Speculative Path Planning

Mohammad Bakhshalipour Mohamad Qadri Dominic Guri

Abstract

Parallelization of A⋆ path planning is mostly limited by the number of possible motions, which is far less than the level of parallelism that modern processors support. In this project, we go beyond the limitations of traditional parallelism of A⋆ and propose **Speculative Path Planning** *to accelerate the search when there are abundant idle resources. The key idea of our approach is* **predicting future state expansions relying on patterns among expansions** *and aggressively parallelize the computations of prospective states (i.e. pre-evaluate the expensive collision checking operation of prospective nodes). This method allows us to maintain the same search order as of vanilla A⋆ and safeguard any optimality guarantees. We evaluate our method on various configurations and show that on a machine with 32 physical cores, our method improves the performance around 11× and 10× on average over a counterpart single-threaded and multi-threaded implementations respectively. The code to our project can be found here https://github.com/bakhshalipour/speculative-path-planning.*

1. Introduction

Graph-based (weighted) A⋆ search is the cornerstone of many planning algorithms. Given a good heuristic, A⋆ can significantly reduce the number of state expansions, thereby reducing the search time. In the context of path planning, A⋆ is widely used in many applications to find a path from a source position to one (or more) destination position(s). A⋆ evaluates various nodes in the state space and finds an optimal path from the source to the destination. Specifically, whenever a node is expanded, A⋆ *evaluates its neighbors*, updates its metadata (setting the node to closed), and then expands another node with the highest prospect of being close to the destination. The evaluation of neighbors is essentially done to find out whether the neighbor nodes should be considered for expansion by the algorithm or not. Being *collision-free* is a condition for a node to be added to the open list (a node that is potentially part of a solution), meaning that the position on the map should not cause the robot to collide with an obstacle. The evaluation of this process is called *collision checking*.

Collision checking can be an extremely compute-intensive task, taking up to 99% of the planning time [1,4]. One solution for improving the execution time is *parallelizing* the collision checking operations. Since the evaluations of neighbors are completely independent, they can be easily parallelized, thereby achieving speedup. However, the number of neighbors is not typically large; in other words, the parallelization degree is severely limited. The number of neighbors is determined by the number of motions the robot can make. For example, on an 8-connected grid, the robot can move in cardinal and intercardinal directions, implying up to eight parallelization operations to precompute the status of the immediate neighbours. On the other hand, today's mainstream computing machines support much more parallelism. CPUs with tens or even beyond a hundred threads, and GPUs with thousands of threads are widespread these days. As a result, when running a path planning kernel on a mainstream machine, most of the physical cores remain idle.

In this project, we propose *Speculative Path Planning*. Our objective is to take advantage of idle resources to accelerate the execution time of the A^* and weighted A^* algorithms without breaking their optimality (of A^*) and ε-suboptimality (of weighted A^*) guarantees. We start by making a couple of observations: 1) collision checking is usually an expansive operation that could dominate the search time and 2) given a good heuristic, the search usually follows a pattern of expanding successive nodes along the same directions (for example, the expansions in orange seen in Figure 1). We aim to utilize the available idle

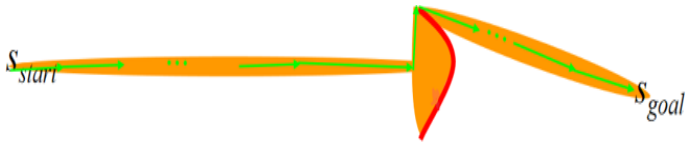


Figure 1: An example of expanded states using weighted A⋆ .

threads to perform speculative collision checking of cells likely to be explored in the future to accelerate the graph-search time of the A^* and $weighted-A^*$ algorithms. When a new node *exp_node* is expanded using A^* , some (or all) the available threads are first used to find the collision status of its immediate neighbors in parallel. If any available idle threads remain, we use them to compute the collision status of the neighbours of nodes most likely to get expanded in the near future and hence, the search becomes less stalled for performing costly collision checking operations. Importantly, we do not limit our method to compute the collision checking of the current node neighbors; instead, we go further and explore multiple nodes if idle threads are available. These nodes are determined speculatively using a speculation strategy detailed in this report and the collision status of these neighboring nodes is computed in parallel, i.e., we perform speculative parallelization.

We extensively evaluate our approach on a 32-core processor and show that it markedly improves execution time. On the four maps of Homework 1, our proposal accelerates execution by 11.1× over a single-threaded implementation. Compared to a non-speculative parallel implementation that uses an equal number of threads, our approach improves the speedup by 10.4×.

