ePA*SE: Edge-Based Parallel A* for Slow Evaluations

Shohin Mukherjee, Sandip Aine, Maxim Likhachev

The Robotics Institute, CMU
{shohinm, asandip, mlikhach}@andrew.cmu.edu

Abstract

Parallel search algorithms harness the multithreading capability of modern processors to achieve faster planning. One such algorithm is PA*SE (Parallel A* for Slow Expansions), which parallelizes state expansions to achieve faster planning in domains where state expansions are slow. In this work, we propose ePA*SE (Edge-Based Parallel A* for Slow Evaluations) that 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. On the theoretical front, we show that ePA*SE provides rigorous optimality guarantees. In addition, ePA*SE can be trivially extended to handle an inflation weight on the heuristic resulting in a bounded suboptimal algorithm w-ePA*SE (Weighted ePA*SE) that trades off optimality for faster planning. On the experimental front, we validate the proposed algorithm in two different planning domains: 1) motion planning for 3D humanoid navigation and 2) task and motion planning for a dual-arm robotic assembly task. We show that ePA*SE can be significantly more efficient than PA*SE and other alternatives. The open-source code for ePA*SE along with the baselines is available here: https://github.com/shohinm/parallel_search

Introduction

Graph search algorithms such as A* and its variants (Hart, Nilsson, and Raphael 1968; Pohl 1970; Aine et al. 2016) are widely used in robotics for task and motion planning problems which can be formulated as a shortest path problem on an embedded graph in the state-space of the domain. A* maintains an open list (priority queue) of discovered states, and at any point in the search, it expands the state with the smallest priority (f-value) in the list. During the expansion, it generates the successors of the state and evaluates the cost of each edge connecting the expanded state to its successors. In robotics applications such as in motion planning, edge evaluation tends to be the bottleneck in the search. For example, in planning for robot-manipulation, edge evaluation typically corresponds to collision-checks of a robot model against the world model at discrete interpolated states on the edge. Depending on how these models are represented (meshes, spheres, etc.) and how finely the edges are sampled for collision-checking, evaluating an edge can get very expensive. As a consequence, the state expansions are typically slow.

In order to speed up planning in domains where state expansions are slow, an optimal parallelized planning algorithm PA*SE (Parallel A* for Slow Expansions) and its (bounded) suboptimal version wPA*SE (Weighted PA*SE) were developed (Phillips, Likhachev, and Koenig 2014). Unlike other parallel search algorithms, in which the number of times a state can be re-expanded increases with the degree of parallelization (Irani and Shih 1986; Zhou and Zeng 2015; He et al. 2021), PA*SE expands states in a way that each state is expanded at most once. The key idea in PA*SE is that a state $\mathbf{s}$ can be expanded before another state $\mathbf{s}'$ if $\mathbf{s}$ is *independent* of $\mathbf{s}'$ i.e. expansion of $\mathbf{s}'$ cannot lead to a shorter path to $\mathbf{s}$. If the independence relationship holds in both directions i.e. $\mathbf{s}'$ is also independent of $\mathbf{s}$, then $\mathbf{s}$ and $\mathbf{s}'$ can be expanded in parallel. Though PA*SE parallelizes state expansions, for a given state, the successors are generated sequentially. This is not the most efficient strategy, especially for domains with large branching factors. In addition, this strategy is particularly inefficient in domains where there is a large variance in the edge evaluation times. Consider a state being expanded with several outgoing edges, such that the first edge is expensive to evaluate, while the others are relatively inexpensive. In this case, since a single thread is evaluating all of the edges in sequence, the evaluations of the cheap edges will be held up by the one expensive edge. This happens often in planning for robotics. Consider full-body planning for a humanoid. Evaluating a primitive that moves just the wrist joint of the robot requires collision checking of just the wrist. However, evaluating the primitive that moves the base of the robot, requires fully-body collision checking of the entire robot. One way to avoid this would be to evaluate the outgoing edges in parallel, which PA*SE doesn't do. However, this seemingly trivial modification does not solve another cause of inefficiency in PA*SE i.e. the evaluation of the outgoing edges from a given state is tightly coupled with the expansion of the state. In other words, all the outgoing edges from a given state must be evaluated at the same time when the state is expanded. This leads to more edges being evaluated than is necessary, as we will show in our experiments.

Therefore in this work, we develop an improved optimal parallel search algorithm, ePA*SE (Edge-Based Parallel A* for Slow Evaluations), that eliminates these inefficiencies by 1) decoupling edge evaluation from state expansions and 2) parallelizing edge evaluations instead of state expansions. ePA*SE exploits the insight that the root cause of slow expansions is typically slow edge evaluations. Each ePA*SE thread is responsible for evaluating a single edge, instead of expanding a state and evaluating all outgoing edges from it, all in a single thread, like in PA*SE. This makes ePA*SE significantly more efficient than PA*SE and we show this by evaluating it on two planning domains that are quite different: 1) 3D indoor navigation of a mobile manipulator and 2) a task and motion planning problem of stacking a set of blocks by a dual-arm robot.

Related Work

Parallel planning algorithms seek to make planning faster by leveraging parallel processing.

Parallel sampling-based algorithms There are a number of approaches that parallelize sampling-based planning algorithms. Probabilistic roadmap (PRM) based methods, in particular, can be trivially parallelized, so much so that they have been described as "embarrassingly parallel" (Amato and Dale 1999). In these approaches, several parallel processes cooperatively build the roadmap in parallel (Jacobs et al. 2012). Parallelized versions of RRT have also been developed in which multiple cores expand the search tree by sampling and adding multiple new states in parallel (Devaurs, Siméon, and Cortés 2011; Ichnowski and Alterovitz 2012; Jacobs et al. 2013; Park, Pan, and Manocha 2016). However, in many planning domains involving planning with controllers (Butzke et al. 2014), sampling of states is typically not possible. One such class of planning domains where state sampling is not possible is simulator-in-the-loop planning, which uses an expensive physics simulator to generate successors (Liang et al. 2021). We, therefore, focus on the more general technique of search-based planning which does not rely on state sampling.

