More Robots and More Cores: Parallelizing the CBS Algorithm

Jaekyung Song
Robotics Institute
Carnegie Mellon University

Matthew Booker
Robotics Institute
Carnegie Mellon University

I. URL to Project Page

https://jsongcmu.github.io/parallel_CBS/

II. Summary

We took two approaches to parallelize a popular Multi-Agent Path Finding algorithm named CBS. Our low level approach showed speedups between 0.3 - 1.3x over a sequential method while our high level approach showed speedups up to 6.2x. Our analysis was performed on the GHC machines and showed that the nature of the problem was a large factor in the performance of our parallel method.

III. Background

A. Overview of MAPF

In the MAPF problem, we are given a graph G(V, E), and a set of N agents. Each agent has a start and goal position $s_i, g_i \in V$ and the task is to find a path through G for each agent from its start to its goal without conflicting with other agents while minimizing a cost function. At a given timestep, an agent may traverse an edge or remain at its current vertex. In our case, the cost function we aim to minimize is the makespan, which is the sum of the lengths of paths of all agents.

B. Conflict-based Search

Conflict-based Search (CBS) [1] is an algorithm used for solving the Multi-Agent Path Finding (MAPF) problem. The algorithm consists of two levels. At the bottom level, it utilizes the A* algorithm to find paths for individual agents, solving a Single-Agent Path Finding (SAPF) problem. At the top level, it creates a binary tree, known as the Constraint Tree (CT), that is used to select which collisions to resolve amongst the agents. The root of this tree contains an instance of the problem where all agents naively plan a shortest path to their goal. If no collisions occur in this instance, then the problem is solved. Otherwise, the first collision between any two agents is selected and two subtrees are created. One subtree contains the same problem instance but prevents the first agent from being at the collision location at the collision timestep, and the second subtree is identical but prevents the second agent from being at the collision location at the collision timestep. One important property of CBS is that it is guaranteed to find the optimal solution if it exists. Figure 1 gives a visualization of the Constraint Tree.

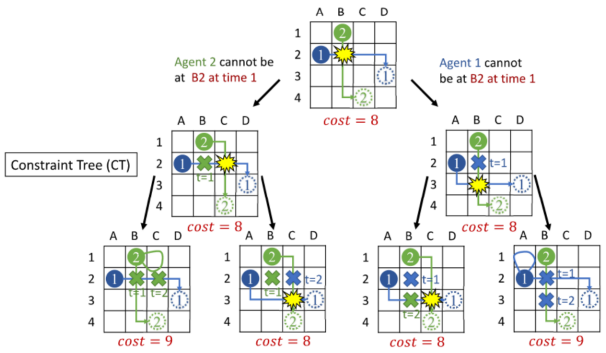


Fig. 1: A visualization of three levels of the Constraint Tree [2]

Although there is a distinct separation between the high level of CBS and its low level, A*, the two function very similarly. Both use priority queues as the data structure for the open list to ensure a best-first search. The algorithm for the high level is given in Algorithm 1 and for A* is given in Algorithm 2.

C. Hash Distributed A*

Hash Distributed A* (HDA*) [3] is an algorithm for parallelizing single agent path finding. In A*, only a single node is considered from the open list at a time, which prevents parallelization. In HDA*, each processor has its own open list, and expands it, computing the cost to get to that node from the starting position. If the node has been expanded before and is in the visited list, and if the new computed cost is lower, then the node in the visited list is updated; otherwise it is discarded and the processor grabs a new node from the open list. If the node is not discarded, then its neighbors are computed. The neighbors are then put through a hashing function to determine which processor should get them; these nodes are then added to the open list of those processors. This allows multiple nodes to be processed in parallel, and each processor generates work for every other processor. The algorithm is shown in Algorithm 3.

