo hash, it surges at $P > 32$ for random hash, corresponding the the same surge in Figure 3. We further show the rate of time spent on MPI communication and the total runtime in Figure 5. The communication time can contribute to as much as 50% of the total time, which is surprisingly high. For the rate of communication time, random and modulo hash perform approximately the same, while the grid hash has significantly less time spent on communication, given that the runtime of grid hash is also smaller than the other two. Nevertheless, the communication still accounts for 25% of runtime using grid

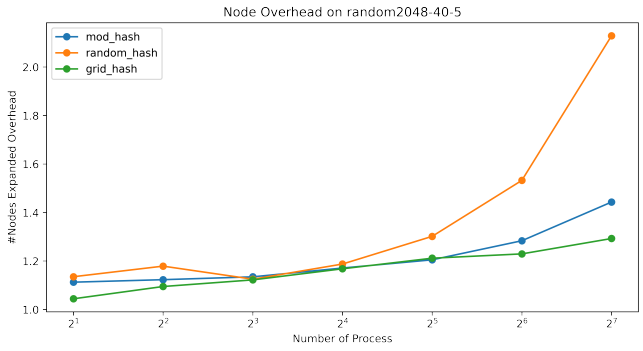

Figure 3. The work node overhead on random2048-40-5 versus different hash functions

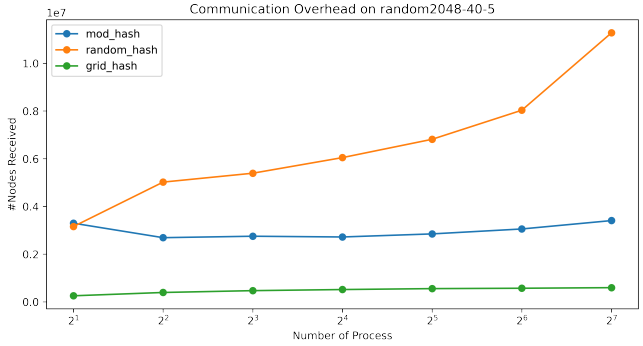

Figure 4. The communication overhead on random2048-40-5 versus different hash functions

hash when P is large. This further manifests the importance of optimizing communication overhead for parallel search algorithms in a message-passing model.

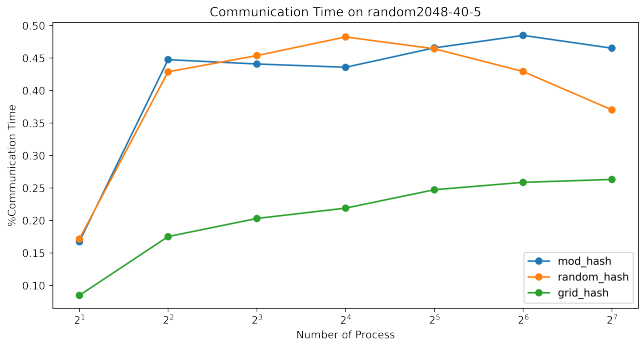

Figure 5. The communication time on random-2048-40-5 versus different hash functions

5. Conclusion and Future Work

In summary, HDA* exhibits good parallel performance in most cases. The speedup of HDA* heavily depends on the size and the structure of the map instance, as well as the structure of the graph instance. In particular, HDA* can fully utilize the parallel resources when the scale of the problem instance is reasonably large, and when there are more obstacles in the graph instance and the heuristics function fails to accurately estimate the actual cost.

The main bottlenecks that prevent HDA* from achieving linear speedup are work and communication overheads. We notice that the choice of hash functions can have a significant impact on the overall work overhead, though in which direction will a hash function influence the performance is still undetermined and may require further research. The locality displayed within a hash function also determines the communication overhead between different MPI processes, with more locality implying less communication overhead.

