Runahead $A^\star$: Speculative Parallelism for $A^\star$ with Slow Expansions

Mohammad Bakhshalipour,[1] Mohamad Qadri,[1] Dominic Guri,[1] Seyed Borna Ehsani,[2] Maxim Likhachev,[1] Phillip B. Gibbons[1]

[1] Carnegie Mellon University
[2] University of Washington

Abstract

$A^\star$ suffers from limited parallelism. The maximum level of *traditional parallelism* in $A^\star$ is the same as the degree of the search graph nodes, which is too small in many applications. As such, $A^\star$ cannot fully leverage the multithreading capabilities of modern processors.

In this paper, we go beyond traditional parallelism and introduce *speculative parallelism* for $A^\star$. We observe that $A^\star$'s node expansions exhibit predictable patterns in applications like path planning. Based on this observation, we propose *Runahead $A^\star$* (*$RA^\star$*). When a node is being expanded, *$RA^\star$* predicts future likely-to-be-expanded nodes, performs their corresponding computation on separate threads, and memoizes the computation results. Later when a predicted node is selected for expansion, rather than performing its computation, the memoized results are used, saving significant time in slow-expansion applications.

We study five applications of $A^\star$. We show that when its prediction accuracy is high, *$RA^\star$* offers significant speedup over vanilla $A^\star$ for slow-expansion applications. With 16 threads, *$RA^\star$*'s speedup for such applications ranges from $3.1\times$ to $14.1\times$. We also study and provide insight into when, why, and to what extent node expansions are predictable. We provide an implementation of *$RA^\star$* at: `https://github.com/cmu-roboarch/runahead-astar/`

Introduction

$A^\star$ (Hart, Nilsson, and Raphael 1968) is a widely-used best-first graph search algorithm which takes advantage of a heuristic cost function to guide the search. Given a consistent heuristic, $A^\star$ guarantees optimality and returns a least-cost solution. $A^\star$ is used in many artificial intelligence applications, like path planning, task planning, protein design, and so on, to find a path from a start state to one (or more) goal states.

$A^\star$ maintains an `OPEN` list of candidate nodes for expansion. At every iteration, $A^\star$ expands the node of the search graph whose f value is the lowest in `OPEN`. For node N, $f(N) = g(N) + h(N)$; $g(N)$ being the cost from the start node to node N, and $h(N)$ being the heuristic cost—a non-overestimate of the cost from node N to the goal node. Whenever a node is expanded, its neighbors are "evaluated" and (conditionally) pushed to `OPEN`.

The evaluation of neighbors entails (*i*) validity checking and (*ii*) heuristic cost calculation. For example, in robot path planning, a state's validity is determined by collision detection: a state is valid if the robot at that state does not collide with obstacles. The heuristic cost is calculated based on a heuristic function; e.g., the Euclidean distance between the state and the goal state.

In a number of applications, evaluating neighbors can be extremely time-consuming. We use the terminology of (Phillips, Likhachev, and Koenig 2014) and call those applications "slow-expansion." In such applications, the majority of execution time is spent evaluating neighbors. An example of such an application is robot path planning with full-body collision detection. Expansions in this application are slow because while expanding a node, the planner must ensure that the robot does not collide with any of the possibly-many obstacles in the environment (Phillips, Likhachev, and Koenig 2014). As another example, heuristic cost calculations in protein design are very slow: the function estimates the energy of all the rotamers (sidechains of an amino acid) that are not assigned in the design (Leach and Lemon 1998).

Fortunately, neighbors of a node are mutually independent and can be evaluated in parallel. However, other than this, the vanilla $A^\star$ is serial, as are most of its extensions and variants (Phillips, Likhachev, and Koenig 2014). Unfortunately, in many applications, nodes have only a few neighbors, which means $A^\star$ can utilize at best a few threads for parallel execution. For example, in path planning for a robot that can move in eight cardinal and inter-cardinal directions, the degree of nodes is eight (i.e., 8-connected grid), which means the maximum level of parallelism is eight. The average level of parallelism is even (far) smaller (e.g., ~ 2.3 with our evaluated map; see **§Evaluations**), since not all neighbors of an expanded node are always evaluated (e.g., already-visited neighbors are skipped, etc.). This limited parallelism makes $A^\star$ unable to fully leverage the multi-threading capabilities of modern processors that support the parallel execution of tens or even hundreds of threads.

Expanding multiple nodes in parallel might seem to be a sensible approach to enhance $A^\star$ parallelism. However, it is not straightforward, because the search optimality depends on the node expansion order: if nodes are expanded

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

in any order other than lowest-f-first, optimality cannot be guaranteed. Naively parallelizing multiple expansions could potentially disturb the correct expansion order and sacrifice the optimality. Prior work proposes workarounds to enable parallelizing multiple expansions *safely*, i.e., without losing optimality guarantees. This is done through a variety of approaches, like discovering independent expansions (Phillips, Likhachev, and Koenig 2014), expanding nodes multiple times (Kishimoto, Fukunaga, and Botea 2009), and so forth. Nevertheless, as we will show, these methods impose non-trivial overheads, which severely limits their effectiveness.

In this work, we take a different approach to parallelize $A^\star$, in a proposal named *Runahead* $A^\star$ ($RA^\star$). Instead of expanding multiple nodes in parallel, $RA^\star$ tries to *run ahead of individual expansions and pave the way for accelerating future expansions*. When a node is expanded and its neighbors are *being evaluated*, $RA^\star$ performs three additional tasks, in parallel: (*i*) prediction, (*ii*) pre-evaluation, and (*iii*) memoization.