2. Approach

2.1. Speculative path planning

Algorithm 1 summarizes the steps of our proposed speculative path planning algorithm. This section will explain each step in detail.

Algorithm 1 Speculative path planning

```
1:  procedure SPECULATIVE PATH PLANNING
2:      INIT global_nodes_state ← ∅
3:      max_threads ← M
4:      max_forward_speculation ← S
5:      while !heap.empty() do
6:          exp_node ← heap.top()
7:          setClosed(exp_node) = true
8:          if exp_node == goal then
9:              return path via backtracking
10:         unkn_neighs = getNeighUnknown(exp_node)
11:         workload_per_thread[] ← assignWorkload(unkn_neighs)
12:         for workload in workload_per_thread do
13:             launch_thread(find_collision(workload))
14:         if available_idle_threads > 0 and unkn_neighs ≠ ∅ then
15:             SPECULATE(exp_node )
16:         join_threads()        ▷ Continue with vanilla A* operations
17:         for n in free_neighbors(exp_node) do
18:             n.g ← cost(exp_node) + exp_node.parent.g
19:             n.h ← get_heuristic(n)
20:             if n.g < previous_g_val_if_exists(n) then
21:                 add_to_open_list(n)
```

We maintain a global map (line 2) indicating the collision status of each node (defined by its x, y position on the map). The collision status can be one of the following:

- `UNKNOWN`: the collision status of the node is unknown and needs to be calculated
- `FREE`: the node is free.
- `COLLISION`: the node is not free (an obstacle).

We determine the maximum number of threads allocated to the search process (line 4). In line 5, we specify the *max_forward_speculation* parameter, which indicates the maximum number of the speculation lookahead searches. Lines 5 to 9 belong to the vanilla A^* algorithm: a new node *exp_node* is retrieved from the open list. It's status is then set to closed. If *exp_node* is the goal position, we return the final planned path via backtracking.

2.2. Parallelizing the collision checking operation of immediate neighbors

Given the newly expanded node *exp_node*, we have N immediate neighbors (in our experiments N=8 immediate neighbors since we are considering an 8-connected grid). We refer to one of these neighbours as $n_{immediate}$. Either the collision status of $n_{immediate}$ has already been computed (while evaluating previous nodes or using the speculation process) or it is still unknown and needs to be computed. In line 10, we retrieve all of *exp_node*'s neighbors that have status equal to `UNKNOWN`. We call this set of nodes *unkn_neighs*. In lines 11-13, we equally split [1] the workload of computing collision checking of all nodes in *unkn_neighs* across the available threads (i.e., each thread could get a list of nodes to compute). In this case, collision checking is performed by what we name "non-speculative threads." If at least a single thread remains, we call the SPECULATE function (lines 14-15). Note that if the set *unkn_neighs* is empty, we do not perform speculation in order to not stall the main search.

2.3. Speculation strategy

Algorithm 2 The speculation function

```
1:  procedure SPECULATE
2:      dirX = exp_node.x - exp_node.parent.x;
3:      dirY = exp_node.y - exp_node.parent.y;
4:      counter = 1
5:      while counter ≤ max_forward_speculation do
6:          spec_node.x = exp_node.x + counter×dirX
7:          spec_node.y = exp_node.y + counter×dirY
8:          for n in neigbors(speculated_node) do
9:              if available_idle_threads > 0 then
10:                 launch_thread(find_collision(n))
11:         counter = counter + 1
```

We converged to a simple speculation strategy that provided a significant increase in search time (Algorithm 2). In lines 2-3, we retrieve the direction d that was taken to reach the currently expanded node *exp_node* from its parent node. We then explore nodes in this same direction d up to the maximum lookahead speculation threshold (lines 6-7), which was defined in the parameter *max_forward_speculation*. If an explored node p along this path has a status equal to `UNKNOWN`, we launch a thread to calculate the collision status for **one of its neighbors only** (line 10). We name these threads *speculative threads*. We compare this to section 2.2 where a thread could potentially calculate the collision status of multiple nodes. By limiting each speculative thread to processing a single neighbor, we are assured that we are never bottlenecked by the speculation process and that we are off the critical path.

Now going back to Algorithm 1, in line 16, we join all threads (speculative and non-speculative) and continue with regular A^* operations in lines 17-21 by adding the eligible free neighbors of *exp_node* to the open list.