One notable limitation we faced in efficiently producing results was the challenge presented by the type of graphs on which we chose to test. We utilize random 2D pathfinding graphs as our benchmarks. While we believe that this is realistic for many use cases, it will not apply to all use cases and is also not difficult enough for specific problems like solving complex puzzles or designing proteins. Since the graphs we use for testing are generally too small in size for parallel search algorithms to fully exploit parallel resources, the speedup in HDA* is capped. Though we manage to magnify maps to increase their scale, it is still worth further testing our algorithm on more and harder graph search scenarios. To build upon our current analysis, We would like to obtain a larger body of results data and test our algorithm on other types of graphs aside from the current 2D grid graphs to see if our results hold across all graph types.

In addition to the hash functions we tested in our experiments, there are some more advanced hash functions proposed by experts in the parallel graph search area that may further improve the performance of HDA*. In the future, we would like to explore the standard Zobrist function[1], abstraction-based node assignment (AHDA*) [11], and the Abstract Zobrist function [4] and compare their parallel performance under different circumstances. We also hope to research on the rich area of distributed determination detection and compare across different detection methods.

Lastly, even though we did not have time to complete the experimental evaluation of our GA* algorithm, we managed to finish a functionally complete version based on our sequential code base and analyzed different parts of parallelism in code. We expect the distributed priority queues to outperform the parallel MPI processes in HDA* and we would like to further explore the comparison between HDA* and GA* with bigger sets of graph benchmarks.

Acknowledgments

We would like to thank the members of the 15-418 course staff for their guidance and support on our final project.

References

[1] A. Kishimoto, A. Fukunaga and A. Botea, ”Scalable, Parallel Best-First Search for Optimal Sequential Planning”, in *19th International Conference on Automated Planning and Scheduling, ICAPS 2009*.

[2] E. Burns, S. Lemons, W. Ruml and Z. Zhou, ”Best-First Heuristic Search for Multicore Machines”, in *Journal of Artificial Intelligence Research*, 2010, pp.689-743.

[3] M. Evett, J. Hendler, A. Mahanti, and D. Nau, ”PRA*: Massively Parallel Heuristic Search”, in *Univerisity of Maryland*.

[4] J. Yuu and A. Fukunaga, ”Abstract Zobrist Hashing: An Efficient Work Distribution Method for Parallel Best-First Search”, in *13th AAAI Conference on Artificial Intelligence, AAAI-16*.

[5] Y. Zhou and J. Zeng, ”Massively Parallel A* Search on a GPU”, in *29th AAAI Conference on Artificial Intelligence1248*, Beijing.

[6] M. Phillips, M. Likhachev, and S. Koenig, ”PA*SE: Parallel A* for Slow Expansions”, in *International Conference on Automated Planning and Scheduling, ICAPS 2014*.

[7] X. He, Y. Yao, X. He, Z. Chen, J. Sun, and H. Chen, ”Efficient parallel A* search on multi-GPU system”, in *Future Generation Computer Systems*, 3rd ed. vol 123, 2021, pp.35-37.

[8] N. Sturtevant, ”2D Pathfinding Benchmarks”, *https://movingai.com/benchmarks/grids.html*.

[9] S. Mukherjee, S. Aine, and M. Likhachev, ”ePA*SE: Edge-Based Parallel A* for Slow Evaluations”, in *The Robotics Institute, CMU*, 2023.

[10] J. Romein, A. Plaat, H. Bal, and J. Schaeffer, ”Transposition Table Driven Work Scheduling in Distributed Search”, in *Proceedings of AAAI-99*, 1999, 725–731.

[11] A. Burns, S. Lemons, W. Ruml, and R. Zhou, ”Best-first heuristic search for multicore machines”, in *Journal of Artificial Intelligence Research*, 2010, 39:689–743.

Work Credit

We divided work evenly, with approximately 50%-50%.

Yu Wu: Sequential Implementation, Driver, Debugging of HDA*, Debugging of GA*, Benchmarking and Testing, Final Report.

Yuqiao Hu: Sequential Implementation, Draft of HDA*, Debugging of HDA*, Draft of GA*, Debugging of GA*, Final Report.

Website URL

https://katrina0406.github.io/parallel_astar_search/

Github URL

https://github.com/Lucas-707/PAStar