In detail, $RA^\star$ first predicts future expansions: which nodes are likely to be expanded in the future? Prediction is the focal point of $RA^\star$. We will provide insight into when, why, and to what extent the node expansions are predictable, and how $RA^\star$ predicts them. Then, $RA^\star$ evaluates the predicted nodes' neighbors. These "pre-evaluations" run on separate threads, in parallel with the expanded node's evaluations. This parallelism is *speculative*, meaning that the predictions could be wrong (i.e., the predicted nodes are never expanded). Finally, $RA^\star$ memoizes the pre-evaluation results. Later when a predicted node is actually expanded, instead of performing time-consuming evaluations, the memoized results are used, saving a large amount of time. As we will discuss, $RA^\star$ neither jeopardizes the optimality guarantees of vanilla $A^\star$ nor does it affect the search outcome.

Finally, we evaluate $RA^\star$ for five different applications of $A^\star$, namely robot path planning in (x, y), (x, y, θ), (x, y, z) domains, protein design, and symbolic task planning. We present its speedup, prediction performance, and overheads.

Related Work

Parallelizing $A^\star$

Prior work on parallelizing $A^\star$ can be divided into two groups: methods that need to (conditionally) *re-expand* nodes to guarantee optimality, and methods that do not. In this paper, we will compare $RA^\star$ against both. Also, in this paper, we only consider methods that preserve the optimality guarantees of $A^\star$, as $RA^\star$ does.

The first group of methods (Kishimoto, Fukunaga, and Botea 2009; Burns et al. 2010; Kishimoto, Fukunaga, and Botea 2013; Jinnai and Fukunaga 2016) are effective when expansions are fast; i.e., when evaluations are not costly, which is the case in applications like symbolic task planning. In slow-expansion applications, where evaluations are time-consuming, these methods suffer from significant challenges. Specifically, re-expansions in these methods can happen exponentially-many times, which means the already-slow evaluations can repeat exponentially-many times, hurting performance.

The second group of methods (Irani and Shih 1986; Zhou and Hansen 2007; Phillips, Likhachev, and Koenig 2014) do not require re-expansions to guarantee optimality. As such, they can parallelize and reduce the execution time of slow-expansion applications of $A^\star$. These methods, on the flip side, impose non-trivial overheads that limit their performance. For example, $PA^\star SE$ (Phillips, Likhachev, and Koenig 2014) spends significant time serially searching the states to find the parallelizable ones.

While there are differences in how prior proposals parallelize $A^\star$, most of them have one thing in common: they parallelize multiple expansions. In contrast, $RA^\star$ runs ahead of individual expansions and paves the way for accelerating future expansions; it exploits a different source of parallelism.

Speculative Parallelism

Generally, speculation is a hardware- and system-level technique that uses additional or available-but-idle resources to improve performance. For example, *helper threads* (Lee et al. 2009; Darabi et al. 2022) use otherwise unused threads to optimize memory resources for the running application. Or, *hardware prefetchers* (Somogyi et al. 2009) use additional contexts to predict future memory references of the running application and fetch them from slow memory to fast memory (e.g., from DRAM to on-chip caches); this way, when predicted data are requested, the application enjoys the low latency of the fast memory.

In this paper, we draw insight from such system-level techniques and propose an algorithm-level speculative approach for parallelizing $A^\star$. Throughout the execution, when a node is being expanded, we predict future expansions, pre-evaluate their neighbors on separate threads (i.e., in parallel), and memoize the results. This way, when a predicted node is expanded, we use the memoized results rather than performing time-consuming evaluations, saving significant time.

Definitions, Notations, and Scope

Following, we define some terms that we use throughout the paper:

Evaluation: The process of (*i*) checking whether a state is valid (e.g., collision detection in robot path planning), and (*ii*) calculating its heuristic cost. In slow-expansion applications of $A^\star$, evaluations take the majority of execution time.

Pre-Evaluation: Evaluation done by $RA^\star$, ahead of time.

Expansion: The process of (*i*) picking the node with the lowest f from `OPEN`, (*ii*) checking whether it is the goal, (*iii*) marking it as 'visited' to avoid its re-expansion, and (*iv*) obtaining its neighboring nodes, evaluating them, and (conditionally) pushing them to `OPEN`.

Action/Direction: The operation that must be done to obtain a node from its predecessor node. In this paper, we use the terms 'action' and 'direction' interchangeably.

Figure 1 shows a general example of an $A^\star$ graph search. Every node represents a state with n possible actions. For example, in 2D path planning with a robot that can move in four cardinal directions, every node represents an (x, y) state, and every action represents a direction the robot can take (e.g., a_1: up, a_2: right, a_3: down, a_4: left). In this paper,

we assume that the maximum number of actions in every node is the same (e.g., $n = 4$ in the example). This assumption holds in a variety of applications of $A^\star$, including but not limited to the ones that we evaluate in this paper (see **§Evaluation**).

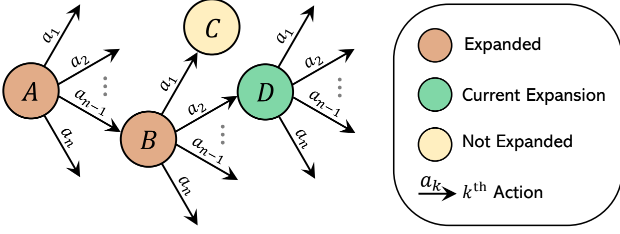


Figure 1: Graph search using $A^\star$.