Note that we have designed and attempted different speculation strategies. For example, by looking further back in the search history (the ancestors of *exp_node*). However, we have found that the strategy described above provided the best performing and stable results across different maps.

[1] Depending on the number of threads available, some threads might be assigned one extra neighbor for computation

2.4. The optimality of the proposed method

Note that the proposed algorithm maintains the same order of opening and closing of nodes compared to the regular A^* algorithm. As such, our algorithm returns an optimal solution when using A^* search and an $\varepsilon-$suboptimal solution when using weighted A^*.

3. Evaluation

3.1. Experimental Setup

We evaluate our proposal on a WF node of the Narwhal cluster [3], with 32 cores. We consider a 2D state space, an 8-connected grid, and a static target. We conduct the experiments on the four maps of Homework 1. As a quick proof-of-concept, we use a long latency dummy function named *find_collision* (line 13 in Algorithm 1 and line 10 in Algorithm 2) as the collision checking function . This function decrements a large counter to simulate large collision checking time, updates *global_nodes_state* with the correct status of a node (free or an obstacle), and returns. We set the counter such that each collision checking operations takes around 25ms. We use the Euclidean distance as the search heuristic and set the ε value of the search to 1. We use POSIX thread execution model [2] for implementing multi-threaded programs.

3.2. Execution Time

Figure 7 shows the effect of multi-threading and multi-threading plus speculation on execution time. The execution time is normalized relative to the non-speculative single-threaded execution time.

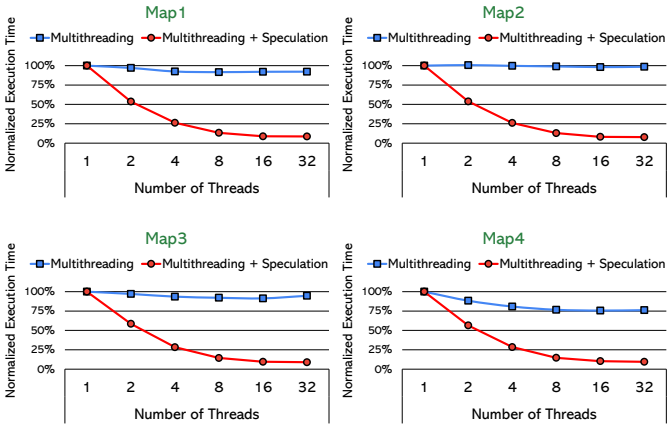


Figure 2: Execution time with various numbers of threads normalized to a single-threaded implementation.

Results show that increasing the number of threads with pure multi-threading (i.e., without speculation) gives diminishing returns, and beyond an extent, there is no speedup. Such a limitation is not unexpected since the maximum available parallelism with pure multi-threading equals the number of possible motions (8 in our experiments) and is far less than the maximum available threads (32 in our experiments). However, speculation breaks the parallelization barriers and substantially accelerates the execution time with larger numbers of threads. The average execution time improvement for our approach (with 32 threads) is 11.1× and 10.4× relative to non-speculative single-threaded and multi-threaded implementations, respectively.

3.3. Speculation Accuracy

Figure 3 shows the *speculation accuracy* of our approach with different numbers of threads on all four maps. We define speculation accuracy as the percentage of speculative computations (i.e., collision checkings) whose result is eventually used by the search. In other words, we consider speculative collision checkings that are performed on nodes that never get used as inaccurate speculations.

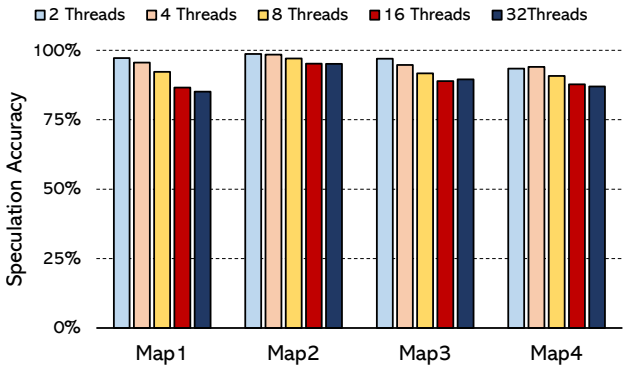


Figure 3: The accuracy of speculations in different configurations.

As the figure shows, most speculations are accurate, thus substantiating our insight that expansions exhibit specific patterns, leading to their predictability. The average accuracy with 32 threads (worst case) is 89.1%. For more results on more maps (from https://movingai.com), see the Appendix.