In A*, because only the most promising node in the open list is expanded per iteration, when a node is expanded, the shortest path to that node has been found. For this reason, when the goal node is expanded, we can terminate the search as the shortest path to the goal has been found. However,

Algorithm 1 High-level of CBS [1]

Input: MAPF Instance

$R.constraints \leftarrow \emptyset$
$R.solution \leftarrow$find individual paths using low-level()
$R.cost \leftarrow 0$
Insert R into Open
while Open not empty **do**
    $P \leftarrow$ node from Open with lowest cost
    Detect conflicts in P
    if P has no conflict **then**
        return $P.solution$
    end if
    $C \leftarrow$ first conflict (a_i, a_j, v, t) in P
    for each agent $a_i \in C$ **do**
        $A \leftarrow$ new node
        $A.constraints \leftarrow P.constraints + (a_i, s, t)$
        $A.solution \leftarrow P.solution$
        Update $A.solution$ by invoking low-level(a_i)
        $A.cost \leftarrow Cost(A.solution)$
        Insert A into Open
    end for
end while

Algorithm 2 A* Algorithm

Input: SAPF Instance

Open $\leftarrow \emptyset$
Visited $\leftarrow \emptyset$
$R.position \leftarrow start$
Insert R into Open
while Open not empty **do**
    $C \leftarrow$ node from Open with lowest f-value
    if $C.position == goal$ **then**
        return C
    end if
    $C.closed \leftarrow true$
    for each valid neighbor N of C **do**
        if N is in Visited **then**
            if $N.closed$ **then**
                continue
            end if
            if $C.g + cost(C, N) < N.g$ **then**
                $N.g \leftarrow C.g + cost(C, N)$
                $N.f \leftarrow N.g + N.h$
                $N.parent \leftarrow C$
                Insert N into Open
            end if
        else
            Insert N into Visited
            Insert N into Open
        end if
    end for
end while

in HDA*, multiple nodes are considered at once, so early termination is not possible. Termination can only occur when all open lists are empty. As a result, HDA* has more work it must do, as it expands significantly more nodes and explores more of the map than A*.

Algorithm 3 HDA* Algorithm [3]

Input: SAPF Instance

for each processor P **do**
    Open[P] $\leftarrow \emptyset$
end for
Visited $\leftarrow \emptyset$
$R.position \leftarrow start$
$H \leftarrow$ Hash(R)
Insert R into Open[H]
while all Open lists are not empty **do**
    for each processor P, parallel, no wait **do**
        $C \leftarrow$ node from Open[P] with lowest f-value
        if $C.pos$ is in Visited as C' **then**
            if $C.g < C'.g$ **then**
                $C'.g \leftarrow C.g$
                $C'.f \leftarrow C.f$
                $C'.parent \leftarrow C.parent$
            else
                continue
            end if
        else
            Insert C into Visited
        end if
        for each neighbor N of C **do**
            if N outside map or inside obstacle **then**
                continue
            end if
            $T \leftarrow$ hash(N)
            Insert N into Open[T]
        end for
    end for
end while

IV. Approach

In our approach, we chose to tackle the problem in two different ways. In the first, we approach the problem by tackling the low level search. Since the A* search takes the most amount of time, we felt that parallelizing it would help to improve runtime performance. In the second approach, we use the fact that the CT nodes are independent to parallelize the high-level search. In both cases, our implementation was written in C++ to leverage OpenMP. We targeted running on the GHC machines. Our code was written by us and from scratch. Although, we had both implemented the CBS algorithm as part of another class at CMU (16-891: Multi-robot Planning and Coordination) it was implemented in Python.