Following, we define some notations that will help us explain *RA**. The examples refer to Figure 1's graph.

- $pa(S, i)$: the ith recursive *pa*rent of S.
  - ◇ Examples: $pa(B, 1) = A$, $pa(C, 1) = B$, $pa(C, 2) = A$, $pa(D, 1) = B$, $pa(D, 2) = A$.
  - ◇ The 0th recursive parent of a node is the node itself: $pa(S, 0) = S$.
  - ◇ $pa(S, i) = pa(pa(S, j), i - j)$ holds for any valid $i \geq j$.
- $a(S)$: the preceding *a*ction of S. It is the action that led to obtaining S and adding it to `OPEN`.
  - ◇ Examples: $a(B) = a_{n-1}$, $a(C) = a_1$, $a(D) = a_2$.
- $a(S, i)$: the preceding *a*ction of $pa(S, i - 1)$.
  - ◇ Examples: $a(B, 1) = a_{n-1}$, $a(C, 1) = a_1$, $a(C, 2) = a_{n-1}$, $a(D, 1) = a_2$, $a(D, 2) = a_{n-1}$.

Runahead $A^\star$

In this section, we explain *RA** with an *abstract* predictor. We defer the explanation of the evaluated predictors to the next section.

Prediction

The first operation *RA** performs upon an expansion is predicting *which successor of the expanded node is more likely to be expanded in the future*. We call it the Most Likely Successor (MLS). Note that, MLS is *not* necessarily the very next expansion in time order; MLS is the next expansion whose parent is the current expansion (details in Figure 2).

RA*'s predictor exploits correlations among previous preceding actions of an expanded node to predict the preceding action of its MLS (to reiterate, this is the action that leads to obtaining MLS and adding it to `OPEN`). Once the preceding action is predicted, the MLS can be trivially computed.

The prediction problem can be viewed as a classification problem where every action is a *class*, and the learning task is to learn the probability of an action Act being the preceding action of MLS in state S, given the history of past actions (Eq. 1). And, the prediction goal is finding the action that maximizes this probability (Eq. 2).

$$P(Act|a(S, 1), a(S, 2), a(S, 3), \dots) \tag{1}$$

$$\underset{i \in \{1, \dots, n\}}{\arg\max}\ P(a_i|a(S, 1), a(S, 2), a(S, 3), \dots) \tag{2}$$

For example, in Figure 1's graph, if a_k is the preceding action of D's MLS, we can write Eq. 1 as $P(a_k|a_2, a_{n-1})$.

Speculative Parallelism

While a node is being expanded, *RA** predicts its MLS, pre-evaluates the MLS's neighbors, and memoizes the results. These pre-evaluations are done by separate threads, in parallel with the expanded node's evaluations. Later, when the MLS is actually expanded, instead of waiting for its time-consuming neighbor evaluations, the pre-computed results (memoized) are used, saving a significant amount of time.

This parallelism among evaluations and pre-evaluations is *speculative*, meaning the prediction could be wrong, in which case the pre-evaluation results are not used. Note that speculation does *not* expand any nodes; hence, *RA** maintains the same expansion order as vanilla $A^\star$ and safeguards all optimality guarantees. Also, incorrect predictions lead to useless computations (power overhead) but do not affect the search outcome.

Running Further Ahead Speculation entails threading: preparing the threads' workload and issuing them. We found that threading overheads are *not* negligible. To deal with this issue, we predict multiple nodes at every iteration (i.e., MLS, MLS's MLS, and so on) and pre-evaluate their neighbors (details in Figure 2). This way, threading overheads are *amortized* over many pre-evaluations. We refer to the maximum number of pre-evaluations that we do at every iteration as the Runahead (R) of the method.

Memoization

The pre-evaluation results are stored in memory for later usage. We use a hashmap to memoize pre-evaluations. We find that it imposes negligible memory overhead ($1\% - 10\%$), particularly because $A^\star$ already uses large data structures like `OPEN`, that overshadow the hashmap memory. Overall, in our experiments (see **§Evaluation Methodology**), we do not find memory a limiting factor of *RA**.

Putting It All Together

Figure 2 shows *RA**'s pseudo-code. Lines `01`–`10` show the speculation procedure. *RA** predicts the preceding action of MLS (line `04`) and uses it to generate MLS itself (line `05`). Unless MLS is infeasible (e.g., an out-of-map location; line `06`), *RA** pre-evaluates its neighbors (lines `07`–`09`). This process repeats until the runahead expires (line `03`).

Lines `11`–`21` show every iteration of *RA**. First, the node with minimum f is expanded (line `11`). Then, preliminary operations of $A^\star$ are performed (mark node visited, etc.). Lines `13`–`16` evaluate the expanded node's neighbors.