3.4. Division of Labor

Figure 4 shows the average number of collision checking operations per expansion. The figure further shows how many of the operations are performed by non-speculative threads and how many by speculative threads.

When increasing the number of threads, the contribution of computations performed by speculative threads goes up. As such, contributions from non-speculative threads decrease; this means that collision checking in the critical path of execution is speculatively evaluated; thus, the program is not stalled by non-speculative collision checking.

3.5. Demo

The video[2] demonstration shows the relative performance of A* implemented on a single thread versus 16-threads parallel execution with and without speculation. Because the video is

[2] https://youtu.be/zf6-Sv3IwXg

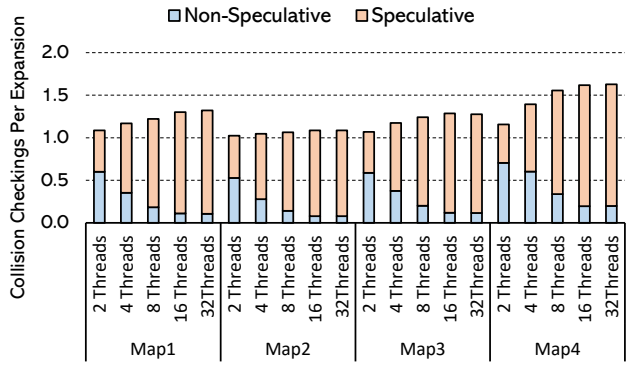


Figure 4: The average number of collision checking operations per node expansion in different setups.

generated from log-files, it shows the correct relative progress per frame but not the correct speed. The single-threaded implementation takes 145.9 sec — comparable to the non-speculative multi-threaded version that took 138.82 sec. Speculation significantly improves the execution time to 18.97 sec.

4. Conclusion and Future Work

The A* algorithm is the backbone of many important planning methods. Accelerating A* can lead to substantial performance improvements in wide-ranging applications. In this project, we proposed a method to opportunistically parallelize A* based on observing that consecutive expansions exhibit recurring patterns. We propose to exploit these patterns by using them to speculatively identify computations that may be required in the future using idle threads; for this project, this entails running expensive collision avoidance for anticipated future states in a grid-world path planning task. Our approach on a multi-core processor showed that it could accelerate planning by 11.1$\times$ and 10.4$\times$ as compared to vanilla single-threaded and multi-threaded implementations, respectively.

The gist of the approach presented is maximizing computational throughput, which is timely because hardware manufacturers are packing more cores per processor, and newer programming languages like Julia, Swift, and Go are designed to take advantage of such parallelization opportunities. Therefore, future work for this project involves evaluating our proposal on other substrates like GPUs and FPGAs that offer parallelization opportunities far beyond CPUs. Additionally, we may evaluate the effectiveness of the speculative computation on settings that include realistic collision checking methods, different motion primitives, various heuristic functions, and other graph search scenarios like symbolic planning and task and motion planning (TAMP).

References

[1] Joshua Bialkowski, Sertac Karaman, and Emilio Frazzoli. Massively Parallelizing the RRT and the RRT. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[2] David R Butenhof. *Programming with POSIX Threads*. 1997.

[3] The PDL Narwhal Cluster. https://wiki.pdl.cmu.edu/Narwhal.

[4] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

Appendix

To further test the performance our proposed method we added more maps from https://movingai.com, see Figure 5.

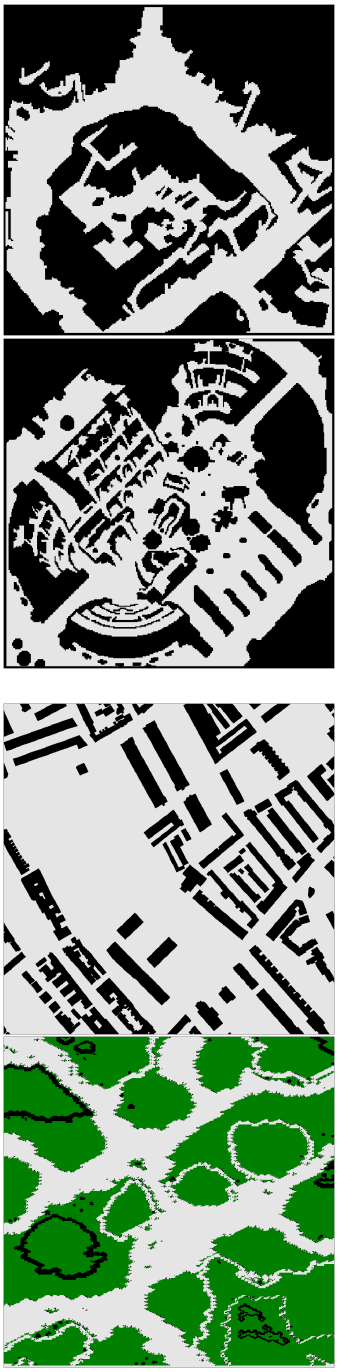


Figure 5: Experimental maps obtained from https://movingai.com: AR0400SR, AR0700SR, London, and Sapphire Isles

The results were consistent with the homework maps' results, showing significant execution time reduction with speculation.