Parallel search-based algorithms A trivial approach to achieve parallelization in weighted A* is to generate successors in parallel when expanding a state. The downside is that this leads to minimal improvement in performance in domains with a low branching factor. Another approach that Parallel A* (Irani and Shih 1986) takes, is to expand states in parallel while allowing re-expansions to account for the fact that states may get expanded before they have the minimal cost from the start state. This leads to a high number of state expansions. There are a number of other approaches that employ different parallelization strategies (Evett et al. 1995; Zhou and Zeng 2015; Burns et al. 2010), but all of them could potentially expand an exponential number of states, especially if they employ a weighted heuristic. In contrast, PA*SE (Phillips, Likhachev, and Koenig 2014) parallelly expands states at most once, in such a way that does not affect the bounds on the solution quality. Though PA*SE parallelizes state expansions while preventing re-expansions, as explained earlier, it is not efficient in domains where edge evaluations are expensive since each PA*SE thread sequentially evaluates the outgoing edges of a state being expanded. A parallelized lazy planning algorithm, MPLP (Mukherjee, Aine, and Likhachev 2022), achieves faster planning by running the search and evaluating edges asynchronously in parallel. Just like all lazy search algorithms, MPLP assumes that successor states can be generated without evaluating edges, which allows the algorithm to defer edge evaluations and lazily proceed with the search. However, this assumption doesn't hold true for a number of planning domains in robotics. In particular, consider planning problems that use a high-fidelity physics simulator to evaluate actions involving object-object and object-robot interactions (Liang et al. 2021). The generation of successor states is typically not possible without a very expensive simulator call. In such domains, where edge evaluation cannot be deferred, MPLP is not applicable. One of the domains in our experiments falls in this class of problems.

GPU-based parallel algorithms There has also been work on parallelizing A* search on a single GPU (Zhou and Zeng 2015) or multiple GPUs (He et al. 2021) by utilizing multiple parallel priority queues. Since these approaches must allow state re-expansions, the number of expansions increases exponentially with the degree of parallelization. In addition, GPU-based parallel algorithms have a more fundamental limitation which stems from the single-instruction-multiple-data (SIMD) execution model of a GPU. This means that a GPU can only run the same set of instructions on multiple data concurrently. This severely limits the design of planning algorithms in several ways. Firstly, the code for expanding a state must be identical, irrespective of what state is being expanded. Secondly, the set of states must be expanded in a batch. This is problematic in domains that have complex actions that correspond to forward simulating dissimilar controllers. In contrast to these approaches, ePA*SE achieves parallelization of edge evaluations on the CPU which has a multiple-instruction-multiple-data (MIMD) execution model. This allows ePA*SE the flexibility to efficiently parallelize dissimilar edges, and therefore generalize across all types of planning domains.

Problem Definition

Let a finite graph $G = (\mathcal{V}, \mathcal{E})$ be defined as a set of vertices $\mathcal{V}$ and directed edges $\mathcal{E}$. Each vertex $v \in \mathcal{V}$ represents a state $\mathbf{s}$ in the state space of the domain $\mathcal{S}$. An edge $e \in \mathcal{E}$ connecting two vertices v_1 and v_2 in the graph represents an action $\mathbf{a} \in \mathcal{A}$ that takes the agent from corresponding states $\mathbf{s}_1$ to $\mathbf{s}_2$. In this work, we assume that all actions are deterministic. Hence an edge e can be represented as a pair $(\mathbf{s}, \mathbf{a})$, where $\mathbf{s}$ is the state at which action $\mathbf{a}$ is executed. For an edge e, we will refer to the corresponding state and action as $e.\mathbf{s}$ and $e.\mathbf{a}$ respectively. In addition, we will use the following notations:

- $\mathbf{s}_0$ is the start state and $\mathcal{G}$ is the goal region.
- $c : \mathcal{E} \rightarrow [0, \infty]$ is the cost associated with an edge.
- $g(\mathbf{s})$ or g-value is the cost of the best path to $\mathbf{s}$ from $\mathbf{s}_0$ found by the algorithm so far.

- $h(\mathbf{s})$ is a consistent and therefore admissible heuristic (Russell 2010). It never overestimates the cost to the goal.

A path π is defined by an ordered sequence of edges $e_{i=1}^{N} = (\mathbf{s}, \mathbf{a})_{i=1}^{N}$, the cost of which is denoted as $c(\pi) = \sum_{i=1}^{N} c(e_i)$. The objective is to find a path π from $\mathbf{s}_0$ to a state in the goal region $\mathcal{G}$ with the optimal cost c^*. There is a computational budget of N_t threads available, which can run in parallel. Similar to PA*SE, we assume there exists a pairwise heurisitic function $h(\mathbf{s}, \mathbf{s}')$ that provides an estimate of the cost between any pair of states. It is forward-backward consistent i.e. $h(\mathbf{s}, \mathbf{s}'') \leq h(\mathbf{s}, \mathbf{s}') + h(\mathbf{s}', \mathbf{s}'') \; \forall \; \mathbf{s}, \mathbf{s}', \mathbf{s}''$ and $h(\mathbf{s}, \mathbf{s}') \leq c^*(\mathbf{s}, \mathbf{s}') \; \forall \; \mathbf{s}, \mathbf{s}'$. Note that using h for both the unary heuristic $h(\mathbf{s})$ and the pairwise heuristic $h(\mathbf{s}, \mathbf{s}')$ is a slight abuse of notation, since these are different functions.

Method

ePA*SE leverages the key algorithmic contribution of PA*SE i.e. parallel expansions of independent states but instead uses it to parallelize edge evaluations. In doing so, ePA*SE further improves the efficiency of PA*SE in domains with expensive to evaluate edges. ePA*SE obeys the same invariant as A* and PA*SE that when a state is expanded, its g-value is optimal. Therefore, every state is expanded at most once. However, unlike in PA*SE, where each thread is responsible for expanding a single state at a time, each ePA*SE thread is responsible for evaluating a single edge at a time. In order to build up to ePA*SE, we first describe a serial version of the proposed algorithm eA* (Edge-based A*). We then explain how eA* can be parallelized to get to ePA*SE, using the key idea behind PA*SE.

eA* The first key algorithmic difference in eA* as compared to A* is that the open list *OPEN* contains edges instead of states. We introduce the term *expansion of an edge* and explicitly differentiate it from the expansion of a state. In A*, during the expansion of a state, all its successors are generated and, unless they have already been expanded, are either inserted into the open list or repositioned with the updated priority. In eA*, expansion of an edge $(\mathbf{s}, \mathbf{a})$ involves evaluating the edge to generate the successor state $\mathbf{s}'$ and adding/updating (but not evaluating) the edges originating from $\mathbf{s}'$ into *OPEN* with the same priority of $g(\mathbf{s}') + h(\mathbf{s}')$. This choice of priority ensures that the edges originating from states that would have the same (state-) expansion priority in A* have the same (edge-) expansion priority in eA*. A state is defined as *partially expanded* if at least one (but not all) of its outgoing edges has been expanded or is under expansion, while it is defined as *expanded* if all its outgoing edges have been expanded. eA* uses the following data structures as the key ingredients of the algorithm.

- *OPEN*: A priority queue of edges (not states) that the search has generated but not expanded, where the edge with the smallest key/priority is placed in the front of the queue. The priority of an edge $e = (\mathbf{s}, \mathbf{a})$ in *OPEN* is $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + h(\mathbf{s})$.
- *BE*: The set of states that are partially expanded.
- *CLOSED*: The set of states that have been expanded.