Line `17` checks whether there are active threads (i.e., issued but not yet finished). If there are none, it means that no evaluation is going on. When so, *RA** does *not* speculate. Otherwise, if there are active threads, it means that some evaluations are being done; in parallel, *RA** triggers speculation (line `18`). Then, the algorithm (conditionally) pushes valid neighbors to `OPEN` (lines `19`–`21`) like vanilla $A^\star$.

```
// expNode: expanded node, R: runahead
// MLS: most likely successor
// precedAct: preceding action

01 def speculate(expNode, R):
02  MLS = expNode

03  while R > 0:
04   Act = predict(MLS.precedAct)
05   MLS = MLS.neighbors()[Act]
06   if MLS == INFEASIBLE: return

07   for n in MLS.neighbors():
08    if n.getStatus() == UNKNOWN:
09     thread { evaluate(n) }
10     R--

11 expNode = OPEN.top()
12 // preliminary operations

13 for n in expNode.neighbors():
14  if n.getStatus() == UNKNOWN:
15   n.setStatus(PENDING)
16   thread { evaluate(n) }

17 if activeThreads > 0:
18  speculate(expNode, R)

19 for n in expNode.neighbors():
20  if n.getStatus() == VALID:
21   // conditionally OPEN.push(n)
```

Figure 2: The pseudo-code of *RA*⋆.

The Prediction Mechanism

As explained, upon expansions, *RA*⋆ predicts MLS by predicting its preceding action, based on the history of past preceding actions (i.e., Eq. 2). Ideally, the correlations among preceding actions can be learned using a recurrent neural network (RNN), where past actions $(a(S, 1), a(S, 2), a(S, 3), \dots)$ are input features, and the future action (Act) is the model's output label. While such a scheme could offer a high level of prediction accuracy, we find that, in practice, its high *prediction latency* (i.e., the time it takes to generate a prediction) presents challenges to *RA*⋆, downgrading its performance.

In this paper, we explore simple, practical prediction approaches to achieve low prediction latency while offering a decent level of accuracy. We leave the exploration of a practical implementation of *RA*⋆ with RNNs, through, say, pruning (Gupta and Agrawal 2022), to future work.

Straight-Line Predictor

Our proposal for making practical predictions is what we call Straight-Line Predictor (SLP). Upon expanding S, SLP predicts that MLS's preceding action will be $a(S)$. In other words, SLP simplifies Eq. 2 to Eq. 3,

$$\underset{i \in \{1,\dots,n\}}{\arg\max}\; P(a_i | a(S)) \tag{3}$$

and considers that argmax i is such that $a_i = a(S)$. For the different applications that we consider, we found that $a(S)$ is a good approximation to the solution of the argmax problem.

Figure 3.a is a motivating example for SLP. It depicts 2D path planning for a mobile robot moving from state S to state G. The planner's purpose is to find the shortest path, and it uses the consistent heuristic function $h(S) = \|G - S\|_2$ (Euclidean distance).

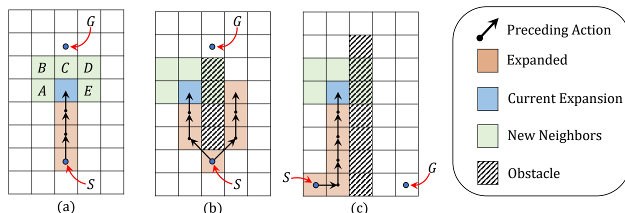


Figure 3: 2D path planning example.

Intuitively, to traverse the shortest path, the robot should head right towards the goal and *keep moving in the same direction* until either hitting an obstacle or reaching the goal. This intuition, in fact, emanates from the geometry principle that the shortest distance between any two points in an obstacle-free plane is a straight line.

Mathematically, after the current expansion, state C is expanded before $\{A, B, D, E\}$, given that C is not occupied. In other words, the next action after the current expansion will be the same as the preceding action—the premise of SLP.

SLP *mispredicts* when it hits an obstacle (generally, an invalid state). One might think SLP would work poorly in real-world environments where there may be many obstacles. However, corroborating (Bakhshalipour et al. 2022), we will show that obstacles in real environments are not so irregular that they destroy SLP's premise: SLP still offers decent prediction accuracy even in crowded environments.

For example, consider Figure 3.b and Figure 3.c, where the environments have obstacles. While moving from free space towards the goal, the direction changes upon hitting the first occupied space, which costs SLP a misprediction. However, *after that, preceding directions remain stable*, which helps SLP make correct predictions. This happens because occupied cells are not randomly scattered in the area; they are largely *co-located*, representing a (large) object (e.g., a wall).

Figure 3.b also shows how SLP deals with "temporally-interleaved" expansions, i.e., when consecutive expansions (in time order) belong to largely separate subgraphs. In Figure 3.b, $A^\star$ (alternately) expands nodes on the two sides of the long obstacle; consecutive expansions in time order belong to largely separate subgraphs. When so, upon an expansion, the very next expansion in time order is *not* a child of the current expansion; however, the next but one expansion is. The intuition of SLP is that the predicted child, if correct, will eventually be expanded; no matter when a node gets expanded, its pre-evaluation will speed up the execution at that point.

Beyond the synthetic environments of Figure 3, we show and argue that SLP's postulations (stable directions, co-located occupied cells) hold in real-world applications. Picture a self-driving car moving in a certain direction, on a street bounded by buildings from the sides. Even in the presence of line changes and overtaking, the vehicle will mostly move in a straight direction, as do nowadays manual cars.

Certainly, one can envision synthetic environments in which SLP would offer poor accuracy. For example, in an environment where there is only a narrow zig-zag, free path between obstacles, SLP would mispredict all expansions. In **§Harnessing SLP**, we propose and evaluate supplementary techniques to handle such corner cases. However, as such cases are rare in real-world environments, we conduct the main experiments of the paper using (not-enhanced) SLP.