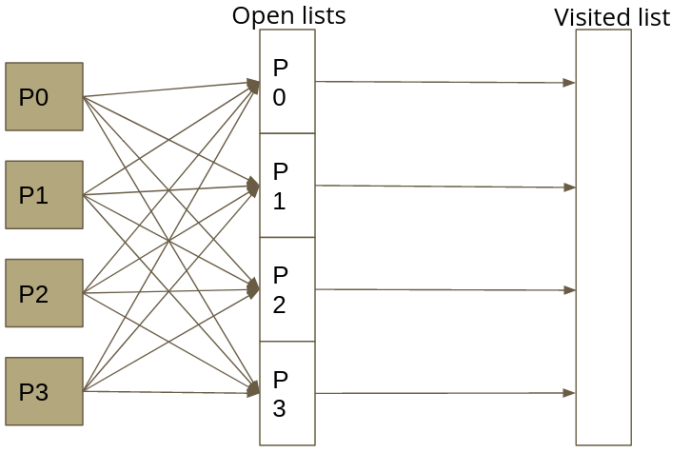


Fig. 2: Naive implementation of HDA*

A. Termination Criteria and Coordination

Before diving into the two separate approaches, it is important to discuss a core idea that is central to both. The high-level and low-level search's have a similar structure and both use priority queues for determining which node to expand next. As a result, a core idea in both algorithms is that the first time they encounter a node that satisfies the termination criteria (e.g. node is at the end goal for A* or node has no collisions for CBS) then they are guaranteed to have found the optimal solution. However, as soon as we begin to add parallelism this idea doesn't hold as we no longer have a strict ordering on which nodes are processed before one another. Consider the following example, the true optimal solution lies at node X. Processor A which is supposed to process node X is taking a while to finish processing previous nodes. Simultaneously, processor B is rapidly going through all its nodes and eventually finds another node Y that also satisfies the termination criteria. Node Y can either have equal or higher cost than X. Thus, taking the first conflict free solution results in potentially incorrect sub-optimal results. To remedy this, we first note that A* (and by extension CBS) only works on graphs whose edges are monotonically non-decreasing. Simply put, this means that any time a node is expanded its total cost must increase or stay the same compared to its parent, it will never decrease. Following from this observation, any valid solution found in parallel provides an upper bound on the optimal cost. As such, we can discard any nodes in the priority queues that have cost higher than the initial solution. The search continues with any node that satisfies the termination criteria and having lower cost replacing the previous solution. Once all priority queues are empty, the saved solution is guaranteed to be optimal.

B. Parallel Low Level Search - HDA*

Using OpenMP, each core was assigned an open list to utilize. The first implementation of HDA* is shown in Figure 2. Here, P0 through P3 each expand a node, generating neighbors. The neighbors are then put through a hashing function and distributed amongst processors. Note that each processor can generate work for any other processor, including itself. When a node is expanded, it may be added to the visited list, or used to update nodes in the visited list.

A key issue with this approach is the amount of contention. Open lists are implemented as priority queues to quickly retrieve nodes with the lowest f-values, so inserting multiple nodes simultaneously is very difficult. To prevent this, we employed locks to force mutual exclusion: only one processor can access the open list at a time. At worst case, a single core can block every other core from operating. For instance, if P0 has the lock, and P1 through P3 are trying to push to P0's open list, then only one out of four cores are making progress, significantly impacting performance.

Contention also occurs in the visited list. Each processor must check the visited list to determine if the node they just expanded already exists in the visited list, then it may add to or modify the visited list. Initially, a fine grained lock was employed on a per-node basis within the visited list: there is no contention if one core reads a node, and another core updates a different node. The fine grained lock would simply prevent a read and write, or multiple writes, from occurring on the same node. Testing revealed that this approach, while reducing blocks, produced segmentation faults. While this approach works with reading and modifying existing nodes, inserting new nodes into the visited list causes the underlying data structure to change. Because the visited list is implemented as a thread unsafe unordered map, no operation can occur in parallel with node insertion. This posed a significant bottleneck for performance.

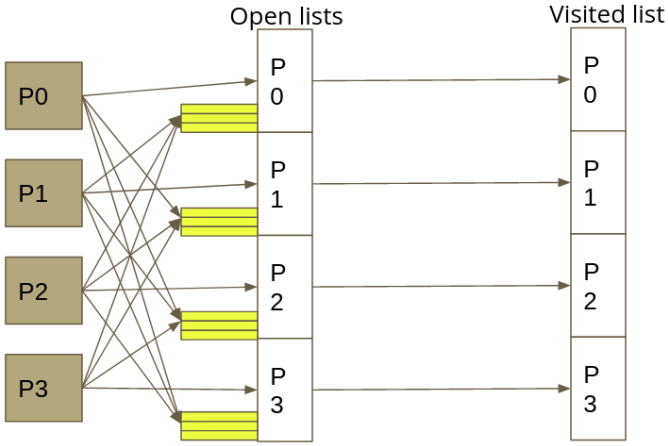


Fig. 3: HDA* redesigned to reduce contention. Yellow blocks are buffers, and visisted list is partitioned for each core

The new implementation is shown in Figure 3. To reduce contention, buffers were added to each open list. When a core, the sender, needs to push to another core's open list, it will instead push to a buffer specifically designated for that sender for that open list. The open list is now exclusively accessed by the core that owns it, and reading or popping from it requires no locks. Updating it does still require a lock on the buffer,

however. For instance, P0 may lock one of its buffers to flush it and update its open list. This will at most block one other core; other cores can still push to their designated buffers without issue during this time. Thanks to the buffers, at worst case one core can only block one other core, rather than all of them. Note that there is no buffer for a core pushing to its own open list; it can do so directly.