Naively storing edges instead of states in *OPEN* introduces an inefficiency. In A*, the g-value of a state $\mathbf{s}$ can change many times during the search until the state is expanded at which point it is added to *CLOSED*. Every time this happens, *OPEN* has to be rebalanced to reposition $\mathbf{s}$. In eA*, every time $g(\mathbf{s})$ changes, the position of all of the outgoing edges from $\mathbf{s}$ need to be updated in *OPEN*. This increases the number of times *OPEN* has to be rebalanced, which is an expensive operation. However, since the edges originating from $\mathbf{s}$ have the same priority i.e. $g(\mathbf{s}) + h(\mathbf{s})$, this can be avoided by replacing all the outgoing edges from $\mathbf{s}$ by a single *dummy* edge $e^d = (\mathbf{s}, \mathbf{a^d})$, where $\mathbf{a^d}$ stands for a dummy action. The dummy edge stands as a placeholder for all the *real* edges originating from $\mathbf{s}$. Every time $g(\mathbf{s})$ changes, only the dummy edge has to be repositioned. Unlike what happens when a real edge is expanded, when the dummy edge $(\mathbf{s}, \mathbf{a^d})$ is expanded, it is replaced by the outgoing real edges from $\mathbf{s}$ in *OPEN*. When a state's dummy edge is expanded or is under expansion, it is also considered to be partially expanded and is therefore added to *BE*. When all the outgoing real edges of a state have been expanded, it is moved from *BE* to *CLOSED*. The g-value $g(\mathbf{s})$ of a state $\mathbf{s}$ in either *BE* or *CLOSED* can no longer change and hence the real edges originating from $\mathbf{s}$ will never have to be updated in *OPEN*.

eA* can be trivially extended to handle an inflation factor on the heuristic like wA* which leads to a more goal-directed search w-eA* (Weighted eA*). Fig. 1 show an example of w-eA* in action. Let e_i^j refer to an edge from state $\mathbf{s}_i$ to $\mathbf{s}_j$ and e_i^d refer to a dummy edge from $\mathbf{s}_i$. The states that are generated are shown in solid circles. The hollow circles represent states that are not generated and hence the incoming edges to these states are not evaluated. During the first expansion, the dummy edge e_0^d originating from $\mathbf{s}_0$ is expanded and the real edges $[e_0^1, e_0^2, e_0^3]$ are inserted into *OPEN*. In the second expansion, the edge e_0^1 is expanded, during which it is evaluated and the successor $\mathbf{s}_1$ is generated. A dummy edge (e_1^d) from $\mathbf{s}_1$ is inserted into *OPEN*. In the third expansion, e_1^d is expanded and the real edges $[e_1^4, e_1^5]$ are inserted into *OPEN*. In the fourth expansion, the edge e_1^4 is expanded, during which it is evaluated and the successor $\mathbf{s}_4$ is generated and a dummy edge e_4^d is inserted into open. This goes on until a dummy edge e_n^d is expanded whose source state belongs to the goal region i.e. $\mathbf{s}_n \in \mathcal{G}$.

If the heuristic is informative, w-eA* evaluates fewer edges than wA*. In the example shown in Fig. 1, the edges $[e_0^2, e_0^3, e_1^5]$ do not get evaluated. Since wA* evaluates all outgoing edges of an expanded state, these edges would be evaluated in the case of wA* (with the same heuristic and inflation factor) when their source states are expanded ($\mathbf{s}_0$ and $\mathbf{s}_1$). Additionally, similar to how wPA*SE parallelizes wA*, w-eA* can be parallelized to obtain a highly efficient algorithm w-ePA*SE. Since w-ePA*SE is a trivial extension of ePA*SE, we instead describe how eA* can be parallelized to obtain ePA*SE.

eA* to ePA*SE eA* can be parallelized using the key idea behind PA*SE i.e. parallel expansion of independent states, and applying it to edge expansions, resulting in ePA*SE.

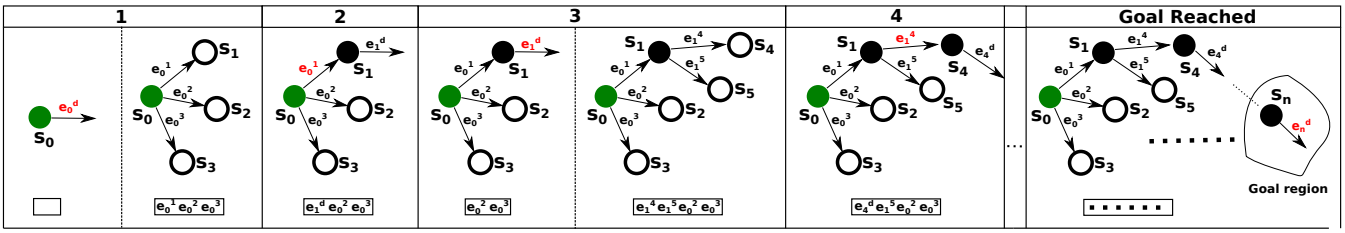

Figure 1: Example of eA*: (1) The dummy edge e_0^d originating from $\mathbf{s}_0$ is expanded and the real edges $[e_0^1, e_0^2, e_0^3]$ are inserted into *OPEN*. (2) e_0^1 is expanded, during which it is evaluated and the successor $\mathbf{s}_1$ is generated. A dummy edge e_1^d from $\mathbf{s}_1$ is inserted into *OPEN*. (3) e_1^d is expanded and the real edges $[e_1^4, e_1^5]$ are inserted into *OPEN*. (4) e_1^4 is expanded, during which it is evaluated and the successor $\mathbf{s}_4$ is generated and a dummy edge e_4^d is inserted into open. This goes on until a dummy edge e_n^d is expanded whose source state belongs to the goal region i.e. $\mathbf{s}_n \in \mathcal{G}$.

ePA*SE has two key differences from PA*SE that makes it more efficient:

1. Evaluation of edges is decoupled from the expansion of the source state giving the search the flexibility to figure out what edges need to be evaluated.
2. Evaluation of edges is parallelized.

In addition to *OPEN* and *CLOSED*, PA*SE uses another data structure *BE* (Being Expanded) to store the set of states currently being expanded by one of the threads. It uses a pairwise independence check on states in the open list to find states that are safe to expand in parallel. A state $\mathbf{s}$ is safe to expand if $g(\mathbf{s})$ is already optimal. In other words, there is no other state that is currently being expanded (in *BE*), nor in *OPEN* that can reduce $g(\mathbf{s})$. Formally, a state $\mathbf{s}$ is defined to be independent of state $\mathbf{s}$' iff

$$g(\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', \mathbf{s}) \tag{1}$$

It can be proved that $\mathbf{s}$ is independent of states in *OPEN* that have a larger priority than $\mathbf{s}$ (Phillips, Likhachev, and Koenig 2014). However, the independence check has to be performed against the states in *OPEN* with a smaller priority than $\mathbf{s}$ as well as the states that are in *BE*.