We will show that SLP effectively predicts expansions in applications even beyond path planning. For example, in protein design, where the goal is minimizing an energy function, expansions follow straight-line patterns: if an action has resulted in (the largest) reduction in energy, repeating the

same action is likely to reduce energy again. We will quantify the prediction accuracy in different applications and discuss when and when not patterns are predictable.

Finally, in this paper, we focus on introducing the runahead concept, and implement *RA*⋆ using a simple-yet-effective predictor (i.e., SLP). In our future work, we will investigate various sophisticated predictors based on the patterns of expansions of each application and whether these patterns can be learned using data-driven approaches.

Random Action Predictor

For reference, we also evaluated a Random Action Predictor (RAP). RAP picks one of the possible n actions randomly. The purpose of including RAP is to compare it with SLP and shed light on the influence of prediction performance on the end-to-end speedup.

Sequitur

For reference, we also include Sequitur (SEQ) in our prediction evaluations. SEQ is originally proposed as a data compression algorithm (Nevill-Manning and Witten 1997); nevertheless, later work (Somogyi et al. 2009) uses it to measure the *prediction opportunity*. The purpose of including SEQ is to have an upper bound on prediction performance, and evaluate how close SLP is to an ideal predictor. Note that SEQ works by offline processing of data (i.e., preceding actions in this work) and is not practical.

Figure 4 shows SEQ's operation with an example input stream depicted using symbols ('w', 'x', 'y', and so on). We take this example from (Wenisch 2007). In the context of *RA*⋆, the input stream is the sequence of preceding actions. SEQ processes the entire stream and builds a grammar whose production rules are formed such that it captures repetitions in the stream, as shown in the figure.

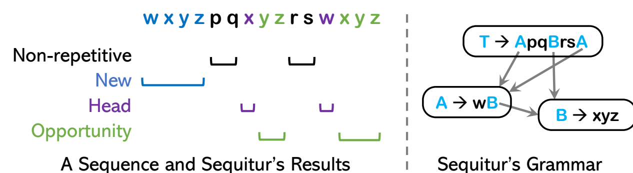


Figure 4: Sequitur.

Using the formed grammar, SEQ classifies the symbols of the input stream into four groups: (*i*) *Non-repetitive* refers to symbols that never recur in the stream, and hence are unpredictable using any repetition-based prediction method; (*ii*) *New* refers to the first occurrence of symbols in the stream (i.e., before the predictor can learn them); (*iii*) *Head* refers to the first symbol of every rule, which cannot be predicted itself, but it can be used to identify and predict the rest of the symbols in the rule; (*iv*) *Opportunity* refers to the rest of the symbols.

Sequitur argues that *Opportunity* is what an ideal predictor can predict by learning the repetitions in the data stream. Hence, processing Sequitur's grammar can provide an upper bound on prediction performance. In the provided example, for instance, the best-performing prediction mechanism can predict up to $\frac{5}{15}$ of the symbols. An implementation of Sequitur processing is available at: `https://github.com/bakhshalipour/SequiturAnalysis/`

Evaluation Methodology

Setup

We conduct our evaluations on up to 16 cores of `Intel Xeon Gold 5218R` processor. *RA*⋆'s runahead is set such that it utilizes all the available cores. Our operating system is `Debian 10`, and our compiler is `GCC 11`. We use `C++17 Thread Pool` library (Shoshany 2021) for threading operations. The default predictor of *RA*⋆ is SLP.

Applications

We evaluate five applications of *A*⋆, namely path planning in (x, y), (x, y, θ), (x, y, z) domains, protein design, and symbolic planning. Excluding symbolic planning, the expansions in these applications are slow.

Path Planning: For (x, y) and (x, y, θ) planning, we use `Boston` map from Moving AI (Sturtevant 2012). For (x, y, z) planning, we use `Freiburg` map from OctoMap (Hornung et al. 2013). The heuristic function of all the planners is Euclidean distance. The (x, y) and (x, y, z) planners can move in 8 and 6 directions, respectively. In (x, y, θ) planning, the discretization granularity of θ is $\frac{\pi}{4}$. The agents in (x, y) and (x, y, z) are rectangle-shaped, and the agent in (x, y, θ) is a polygon. For all path planning scenarios, we choose ten random start-goal points, and report the results for the average of ten executions. We model the other details of (x, y) and (x, y, z) planning respectively based on `pp2d` and `pp3d` of RTRBench (Bakhshalipour, Likhachev, and Gibbons 2022).

Protein Design: As the input set, we use `1I27_A_H` from (Zhou 2015). We use the heuristic function and the implementation of (Zhou and Zeng 2015) for estimating the energy and modeling the details.

Symbolic Task Planning: We evaluate a symbolic planner solving the Blocksworld problem with eight blocks with a random initial organization. The task planner uses symbol differences as the heuristic function. We model the other details based on `sym-blkw` of RTRBench (Bakhshalipour, Likhachev, and Gibbons 2022).

Competitors

We compare *RA*⋆ against *PA*⋆*SE* (Phillips, Likhachev, and Koenig 2014) and *HDA*⋆ (Kishimoto, Fukunaga, and Botea 2009). *PA*⋆*SE* is a state-of-the-art parallel *A*⋆ algorithm proposed for slow-expansion applications. *PA*⋆*SE*, unlike *RA*⋆, expands multiple nodes in parallel. More specifically, *PA*⋆*SE* discovers *independent states* and parallelizes their expansions. s and s' are two independent states if the expansion of s cannot lead to a shorter path to s', and vice-versa.