The contention on the visited list was removed entirely by partitioning it for each core. The hashing function used to determine which core gets a node is deterministic and consistent, which means that a given node will always be assigned to the same core. Additionally, when a core is checking the visited list, it is searching for a node it may have processed in the past, since it is looking for a copy of the node it is currently assigned. It will never look for a node that was processed by another core. This allows us to break up the visited list: each core will have its own visited list, and there is no need for a core to check or modify another core's visited list. This entirely removes the need for locks, further reducing contention and enhancing parallelization.

C. Parallel High-Level Search

We tried two different approaches for parallelizing the high-level search. The Constraint Tree of the high level search is a binary tree, which reminded us of the final two homeworks. As such, we decided that a good first approach would be to assign subtrees to separate cores. To achieve this, we created a priority queue for each thread. Each thread runs Algorithm 1 and only inserts to its own priority queue. The only difference from the given algorithm then becomes coordinating the termination criteria which has been previously discussed.

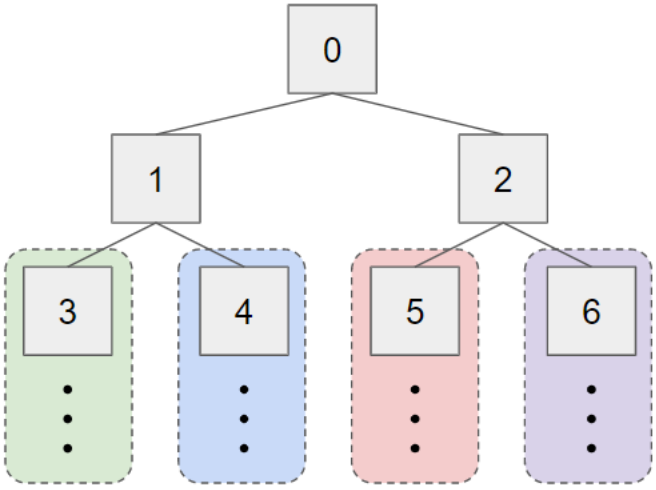


Fig. 4: Subtree assignment used in naive approach. Each color box represents a different core. Each core is assigned a subtree and processes all nodes in that subtree.

One important implementation detail is that we have a warmup period. Cores are only assigned subtrees after enough exist for each core to have its own. For example, in Figure 4 there are four cores. The first three nodes in the tree are processed sequentially by a single core before we have four subtrees that can be distributed across the cores. Not only did this reduce complexity in the code, but it also significantly improved performance on MAPF instances where the number of collisions was low.

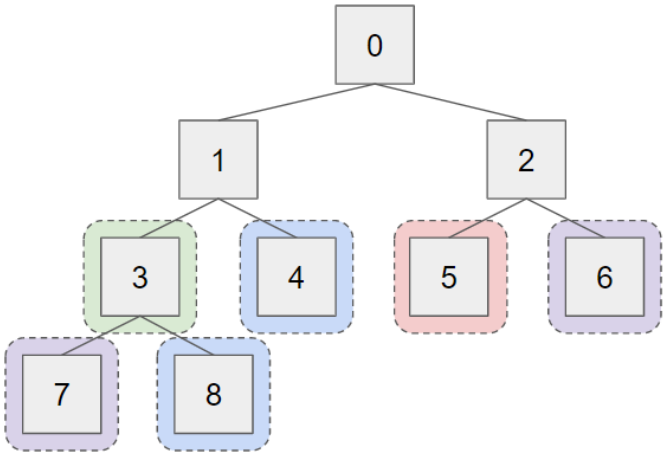


Fig. 5: Node assignment used in second approach. Each color box represents a different core. Nodes are spread evenly across cores and each core finishes processing a given node before moving on to its next assigned nodes.

After preliminary testing, we found that assigning subtrees to cores performed poorly. The main reason behind this was that the subtree assignment meant that we were not focusing resources on the lowest cost nodes. Some subtrees only had nodes with high costs, but the assignment meant that a thread was still processing those nodes leading to wasted processing time. Instead, we took a different approach and applied the HDA* idea of spreading nodes across threads. Figure 5 shows an example assignment of this improved approach. Instead of using a hash like HDA* to assign nodes, we assigned new nodes to the priority queue that had the lowest number of elements. This helped to distribute the work better across cores. Additionally, now that we have different threads accessing each other's priority queue we have to use locks when pushing or popping. This was another reason we chose to initially pursue the subtree assignment strategy. We knew that locks would be needed if we wanted to spread nodes across cores and we felt that it would cause a significant speed decrease. However, it turned out that the benefits gained by spreading the nodes was significantly higher than the slow down introduced by the locks. Similar to the previous method, the warmup period still exists where we process nodes sequentially until there exist enough to give each available thread a separate node.