Like in eA*, *BE* in ePA*SE stores the states that are partially expanded, as per the definition of partial expansion in eA*. Since ePA*SE stores edges in *OPEN* instead of states and each ePA*SE thread expands edges instead of states, the independence check has to be modified. An edge e is safe to expand if Equations 2 and 3 hold.

$$\begin{aligned} g(e.\mathbf{s}) - g(e'.\mathbf{s}) &\leq h(e'.\mathbf{s}, e.\mathbf{s}) \\ \forall e' \in \textit{OPEN} \mid f(e') &< f(e) \end{aligned} \tag{2}$$

$$g(e.\mathbf{s}) - g(\mathbf{s}') \leq h(\mathbf{s}', e.\mathbf{s}) \; \forall \mathbf{s}' \in \textit{BE} \tag{3}$$

Equation 2 ensures that there is no edge in *OPEN* with a priority smaller than that of e, that upon expansion, can lower the g-value of $e.\mathbf{s}$ and hence lower the priority of e. In other words, the source state $\mathbf{s}$ of edge e is independent of the source states of all edges in *OPEN* which have a smaller priority than e. Equation 3 ensures that there is no partially expanded state which can lower the g-value of $e.\mathbf{s}$. In other words, the source state $\mathbf{s}$ of edge e is independent of all states in *BE*.

Details The pseudocode for ePA*SE is presented in Alg. 1. The main planning loop in PLAN runs on a single thread (thread 0), and in Line 13, an edge is removed for expansion from *OPEN* that has the smallest possible priority and is also safe to expand, as per Equations 2 and 3. If such an edge is not found, the thread waits for either *OPEN* or *BE* to change in Line 16. If a safe to expand edge is found, such that the source state of the edge belongs to the goal region, the solution path is returned by backtracking from the state to the start state using back pointers (like in A*) in Line 22. Otherwise, the edge is expanded assigned to an edge expansion thread (thread $i = 1 : N_t$) in Line 30. The edge expansion threads are spawned as and when needed to avoid the overhead of running unused threads (Line 29). The search terminates when either a solution is found, or when *OPEN* is empty and all threads are idle (*BE* is empty), in which case there is no solution.

If the edge to be expanded is a dummy edge, the source $\mathbf{s}$ of the edge is marked as partially expanded by adding it to *BE* (Line 41). The real edges originating from $\mathbf{s}$ are added to *OPEN* with the same priority as that of the dummy edge i.e. $g(\mathbf{s}) + h(\mathbf{s})$. If the expanded edge is not a dummy edge, it is evaluated (Line 47) to obtain the successor $\mathbf{s}'$ and the edge cost $c((\mathbf{s}, \mathbf{a}))$. This is the expensive operation that ePA*SE seeks to parallelize, which is why it happens lock-free. If the expanded edge reduces $g(\mathbf{s}')$, the dummy edge originating from $\mathbf{s}'$ is added/updated in *OPEN*. A counter $n\_successors\_generated$ keeps track of the number of outgoing edges that have been expanded for every state. Once all the outgoing edges for a state have been expanded, and hence the state has been expanded, it is removed from *BE* and added to *CLOSED* (Lines 57 and 58).

Thread management In PA*SE, the state expansion threads are spawned at the start and each of them independently pulls out states from the open list to expand. When the number of threads is higher than the number of independent states available for expansion at any point in time, the operating system has an unnecessary overhead of spinning unused threads. This causes the overall performance to go down as the number of unused threads goes up (see Fig. 6 in (Phillips, Likhachev, and Koenig 2014)). Our initial experiments showed that using a similar thread management strategy in ePA*SE leads to a similar degradation in perfor-

Algorithm 1: ePA*SE

```
 1: A ← action space , N_t ← number of threads, G ← ∅
 2: s_0 ← start state , G ← goal region, terminate ← False
 3: procedure PLAN
 4:     ∀s ∈ G, s.g ← ∞, n_successors_generated(s) = 0
 5:     s_0.g ← 0
 6:     insert (s_0, a^d) in OPEN                    ▷ Dummy edge from s_0
 7:     LOCK
 8:     while not terminate do
 9:         if OPEN = ∅ and BE = ∅ then
10:             terminate = True
11:             UNLOCK
12:             return ∅
13:         remove an edge (s, a) from OPEN that has the
            smallest f((s, a)) among all states in OPEN that
            satisfy Equations 2 and 3
14:         if such an edge does not exist then
15:             UNLOCK
16:             wait until OPEN or BE change
17:             LOCK
18:             continue
19:         if s ∈ G then
20:             terminate = True
21:             UNLOCK
22:             return BACKTRACK(s)
23:         else
24:             UNLOCK
25:             while (s, a) has not been assigned a thread do
26:                 for i = 1 : N_t do
27:                     if thread i is available then
28:                         if thread i has not been spawned then
29:                             Spawn EDGEEXPANDTHREAD(i)
30:                         Assign (s, a) to thread i
31:             LOCK
32:     terminate = True
33:     UNLOCK
34: procedure EDGEEXPANDTHREAD(i)
35:     while not terminate do
36:         if thread i has been assigned an edge (s, a) then
37:             EXPAND ((s, a))
38: procedure EXPAND((s, a))
39:     LOCK
40:     if a = a^d then
41:         insert s in BE
42:         for a ∈ A do
43:             f ((s, a)) = g(s) + h(s)
44:             insert (s, a) in OPEN with f ((s, a))
45:     else
46:         UNLOCK
47:         s', c((s, a)) ← GENERATESUCCESSOR ((s, a))
48:         LOCK
49:         if s' ∉ CLOSED ∪ BE and
50:         g(s') > g(s) + c((s, a)) then
51:             g(s') = g(s) + c((s, a))
52:             s'.parent = s
53:             f((s', a^d)) = g(s') + h(s')
54:             insert/update (s', a^d) in OPEN with f((s', a^d))
55:         n_successors_generated(s)+ = 1
56:         if n_successors_generated(s) = |A| then
57:             remove s from BE
58:             insert s in CLOSED
59:     UNLOCK
```

mance as the number of threads is increased beyond the optimal number of threads, even though the peak performance of ePA*SE is substantially higher than that of PA*SE. In order to prevent this degradation in performance, ePA*SE employs a different thread management strategy. There is a single thread that pulls out edges from the open list and it spawns edge expansion threads as needed but capped at N_t (Line 29). When N_t is higher than the number of independent edges available for expansion at any point in time, only a subset of available threads get spawned preventing performance degradation, as we will show in our experiments.

w-ePA*SE w-ePA*SE is a bounded suboptimal variant of ePA*SE that trades off optimality for faster planning. Similar to wPA*SE, w-ePA*SE introduces two inflation factors, the first of which, $\epsilon \geq 1$, relaxes the independence rule (Equations 2 and 3) as follows.