HDA⋆ assigns nodes to processors (cores) using a hash function. It maintains multiple `OPEN` lists, one per processor. When a processor expands a node, instead of pushing all

neighbors to its own OPEN, *HDA*⋆ hashes each neighbor to determine which processor should process it. That neighbor is then sent to the determined processor and gets processed by it. By doing so, it parallelizes multiple expansions; however, it loses *A*⋆'s guarantee that every node is expanded at most once. This implies that when the goal is expanded for the first time, *HDA*⋆ has not necessarily found the shortest path to it. Hence, *HDA*⋆ might expand nodes multiple times to ensure the goal is achieved by the least-cost path.

Spurred by the observation that re-expansions can happen too many times, we make one modification to the original *HDA*⋆. Namely, after performing a validity checking for a node (e.g., collision detection), we store the validity status (e.g., collision or free) in a global hashmap. When a node gets re-expanded, we query the hashmap to get the validity status. This way, costly operations like collision detection are performed only once when a node is generated for the first time; after that, the memoized results are used. This technique makes re-expansions cheaper, but not free (see **§Evaluation Results**).

Metrics

We consider the following metrics in our evaluations.

Speedup is the execution time of the vanilla *A*⋆ divided by the execution time of every method.

Prediction Accuracy is the percentage of *RA*⋆'s predictions whose pre-evaluation results are eventually used by the algorithm. I.e., correct predictions divided by total predictions. This metric is specific to *RA*⋆.

Prediction Coverage is the percentage of speculated evaluations that must otherwise (i.e., without *RA*⋆) be done non-speculatively. I.e., the total number of pre-evaluations with *RA*⋆ divided by the total number of evaluations with vanilla *A*⋆. This metric is specific to *RA*⋆.

Evaluation Results

Execution Time

Figure 5 shows the speedup of the parallelization methods. The baseline to which the speedups are normalized is a multithreaded implementation of the vanilla *A*⋆: only neighbors of the expanded nodes are evaluated in parallel.

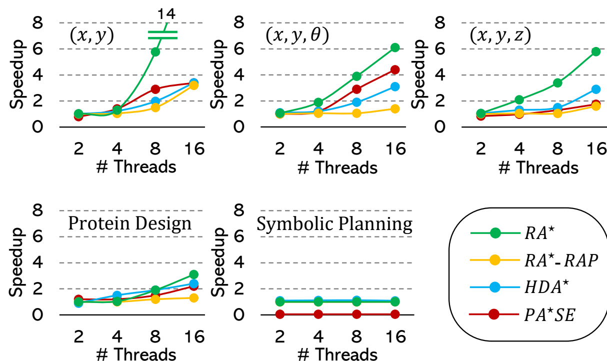


Figure 5: Speedup comparison of methods.

When the number of threads is small (e.g., < 4), the baseline *A*⋆ utilizes most of the threads, leaving few free threads for *RA*⋆. Consequently, *RA*⋆ does not have a considerable opportunity in such settings to improve performance. However, when the number of threads is large, *RA*⋆ offers significant speedups in slow-expansion applications. Across them, *RA*⋆ with 16 threads offers $3.1\times$–$14.1\times$ speedup over the baseline, outperforming *PA*⋆*SE* by $1.4\times$–$4.1\times$ and *HDA*⋆ by $1.3\times$–$4.1\times$. *RA*⋆'s high prediction performance, which we will present, is the main contributor to its large speedup. The importance of accurate prediction can be seen by comparing *RA*⋆ (with its SLP predictor) to *RA*⋆*-RAP* (where a random predictor is used instead of SLP).

RA⋆, nonetheless, fails to accelerate the symbolic task planner, a fast-expansion program. The reasons are (*i*) the evaluations are not so costly that the parallelism benefits could outweigh the threading overheads, (*ii*) *RA*⋆'s predictions are not accurate in this domain, and (*iii*) the baseline vanilla *A*⋆ almost always uses all the available threads to evaluate a large number of neighbors (the degree of graph vertices is significantly larger than the available threads), leaving few free threads for *RA*⋆.

We find that *PA*⋆*SE* suffers from two major issues. First, there are not always many independent states. This limits the amount of parallelism as the algorithm cannot find enough independent states to utilize all the available threads. Especially, when the number of threads is large (e.g., 16), the thread utilization can drop to even below 25%; this happens too frequently in applications in which the nodes in OPEN have close g values, and hence, they cannot pass *PA*⋆*SE*'s "independence checks." Meanwhile, as we show in **§Workload Distribution and Thread Utilization**, *RA*⋆ is often able to keep all the threads highly utilized.

The second problem with *PA*⋆*SE* is the high overhead of discovering independent states. *PA*⋆*SE* repeatedly checks *all* the states in OPEN and in its BE (which tracks *being expanded* nodes) to discover independent states. This imposes significant overheads proportional to the number of nodes in OPEN and BE at every check, which could be many. For example, in protein design, 53% of the entire execution time (averaged across all threads) is spent discovering independent states.

On the other hand, *HDA*⋆ suffers from other types of overheads. As stated in **§Parallelizing *A*⋆**, *HDA*⋆ allows the re-expansion of nodes to account for the fact that states may get expanded before they have the minimal cost from the start state. The re-expansions can happen exponentially-many times, imposing a massive extra workload. For example, in (x, y) planning, *HDA*⋆ increases the number of expansions by an average of $10.7\times$. Notice, memoizing the validity status that we add to *HDA*⋆ (see **§Competitors**) makes re-expansions significantly cheaper, but not free. Re-expansions still need to update *HDA*⋆'s data structures and send messages to the other cores; when re-expansions are too many, their cumulative overhead becomes significant.