V. Results

The maps used for evaluation are shown in Figure 6. The maps cover a wide range of different environments, allowing us to see which types of maps the parallelized CBS performed

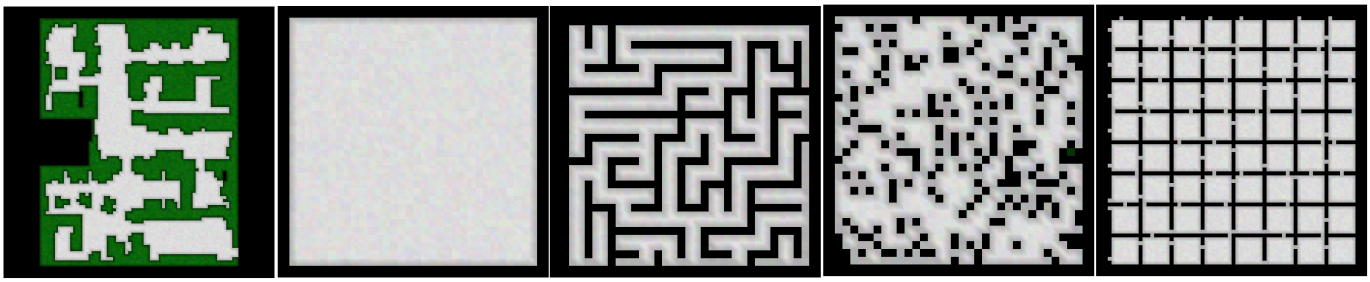


Fig. 6: Maps used for evaluation. Left to right: den312d, empty, maze, random, room

TABLE I: HDA* Speedup relative to sequential A*

Map	A* runtime (ms)	HDA* runtime (ms)	Speedup
den312d	237.1	212.2	1.12
empty	24.9	16.6	1.51
maze	580.8	502.1	1.16
random	19.5	16.5	1.18
room	592.3	598.0	0.99

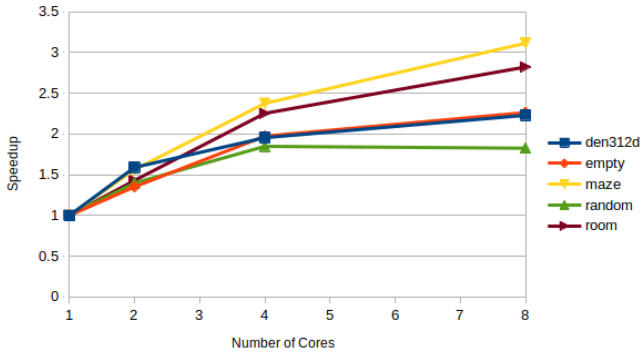


Fig. 7: HDA* multi-core speedup relative to single core HDA* performance

well and poorly on. All experiments were performed on the GHC machines using 8 cores unless stated otherwise.

A. Parallel Low Level Search - HDA*

The runtimes and relative speedups of HDA* compared to A* are shown in Table I. A random MAPF instance was generated for each environment using 40 agents, then the median of 100 tests per map was used to represent the overall runtime for the algorithm. The results showed that HDA* tended to perform better than A*, averaging 20 percent increase in performance over all five maps. This is significantly lower speedup than we had anticipated. Figure 7 shows that as the number of cores increases, the overall speedup quickly plateaus, which explains the lower than expected performance. We analyzed the algorithm and implementation to determine the root cause: the main lines of investigation were workload increase, workload balance, and coordination overhead.

1) Workload Increase: Table II shows the number of nodes expanded in the sequential A* algorithm, and the parallelized HDA* algorithm. Since HDA* is an exhaustive search algorithm, it must expand significantly more nodes than A*, which means its workload is much larger. As a result, the ideal speedup is not equal to the number of cores: algorithmically, the workload increases by a factor of 4.39 on average across all five maps. This limits the ideal speedup to an average of 1.93 across all five maps when using 8 cores. Table II also shows the ideal speedup for each map, assuming no duplicated effort or coordination overhead.