$$\begin{aligned} g(e.\mathbf{s}) - g(e'.\mathbf{s}) &\leq \epsilon h(e'.\mathbf{s}, e.\mathbf{s}) \\ \forall e' \in \textit{OPEN} \mid f(e') &< f(e) \end{aligned} \tag{4}$$

$$g(e.\mathbf{s}) - g(\mathbf{s}') \leq \epsilon h(\mathbf{s}', e.\mathbf{s}) \; \forall \mathbf{s}' \in \textit{BE} \tag{5}$$

The second factor $w \geq 1$ is used to inflate the heuristic in the priority of edges in *OPEN* i.e. $f((\mathbf{s}, \mathbf{a})) = g(\mathbf{s}) + w \cdot h(\mathbf{s})$ which makes the search more goal directed. As long as $\epsilon \geq w$, the solution cost is bounded by $\epsilon \cdot c^*$ (Theorem 3). Note that w can be greater than ϵ, but then Equation 4 has to consider source states of all edges in *OPEN* and the solution cost will be bounded by $w \cdot c^*$ (Theorem 1). Since this leads to significantly more independence checks, the $\epsilon \geq w$ relationship is typically recommended in practice.

Properties

w-ePA*SE has identical properties to that of wPA*SE (Phillips, Likhachev, and Koenig 2014) and can be proved similarly with minor modifications.

Theorem 1 (Bounded suboptimal expansions) *When w-ePA*SE that performs independence checks against all states in BE and source states of **all** edges in OPEN, chooses an edge e for expansion, then $g(e.\mathbf{s}) \leq \lambda g^*(\mathbf{s})$, where $\lambda = \max(\epsilon, w)$.*

Proof Assume, for the sake of contradiction, that $g(e.\mathbf{s}) > \lambda g^*(e.\mathbf{s})$ directly before edge e is expanded, and without loss of generality, that $g(e'.\mathbf{s}) \leq \lambda g^*(e'.\mathbf{s})$ for all edges e' selected for expansion before e (**Assumption**). Consider any cost-minimal path $\pi(\mathbf{s}_0, \mathbf{s})$ from $\mathbf{s}_0$ to $\mathbf{s}$. Let $\mathbf{s}_m$ be the closest state to $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ such that either 1) there exists atleast one edge in *OPEN* with source state $\mathbf{s}_m$ or 2) $\mathbf{s}_m$ is in *BE*. $\mathbf{s}_m$ is no farther away from $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ than $\mathbf{s}$ since $\mathbf{s}$ is in *OPEN*. Therefore, let $\pi(\mathbf{s}_0, \mathbf{s}_m)$ and $\pi(\mathbf{s}_m, \mathbf{s})$ be the subpaths of $\pi(\mathbf{s}_0, \mathbf{s})$ from $\mathbf{s}_0$ to $\mathbf{s}_m$ and from $\mathbf{s}_m$ to $\mathbf{s}$, respectively.

If $\mathbf{s}_m = \mathbf{s}_0$, then $g(\mathbf{s}_m) \leq \lambda g^*(\mathbf{s}_m)$ since $g(\mathbf{s}_0) = g^*(\mathbf{s}_0) = 0$ (**Contradiction 1**).

Otherwise, let $\mathbf{s}_p$ be the predecessor of $\mathbf{s}_m$ on $\pi(\mathbf{s}_0, \mathbf{s})$. $\mathbf{s}_p$ has been expanded (i.e. all edges outgoing edges of $\mathbf{s}_p$ have

been expanded) since every state closer to $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ that $\mathbf{s}_m$ has been expanded (since every unexpanded state on $\pi(\mathbf{s}_0, \mathbf{s})$ different from $\mathbf{s}_0$ is either in *BE* or has an outgoing edge in *OPEN*, or has a state closer to $\mathbf{s}_0$ on $\pi(\mathbf{s}_0, \mathbf{s})$ that is either in *BE* or has an outgoing edge in *OPEN*). Therefore, since all outgoing edges from $\mathbf{s}_p$ have been expanded, $g(\mathbf{s}_p) \leq \lambda g^*(\mathbf{s}_p)$ because of **Assumption**. Then, because of the g update of $\mathbf{s}_m$ when the edge from $\mathbf{s}_p$ to $\mathbf{s}_m$ was expanded,

$$\begin{aligned} g(\mathbf{s}_m) &\leq g(\mathbf{s}_p) + c(\mathbf{s}_p, \mathbf{s}_m) \\ &\leq \lambda g^*(\mathbf{s}_p) + c(\mathbf{s}_p, \mathbf{s}_m) \end{aligned} \tag{6}$$

Since $\mathbf{s}_p$ is the predecessor of $\mathbf{s}_m$ on the cost-minimal path $\pi(\mathbf{s}_0, \mathbf{s})$,

$$\begin{aligned} & g^*(\mathbf{s}_m) = g^*(\mathbf{s}_p) + c(\mathbf{s}_p, \mathbf{s}_m) \\ \implies & g^*(\mathbf{s}_p) = g^*(\mathbf{s}_m) - c(\mathbf{s}_p, \mathbf{s}_m) \end{aligned} \tag{7}$$

Substituting $g^*(\mathbf{s}_p)$ from Equation 7 into Equation 6

$$\begin{aligned} & \implies g(\mathbf{s}_m) \leq \lambda g^*(\mathbf{s}_m) - (\lambda - 1) c(\mathbf{s}_p, \mathbf{s}_m) \\ & \implies g(\mathbf{s}_m) \leq \lambda g^*(\mathbf{s}_m) \\ & \implies \lambda c(\pi(\mathbf{s}_0, \mathbf{s}_m)) = \lambda g^*(\mathbf{s}_m) \geq g(\mathbf{s}_m) \end{aligned} \tag{8}$$

Since $h(\mathbf{s}_m, \mathbf{s})$ satisfies forward-backward consistency and is therefore admissible, $h(\mathbf{s}_m, \mathbf{s}) \leq c(\pi(\mathbf{s}_m, \mathbf{s}))$. Since, $\lambda = \max(\epsilon, w)$, $\epsilon \leq \lambda$. Therefore,

$$\lambda c(\pi(\mathbf{s}_m, \mathbf{s})) \geq \lambda h(\mathbf{s}_m, \mathbf{s}) \geq \epsilon h(\mathbf{s}_m, \mathbf{s}) \tag{9}$$

Adding 8 and 9,

$$\begin{aligned} \lambda c(\mathbf{s}_0, \mathbf{s}) &= \lambda c(\mathbf{s}_0, \mathbf{s}_m) + \lambda c(\mathbf{s}_m, \mathbf{s}) \\ &\geq g(\mathbf{s}_m) + \epsilon h(\mathbf{s}_m, \mathbf{s}) \end{aligned} \tag{10}$$