The other performance-limiting components of *HDA*⋆ are (P1) long waits at synchronization barriers, and (P2) long path reconstruction time (Weinstock and Holladay 2017). P1 slows down execution, especially when the number of

threads is large. For example, with two cores, only one core would ever wait for the other. However, with 16 cores, 15 of them could be waiting for 1 core to complete processing, even if every one of those 15 cores has work added to its OPEN. This wait time downgrades performance. P2 is a serial process with many send-and-receive messages. Throughout this process, the goal vertex's owner core should send and receive messages for every vertex in the shortest path whose owner is different. With more cores, it is more likely that vertices on the shortest path belong to other owners, and hence, more send and receive messages must be communicated.

We also found synchronization overheads non-trivial in both *PA⋆SE* and *HDA⋆*. *PA⋆SE* locks all insertions to and deletions from its heavily-used data structures like OPEN and BE. *HDA⋆*, when running on a shared-memory system, needs to lock cost and parent lookup tables for every vertex (Weinstock and Holladay 2017). In contrast, *RA⋆* only needs to lock the hashmap of results[1] when it inserts evaluation and pre-evaluation results. As such, synchronization overheads are significantly lighter in *RA⋆*.

Prediction Performance

Figure 6 shows the prediction coverage and accuracy of different predictors. Recall that SEQ is impractical. Its accuracy is always 100%, and it makes a varying but *pattern-dependent* number of predictions per expansion; in contrast, SLP and RAP make R predictions, where *RA⋆*'s runahead parameter R depends on the number of free threads in the system. The purpose of including SEQ is to gauge how practical predictors compare to an ideal predictor.

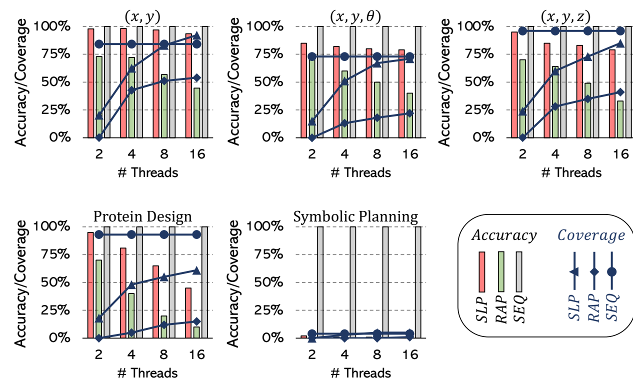


Figure 6: Performance comparison of different predictors. The default *RA⋆* predictor is SLP. SEQ is offline (impractical).

With increasing runahead (achieved by increasing the number of threads), *RA⋆* makes more predictions per expansion. By increasing the number of predictions, generally, the prediction accuracy drops and the coverage rises. The accuracy drops because it becomes less likely for a pattern to remain stable (e.g., a preceding action keeps repeating) for a larger number of iterations. On the other hand, the coverage increases as *RA⋆* runs *further ahead* and pre-evaluates more nodes.

In slow-expansion applications (i.e., all but symbolic task planning), with two threads (i.e., up to one prediction per expansion), SLP's accuracy is 85%–97.9%, which is 15%–25% higher than RAP and is within 2.1%–15% of SEQ. Interestingly, RAP's accuracy is far higher than $\frac{1}{n}$. We found that, in many cases, though RAP mispredicts the MLS, the *mispredicted MLS shares some neighbors with the actual MLS*; some of the pre-evaluations are eventually used. However, this is true when the runahead is small; with larger runaheads, RAP's predictions progressively diverge from the correct path. As a result, with large runaheads, *RA⋆* with RAP significantly underperforms *RA⋆* with the proposed SLP, as shown in Figure 5. With 16 threads, SLP's accuracy is 45%–93.5%, which is 35%–48.9% higher than RAP and is within 6.5%–55% of SEQ.

Also, with 16 threads, SLP offers 61.2%–92.2% coverage, which is 38.2%–49% higher than RAP and within an average 7.4% proximity of SEQ. In (x, y), SLP's coverage is even up to 8% better than SEQ; however, this higher coverage comes at the cost of 6.5% less accuracy caused by aggressively making many predictions at once when there is a high number of free threads.

In symbolic planning, the predictors offer very poor performance. SLP fails because its postulations do not hold in the application; RAP fails because the number of actions (i.e., node degree) is large. SEQ offers only slightly better prediction coverage.

Note that mispredictions, when they are numerous, can waste power; threads perform computations whose results are not used. Importantly, the waste happens only at the computation level (CPU) and not the entire system (memory, robot's mechanics, etc.). Hence, the overhead can be neglected in many settings like with mobile robots where the *entire* computation contributes to $< 5\%$ of the total power consumption (Boroujerdian et al. 2018), or in the cloud where components like network and memory consume the majority of power (Raoufi, Zhang, and Yang 2022).

Harnessing SLP

Our prediction performance results showed that SLP effectively predicts expansions in real-world path planning applications (and even beyond, i.e., protein design). However, one can indeed envision (synthetic) environments in which SLP is misled, exhibiting poor accuracy. In settings where the prediction accuracy is of significant importance, e.g., processors with few cores in which the system cannot afford to waste threads doing useless computation, this could lead to performance degradation. We propose a supplementary technique to "harness" SLP in such cases.