TABLE II: Number of Nodes per Algorithm

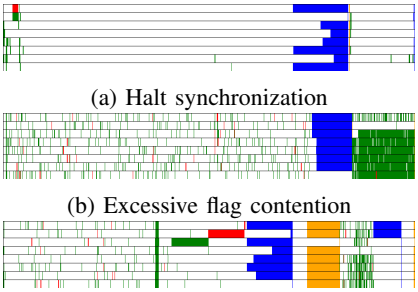
Map	A*	HDA*	Factor Increase	Ideal Speedup
den312d	69567	412653	5.93	1.35
empty	4673	14057	3.01	2.66
maze	195222	926298	4.74	1.69
random	4425	15707	3.55	2.25
room	179490	846236	4.71	1.70

2) Workload Balance: The workload balance was also investigated, as shown in Table III. Of the 8 cores, we noticed that the first core, P0, tended to have less work than the other 7 cores. P1-P7 had very good workload balancing, remaining very close to the ideal workload percentage of 12.5%. The workload balancing is determined by the hashing function: if the hashing function is biased, then certain cores will get more work than others. Though the balance could be improved by giving more work to P0, since seven of the eight cores had excellent balance, and the remaining core was still contributing to the overall performance, we determined that the workload balance was not the primary bottleneck.

TABLE III: Percentage of Nodes Processed by each core

Map	P0 (%)	P1-P7 Min (%)	P1-P7 Max (%)
den312d	8.59	12.84	13.58
empty	9.23	12.64	13.21
maze	8.98	12.76	13.18
random	9.16	12.06	13.58
room	8.18	11.93	13.81

3) Coordination Overhead: Figure 8 shows a visualization of contention and synchronization across 8 cores during HDA* operation. Each subfigure shows 8 timelines, one for each core, showing when that core is blocked. For most of the algorithm, cores must access each other's buffers, as well as check each other's status flag. Both of these operations require acquiring a lock, and the block time for these operations are



(a) Halt synchronization

(b) Excessive flag contention

(c) Termination synchronization

Fig. 8: Visualization of contention and synchronization. Colors: red = buffer contention, green = flag contention, blue = halt synchronization, orange = exit synchronization

shown in red and green, respectively. Once all status flags are set, a halt synchronization occurs, shown in blue. Once synchronized, each core must double check that their own open lists and buffers are empty to ensure completeness. They will then synchronize once more before terminating the program or resuming the search; the second synchronization is shown in orange. Figure 8a shows a case where most of the overhead is due to halt synchronization. There is very little buffer and flag contention, so the cores are mostly unimpeded throughout the search. Figure 8b shows an instance where there is significant flag contention on top of waiting due to synchronization. Before the halt synchronization, there is an acceptable amount of flag contention, and very little buffer contention. However, after halt synchronization, the algorithm has resumed search as it determined that it cannot exit yet. Though there is still work to do, most cores are done or nearly done. Cores that have finished their work will constantly access the flags of other cores to check their status, resulting in large amounts of flag contention. Note that 7 of the 8 cores have significant contention; one core still has work to do, so it never checks the flags and therefore has very little flag contention. Flag contention is not ideal, but it is generally acceptable as a core will only suffer flag contention if it has finished all of its local work. While waiting to acquire the flag lock, there is very little work for it to do, so the contention isn't blocking a significant amount of progress.

Figure 8c shows a case that occurs quite rarely. The second synchronization is typically very quick, as each core checking their own open lists and buffers requires no locks, and the checking itself is very fast. However, when an algorithm uses all the cores on a machine, it's possible that one of the cores gets preempted by the operating system. Preemption will force one of the cores to be far behind the other cores, which will significantly increase the cost of synchronization. This may also occur for halt synchronization, but it is most noticeable with exit synchronization. Also note that this map has large buffer and flag contention before the first halt synchronization. This is consistent when running on certain maps. This shows that buffer and flag accessing is heavily impacted by the nature of the problem, as some maps will generate nodes such that contention is more likely to occur.