Assuming w-ePA*SE performs independence checks against states in *BE* and source states of all edges in *OPEN* when choosing an edge e with source $e.\mathbf{s}$ to expand, and $\mathbf{s}_m$ is either in *BE* or there exists atleast one edge with source $\mathbf{s}_m$ in *OPEN*,

$$\begin{aligned} & \epsilon h(e'.\mathbf{s}, e.\mathbf{s}) \geq g(e.\mathbf{s}) - g(e'.\mathbf{s}) \\ & \forall e' \in \textit{OPEN} \mid e'.\mathbf{s} = \mathbf{s}_m \\ \implies & g(\mathbf{s}_m) + \epsilon h(\mathbf{s}_m, \mathbf{s}) \geq g(\mathbf{s}) \end{aligned} \tag{11}$$

Therefore,

$$\begin{aligned} \lambda g^*(\mathbf{s}) =& \lambda c(\pi(\mathbf{s}_0, \mathbf{s})) \\ \geq & g(\mathbf{s}_m) + \epsilon h(\mathbf{s}_m, \mathbf{s}) && \text{(Using Eq. 10)} \\ \geq & g(\mathbf{s}) && \text{(Using Eq. 11)} \end{aligned}$$

(**Contradiction 2**)

Contradiction 1 and **Contradiction 2** invalidate the **Assumption**, which proves **Theorem 1**.

Theorem 2 *If $w \leq \epsilon$, and considering any two edges e and e' in OPEN, the source state of e is independent of the source state of e' if $f(e) \leq f(e')$.*

Proof

$$\begin{aligned} & f(e) \leq f(e') \\ \implies & g(e.\mathbf{s}) + wh(e.\mathbf{s}) \leq g(e'.\mathbf{s}) + wh(e'.\mathbf{s}) \\ \implies & g(e.\mathbf{s}) \leq g(e'.\mathbf{s}) + w(h(e'.\mathbf{s}) - h(e.\mathbf{s})) \\ & \leq g(e'.\mathbf{s}) + wh(e'.\mathbf{s}, e.\mathbf{s}) \\ & \text{(forward-backward consistency)} \\ & \leq g(e'.\mathbf{s}) + \epsilon h(e'.\mathbf{s}, e.\mathbf{s}) \\ & \text{(since } w \leq \epsilon\text{)} \end{aligned}$$

Therefore, $e.\mathbf{s}$ is independent of $e'.\mathbf{s}$ by definition (Eq. 1).

Theorem 3 (Bounded suboptimality) *If $w \leq \epsilon$, and w-ePA*SE chooses a dummy edge $e^d = (\mathbf{s}, \mathbf{a^d})$ for expansion, such that the source state $\mathbf{s}$ belongs to the goal region i.e. $\mathbf{s} \in \mathcal{G}$, then $g(\mathbf{s}) \leq \epsilon g^*(\mathbf{s}) = \epsilon \cdot c^*$.*

Proof This directly follows from Theorems 1 and 2.

Theorem 4 (Completeness) *If there exists at least one path π in G from $\mathbf{s}_0$ to $\mathcal{G}$, w-ePA*SE will find it.*

Proof This proof makes use of Theorem 3 and is similar to the equivalent proof of serial wA*.

Evaluation

We evaluate w-ePA*SE in two planning domains where edge evaluation is expensive. All experiments were carried out on Amazon Web Services (AWS) instances. All algorithms were implemented in C++.

3D Navigation

The first domain is motion planning for 3D (x, y, θ) navigation of a PR2, which is a human-scale dual-arm mobile manipulator robot, in an indoor environment similar to the one used in (Narayanan and Likhachev 2017) and shown in Fig. 2. Here x, y are the planar coordinates and θ is the orientation of the robot. The robot can move along 18 simple motion primitives that independently change the three state coordinates by incremental amounts. Evaluating each primitive involves collision checking of the robot model (approximated as spheres) against the world model (represented as a 3D voxel grid) at interpolated states on the primitive. Though approximating the robot with spheres instead of meshes speeds up collision checking, it is still the most expensive component of the search. The computational cost of edge evaluation increases with an increasing granularity of interpolated states at which collision checking is carried out. For our experiments, collision checking is carried out at interpolated states 1 cm apart. The search uses Euclidean distance as the admissible heuristic. We evaluate on 50 trials in each of which the start configuration of the robot and goal region are sampled randomly. We compare w-ePA*SE with other CPU-based parallel search baselines. The first baseline is a variant of weighted A* in which during a state expansion, the successors of the state are generated and the corresponding edges are evaluated in parallel. For lack of a better term, we call this baseline Parallel Weighted A* (PwA*). Note that this is very different from the Parallel A* (PA*) algorithm (Irani and Shih 1986) which has already

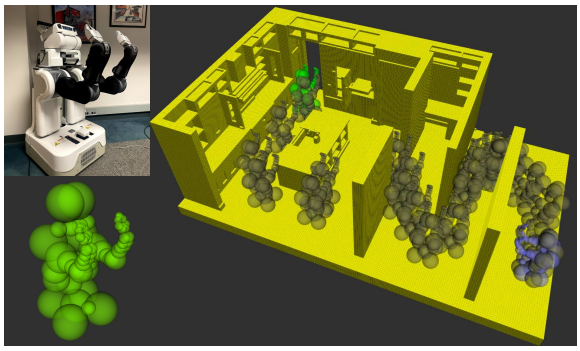

Figure 2: (Navigation) Left: The PR2's collision model is approximated with spheres. Right: The task is to navigate in an indoor map from a given start (purple) and goal (green) states using a set of motion primitives. States at the end of every primitive in the generated plan are shown in black.

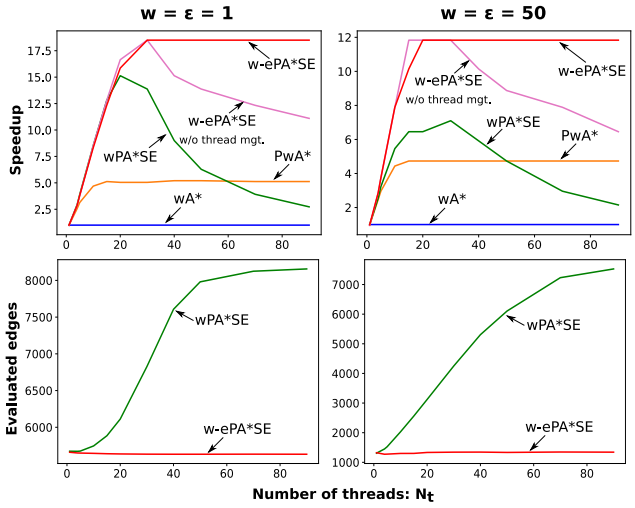

Figure 3: (Navigation) Top: Average speedup achieved by PwA*, wPA*SE and w-ePA*SE over wA*. Bottom: Number of edges evaluated by wPA*SE and w-ePA*SE.

been shown to underperform wPA*SE (Phillips, Likhachev, and Koenig 2014). The second baseline is wPA*SE. These two baselines leverage parallelization differently. PwA* parallelizes the generation of successors, whereas wPA*SE parallelizes state expansions. w-ePA*SE on the other hand parallelizes edge evaluations. We also compare against a variation of w-ePA*SE (w-ePA*SE w/o thread mgt.) that uses the thread management strategy of wPA*SE as opposed to the improved thread management strategy described in the Method. Speedup over wA* is defined as the ratio of the average runtime of wA* over the average runtime of a specific algorithm.