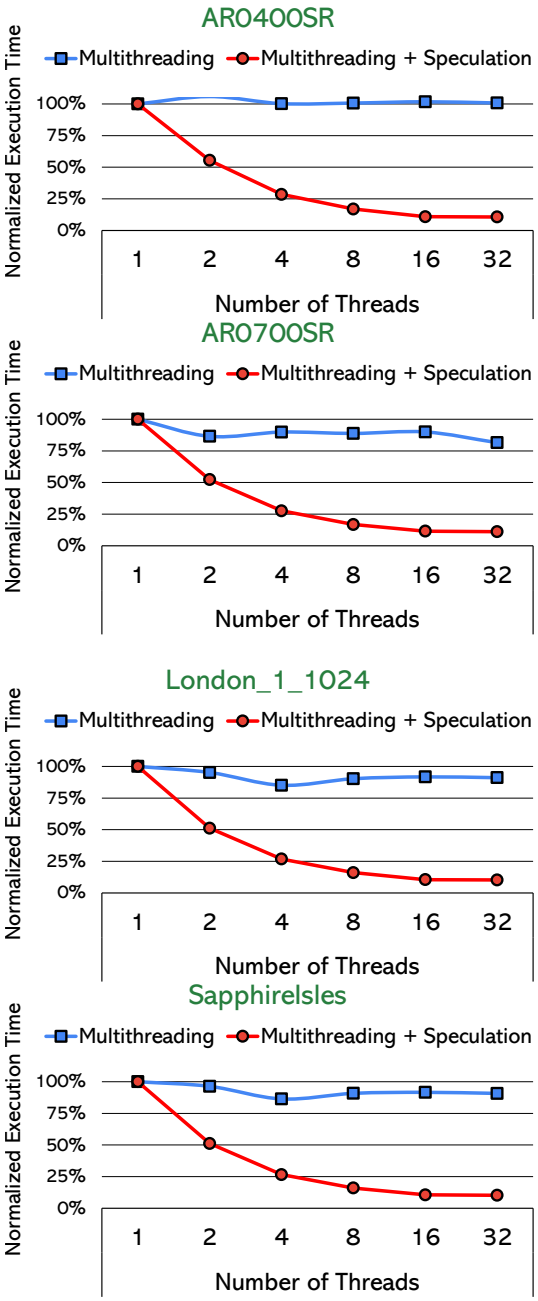
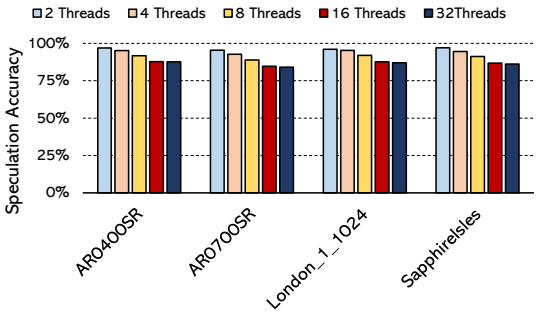
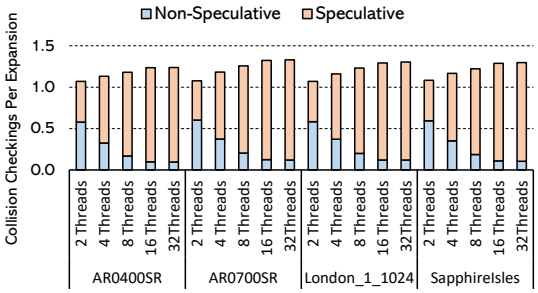


Figure 6: Execution time with various numbers of threads normalized to a single-threaded implementation for https://movingai.com maps



(a) The accuracy of speculations in different configurations.



(b) The average number of collision checking operations per node expansion in different setups.

Figure 7: https://movingai.com maps — speculation accuracy and collision checkings per expansion.