HDA* was implemented using OpenMP, which uses a Shared Address Space (SAS) model. SAS models require locks to avoid race conditions, which results in the blocking and inefficiency. Implementing HDA* using a message passing model may have been the better choice: if each core sent nodes as a message and published their status flag, then there would be no contention, thus improving performance and scaling with number of processors. One key weakness of the SAS is the barrier: when a core reaches a barrier, it must wait for all other cores, and cannot abort the barrier if too much time passes. Message passing could also be used as a way around this issue, as received information could be used to determine if halting and exiting are appropriate actions.

B. Parallel High Level Search

In our project proposal we stated that we would measure the success rate of our parallel implementation compared to a sequential implementation. Our experiment setup was as follows: for each map and for each implementation, run the algorithm 10 times with a random set of start and goal locations. Repeat this with an increasing number of agents until the success rate is less than 10%. The timeout limit was set to 1 minute and the algorithm used all 8 cores available. The results for the 5 different maps are given in Figure 9. As can be seen from the graphs, the parallel implementation in orange has a consistently higher success rate than the sequential method. However, the success rate was not as high as we initially aimed for.

In order to better understand why this was the case we also performed another experiment where we compared the relative speedup gained from using the parallel approach. For this experiment, we again used a random set of start and goal locations but fixed the number of agents. For the den312d, empty-32-32, and random-32-32 maps we used 20 agents. For the maze-32-32 and room-64-64 maps we used 10 agents. These numbers were chosen as they were the points at which those maps had displayed around 80% success rate. The timeout limit in this experiment was set to 10 minutes. The resulting relative speedup is shown in Figure 10. As the results show, there is an extremely large spread in the relative speedup but our parallel method is on average 2.23× faster than the sequential method. Despite this we were still disappointed with the performance. Looking closer at the data we noticed that when the run-time was over 1 second, the relative speedup was significantly larger. Thus, we chose to create a separate plot where we only considered data points where the runtime was over 1 second which can be seen in Figure 11. In this case, the average relative speedup jumps to 4.83× with a maximum speedup of 6.18× which is significantly higher than before.

One important property of the MAPF problem is that it is NP-Hard. As a result, some problems can be trivially simple and take the sequential implementation under 5ms to solve while other cases can be exceedingly difficult requiring over 50 minutes to solve even for the same map and number of agents. The run-time is heavily dependent on the configuration

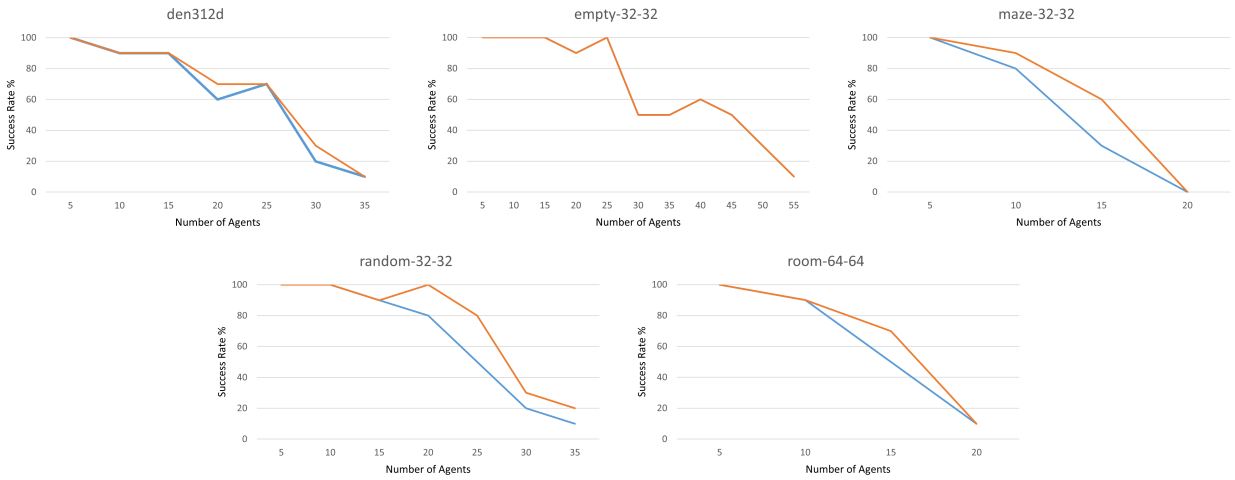


Fig. 9: Graphs showing success rates for all maps. Success rate for the parallel approach with 8 cores is shown in orange and the sequential approach is shown in blue.