Fig. 3 (top) shows the average speedup achieved by wPA*SE and the baselines over wA* for varying N_t, for $w = \epsilon = 1$ and $w = \epsilon = 50$. The corresponding raw planning times are shown in Table 1. The speedup achieved by PwA* saturates at the branching factor of the domain. This is expected since PwA* parallelizes the evaluation of the outgoing edges of a state being expanded. If N_t is greater than the branching factor M, $N_t - M$ threads remain unutilized. For low N_t, the speedup achieved by w-ePA*SE matches that of wPA*SE. However, for high N_t, the speedup achieved by w-ePA*SE rapidly outpaces that of w-ePA*SE, especially for the inflated heuristic case. This is because w-ePA*SE is much more efficient than wPA*SE since it parallelizes edge evaluations instead of state expansions. This increased efficiency is more apparent with the availability of a larger computational budget in the form of a greater number of threads to allocate to evaluating edges. The speedup of wPA*SE reaches a peak and then rapidly deteriorates. This is also the case for w-ePA*SE w/o (improved) thread mgt. (described in the Method), even though the peak speedup of w-ePA*SE w/o thread mgt.is higher than that of wPA*SE. However, the speedup of w-ePA*SE with the improved thread management strategy reaches a maximum and then saturates instead of degrading. This is due to the difference in the multithreading strategy employed by w-ePA*SE as explained in the Method.

Fig. 3 (bottom) and Table 2 show that w-ePA*SE evaluates significantly fewer edges as compared to wPA*SE. With a greater number of threads, the difference is significant. This indicates that beyond parallelization of edge evaluations, the w-eA* formulation that w-ePA*SE uses has another advantage that if the heuristic is informative, w-ePA*SE evaluates fewer edges than wPA*SE, which contributes to the lower planning time of w-ePA*SE. The intuition behind this is that in wA*, the evaluation of the outgoing edges from a given state is tightly coupled with the expansion of the state because all the outgoing edges from a given state must be evaluated at the same time when the state is expanded. In w-eA* the evaluation of these edges is decoupled from each other since the search expands edges instead of states.

Assembly Task

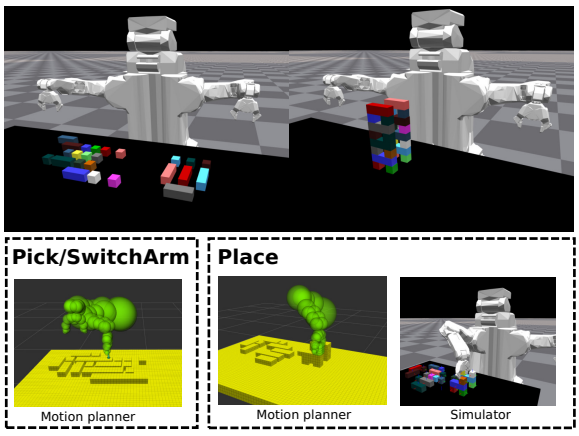

Figure 4: (Assembly) Top: The PR2 has to arrange a set of blocks on the table (left) into a given configuration (right). Bottom: It is equipped with PICK, PLACE and SWITCHARM controllers.

N_t	1	4	5	10	15	20	30	40	50	70	90	
wA*	3.33	-	-	-	-	-	-	-	-	-	-	
PwA*	3.37	1.31	1.06	0.71	0.65	0.66	0.66	0.64	0.64	0.65	0.65	
wPA*SE	3.37	1.14	0.85	0.39	0.26	0.22	0.24	0.37	0.53	0.85	1.22	$w = \epsilon = 1$
w-ePA*SE w/o thread mgt.	3.43	1.17	0.88	0.39	0.26	0.20	0.18	0.22	0.24	0.27	0.30	
w-ePA*SE	3.34	1.17	0.87	0.40	0.27	0.21	0.18	0.18	0.18	0.18	0.18	
wA*	0.71	-	-	-	-	-	-	-	-	-	-	
PwA*	0.72	0.29	0.24	0.16	0.15	0.15	0.15	0.15	0.15	0.15	0.15	
wPA*SE	0.71	0.28	0.22	0.13	0.11	0.11	0.10	0.12	0.15	0.24	0.33	$w = \epsilon = 50$
w-ePA*SE w/o thread mgt.	0.75	0.25	0.19	0.09	0.06	0.06	0.06	0.07	0.08	0.09	0.11	
w-ePA*SE	0.72	0.25	0.19	0.09	0.07	0.06	0.06	0.06	0.06	0.06	0.06	

Table 1: (Navigation) Average planning times (s) for wA*, PwA*, wPA*SE and w-ePA*SE for varying N_t, with $w = \epsilon = 1$ (top) and with $w = \epsilon = 50$ (bottom).

N_t	1	4	5	10	15	20	30	40	50	70	90	
wPA*SE	5674	5673	5676	5746	5885	6112	6826	7607	7980	8125	8156	$w = \epsilon = 1$
w-ePA*SE	5660	5650	5649	5645	5640	5637	5634	5633	5633	5634	5633	
wPA*SE	1309	1451	1526	2028	2561	3121	4251	5307	6105	7231	7526	$w = \epsilon = 50$
w-ePA*SE	1324	1273	1277	1300	1301	1333	1343	1345	1334	1348	1343	

Table 2: (Navigation) Number of edges evaluated by wPA*SE and w-ePA*SE for varying N_t, with $w = \epsilon = 1$ (top) and with $w = \epsilon = 50$ (bottom).

	wA*	PwA*	wPA*SE	w-ePA*SE
N_t	1	25	10	10
Time (s)	3010	1066	419	301
Speedup	1	2.8	7.2	10

Table 3: (Assembly) Average planning times and speedup over wA* for w-ePA*SE and the baselines.

The second domain is a task and motion planning problem of assembling a set of blocks on a table into a given structure by a PR2, as shown in Fig. 4. This domain is similar to the one introduced in (Mukherjee, Aine, and Likhachev 2022), but in this work, we enable the dual-arm functionality of the PR2. We assume full state observability of the 6D poses of the blocks and the robot's joint configuration. The goal is defined by the 6D poses of each block in the desired structure. The PR2 is equipped with PICK and PLACE controllers which are used as macro-actions in the high-level planning. In addition, there is a SWITCHARM controller which switches the active arm by moving the current active arm to a home position. All of these actions use a motion planner internally to compute collision-free trajectories in the workspace. Additionally, PLACE has access to a simulator (NVIDIA Isaac Gym (Makoviychuk et al. 2021)) to simulate the outcome of placing a block at its desired pose. For example, if the planner tries to place a block at its final pose but has not placed the block underneath yet, the placed block will not be supported and the structure will not be stable. This would lead to an invalid successor during planning. We set a simulation timeout of $t_s = 0.2$ s to evaluate the outcome of placing a block. Considering the variability in the simulation speed and the overhead of communicating with the simulator, this results in a total wall time of less than 2 s for the simulation. The motion planner has a timeout of $t_p = 60$ s based on the wall time, and therefore that is the maximum time the motion planning can take. Successful PICK, PLACE and SWITCHARM actions have unit costs, and infinite otherwise. A PICK action on a block is successful if the motion planner finds a feasible trajectory to reach the block within t_p. A PLACE action on a block is successful if the motion planner finds a feasible trajectory to place the block within t_p and simulating the block placement results in the block coming to rest at the desired pose within t_s. A SWITCHARM action is successful if the motion planner finds a feasible trajectory to the home position for the active arm within t_p. The number of blocks that are not in their final desired pose is used as the admissible heuristic, with $w = \epsilon = 5$. Table 3 shows planning times and speedup over wA* for w-ePA*SE and those of the baselines. We use 25 threads in the case of PwA* because that is the maximum branching factor in this domain. The numbers are averaged across 20 trials in each of which the blocks are arranged in random order on the table. Table 3 shows the average planning times and speedup over wA* of w-ePA*SE as compared to those of the lazy search baselines. w-ePA*SE achieves a 10x speedup over wA* and outperforms the baselines in this domain as well.

Conclusion and Future Work

We presented an optimal parallel search algorithm ePA*SE, that improves on PA*SE by parallelizing edge evaluations instead of state expansions. We also presented a suboptimal variant w-ePA*SE and proved that it maintains

bounded suboptimality guarantees. Our experiments showed that w-ePA*SE achieves an impressive reduction in planning time across two very different planning domains, which shows the generalizability of our conclusions. Empirically, we have observed w-ePA*SE to be a strict improvement over wPA*SE for domains with expensive to compute edges. Even though we also test with a relatively large budget of threads, the performance improvement is significant even with a smaller budget of fewer than 10 threads, which is the case with typical mobile computers. Therefore in practice, we recommend using w-ePA*SE in a planning domain where 1) the computational bottleneck is edge evaluations and 2) successor states cannot be generated without evaluating edges and therefore parallelized lazy planning i.e. MPLP (Mukherjee, Aine, and Likhachev 2022) is not applicable.

MPLP (Mukherjee, Aine, and Likhachev 2022) and ePA*SE use fundamentally different parallelization strategies. MPLP searches the graph lazily while evaluating edges in parallel, but relies on the assumption that states can be generated lazily without evaluating edges. On the other hand, ePA*SE evaluates edges in a way that preserves optimality without the need for state (and edge) re-expansions but does not rely on lazy state generation. In domains where states can be generated lazily, however, the lazy state generation takes a non-trivial amount of time, these two different parallelization strategies can be combined. Both MPLP and ePA*SE achieve a speedup at a certain number of threads, beyond which the speedup saturates. Therefore, utilizing both of them together will allow us to leverage more threads, which either of them on their own cannot.

Acknowledgements

This work was supported by the ARL-sponsored A2I2 program, contract W911NF-18-2-0218, and ONR grant N00014-18-1-2775.

References

Aine, S.; Swaminathan, S.; Narayanan, V.; Hwang, V.; and Likhachev, M. 2016. Multi-heuristic A*. *The International Journal of Robotics Research*, 35(1-3): 224–243.

Amato, N. M.; and Dale, L. K. 1999. Probabilistic roadmap methods are embarrassingly parallel. In *Proceedings 1999 IEEE International Conference on Robotics and Automation*, volume 1, 688–694.

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39: 689–743.

Butzke, J.; Sapkota, K.; Prasad, K.; MacAllister, B.; and Likhachev, M. 2014. State lattice with controllers: Augmenting lattice-based path planning with controller-based motion primitives. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 258–265.

Devaurs, D.; Siméon, T.; and Cortés, J. 2011. Parallelizing RRT on distributed-memory architectures. In *2011 IEEE International Conference on Robotics and Automation*, 2261–2266.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1995. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25(2): 133–143.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

He, X.; Yao, Y.; Chen, Z.; Sun, J.; and Chen, H. 2021. Efficient parallel A* search on multi-GPU system. *Future Generation Computer Systems*, 123: 35–47.

Ichnowski, J.; and Alterovitz, R. 2012. Parallel sampling-based motion planning with superlinear speedup. In *IROS*, 1206–1212.

Irani, K.; and Shih, Y.-f. 1986. Parallel A* and AO* algorithms- An optimality criterion and performance evaluation. In *1986 International Conference on Parallel Processing, University Park, PA*, 274–277.

Jacobs, S. A.; Manavi, K.; Burgos, J.; Denny, J.; Thomas, S.; and Amato, N. M. 2012. A scalable method for parallelizing sampling-based motion planning algorithms. In *2012 IEEE International Conference on Robotics and Automation*, 2529–2536.

Jacobs, S. A.; Stradford, N.; Rodriguez, C.; Thomas, S.; and Amato, N. M. 2013. A scalable distributed RRT for motion planning. In *2013 IEEE International Conference on Robotics and Automation*, 5088–5095.

Liang, J.; Sharma, M.; LaGrassa, A.; Vats, S.; Saxena, S.; and Kroemer, O. 2021. Search-Based Task Planning with Learned Skill Effect Models for Lifelong Robotic Manipulation. *arXiv preprint arXiv:2109.08771*.

Makoviychuk, V.; Wawrzyniak, L.; Guo, Y.; Lu, M.; Storey, K.; Macklin, M.; Hoeller, D.; Rudin, N.; Allshire, A.; Handa, A.; et al. 2021. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. *arXiv preprint arXiv:2108.10470*.

Mukherjee, S.; Aine, S.; and Likhachev, M. 2022. MPLP: Massively Parallelized Lazy Planning. *IEEE Robotics and Automation Letters*, 7(3): 6067–6074.

Narayanan, V.; and Likhachev, M. 2017. Heuristic search on graphs with existence priors for expensive-to-evaluate edges. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

Park, C.; Pan, J.; and Manocha, D. 2016. Parallel motion planning using poisson-disk sampling. *IEEE Transactions on Robotics*, 33(2): 359–371.

Phillips, M.; Likhachev, M.; and Koenig, S. 2014. PA* SE: Parallel A* for slow expansions. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204.

Russell, S. J. 2010. *Artificial intelligence a modern approach*. Pearson Education, Inc.

Zhou, Y.; and Zeng, J. 2015. Massively parallel A* search on a GPU. In *Proceedings of the AAAI Conference on Artificial Intelligence*.