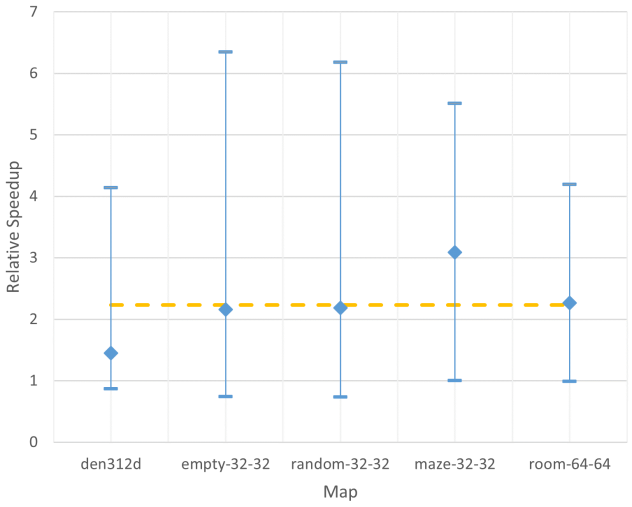


Fig. 10: The average, max, and min relative speedup of our parallel approach using 8 cores across 10 runs for each map. The yellow dotted line shows the average speedup across all maps which was 2.23.

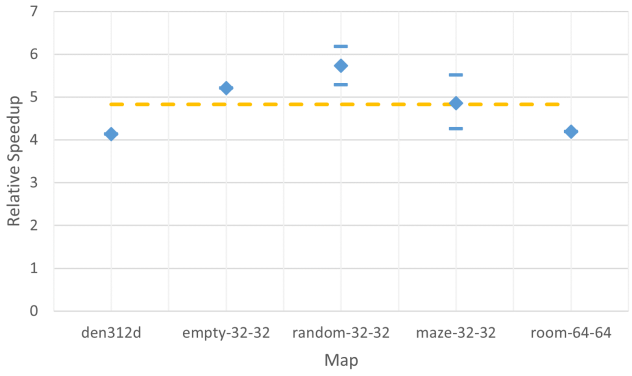


Fig. 11: The average, max, and min relative speedup of our parallel approach using 8 cores for only runs that took longer than 1 second. The yellow dotted line shows the average speedup across all maps which was 4.83.

of starts and goals and how many conflicts the optimal solution contains. The main driving force behind why our algorithm performs better on problems that take over a second is due to the warmup period described in the approach section. There is an overhead cost associated with the initial spreading of the nodes to the cores so when the sequential warmup period of the parallel algorithm is a higher percentage of the overall runtime we typically see poor relative speedups. In general, more conflicts in the optimal solutions means more nodes need to be processed which leads to the initial warmup being amortized and thus leading to a higher relative speedup.

VI. Conclusion

We attempted two approaches to parallelizing the CBS algorithm, one targeting the low-level and another targeting the high-level. In our project proposal we set out to achieve a 25% increase in success rate for our parallel implementation. While we did not necessarily find this performance improvement we found that it may have been a misguided target. Instead, evaluating the relative speedup showed that in the best case we were able to solve for the optimal solution 6.18× faster than the sequential method. However, the large variations in complexity of MAPF problems meant that only occasionally did we see this performance boost. On average, we were 2.23× faster but could occasionally be slightly slower for very trivial problems. Overall, we found that targeting the high level search for paralllelization was more effective than targeting the low level. Future work could look to implement lock-free

priority queues as these proved to be a significant bottleneck in our implementations. A message passing implementation of HDA* should also be evaluated to determine how beneficial such an approach can be for speeding up CBS.

VII. Work Contribution

Both team members contributed equally, 50%-50%. Matthew Booker implemented, tested, and analyzed the high-level parallelization of CBS. Jaekyung Song implemented, tested, and analyzed the HDA* algorithm.

References

[1] G. Sharon, R. Stern, A. Felner, N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding", Artificial Intelligence, Volume 219, 2015, Pages 40-66, https://doi.org/10.1016/j.artint.2014.11.006.

[2] Jiaoyang Li., CMU 16-891: Multi-robot Planning and Coordination

[3] Kishimoto, A., Fukunaga, A., Botea, A. (2009). Scalable, Parallel Best-First Search for Optimal Sequential Planning. Proceedings of the International Conference on Automated Planning and Scheduling, 19(1), 201-208. https://doi.org/10.1609/icaps.v19i1.13350