We speculate only if the path leading to expansion was *stable* in the last s steps; i.e., $a(S, 1) = a(S, 2) = \ldots = a(S, s)$. For example, with $s = 3$, speculation runs if the

[1]The lock is implementation-dependent. For example, with a bucketized hashmap, only the manipulated bucket must be locked, not the entire hashmap; or, with a state-space-sized array as the hashmap, the lock is not needed at all, because there is a 1:1 mapping from states to array elements.

expanded node's preceding action is the same as its parent's preceding action, and its parent's parent's preceding action.

To evaluate this mechanism, we generate synthetic 2D maps (like the evaluated `Boston`) for (x, y) planning. The maps are initially empty; with a probability of p, we add *random* obstacles to them. Figure 7 shows SLP's prediction performance with 16 threads and different p and s. The bars show prediction accuracy and the lines show prediction coverage.

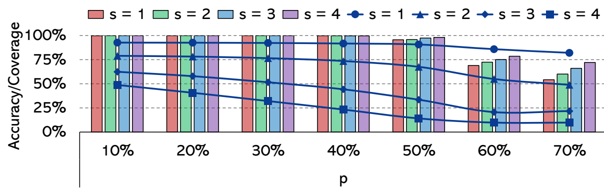


Figure 7: Harnessing SLP on random maps.

As the results show, the harnessing mechanism effectively regulates the predictor's aggressiveness: with $s = 4$, it harnesses the aggressiveness such that even in a 70%-occupied environment with random obstacles, the accuracy is still above 72.1%. On the flip side, the coverage drops as a result of reduced prediction opportunities.

Another point revealed by this experiment is the large difference in the prediction performance of SLP in realistic and random environments. For example, with $s = 1$, SLP offers 39.3% more accuracy in `Boston` compared to the 70%-random map, and also, it covers 10.1% more evaluations. Our results corroborate prior work (Bakhshalipour et al. 2022) that path planning in real-world environments exhibits predictable patterns. To illuminate this, Figure 8 compares SLP's performance in multiple cities' maps of Moving AI (Sturtevant 2012) with the 70%-random map. The bars show the accuracy and the dots show the coverage.

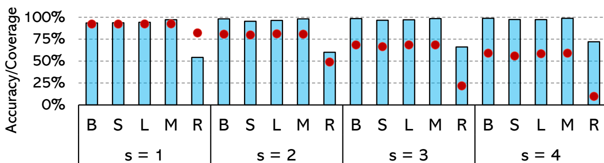


Figure 8: SLP in real-world (Boston, London, Moscow, Shanghai) and random (70%-Random) environments.

With $s = 1$, the worst-case accuracy/coverage of SLP in the real-world maps is 39.3%/10.1% higher than in the random environment. With $s = 4$, it is 25.3%/46% better.

Planning Resolution

To study the impact of *evaluation time* on *RA**'s speedup, we coarsen the resolution in (x, y) planning. With such coarsening, the *number* of obstacles decreases, because multiple nearby obstacles are considered one big obstacle. As a result, collision detection becomes faster; on the downside, the free space is underestimated. The bars in Figure 9 show *RA**'s speedup with different numbers of threads and different coarsening factors. A coarsen factor of M means that the planner considers M times fewer units (pixels) in every map dimension. For example, a $16\times$ coarsening means the considered map has 16×16 times fewer units than the original map. The blue line with solid dots in the figure shows the average evaluation (collision detection) time with the corresponding resolution.

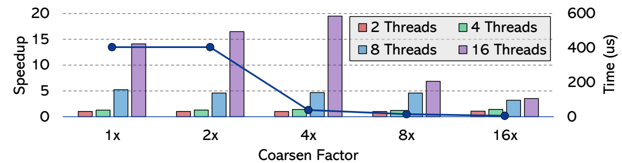


Figure 9: *RA**'s speedup with coarsening the resolution.

The results show that *RA** offers considerable speedups with different resolutions. It is expected that with decreasing evaluation time, the speedup provided by parallel methods becomes smaller (see **§Speedup Projection**), as a result of increasing the *fraction* of time the program spends in the serial part (e.g., manipulating `OPEN`). This expectation generally holds true for *RA** with a few anomalies that happen because of taking different execution paths; i.e., with different coarsening factors, the planning is essentially done on *different maps*, which results in taking different execution paths. When the collision detection is costly ($38\mu s - 404\mu s$), *RA** offers $14.1\times$–$19.5\times$ speedup; when the collision detection becomes faster ($5\mu s - 14\mu s$) its speedup drops to $3.5\times$–$6.9\times$.

Weighted $A^\star$

Weighted $A^\star$ (*WA**) (Pohl 1970) inflates the nodes' heuristic cost by a factor of $\varepsilon > 1$. Doing so, nodes are expanded in the order of $f = g + \varepsilon \times h$. This way, the search is biased towards the nodes that are closer to the goal. With *WA**, the search terminates earlier (i.e., the goal itself is expanded sooner), but the solution could become ε times costlier.

Figure 10 shows the speedup of the methods for *WA** with $\varepsilon = \{1, 2, 4\}$ for the average of slow-expansion applications. As the results show, *RA** consistently outperforms *PA***SE* and *HDA**. However, with increasing ε, the speedup of *RA** drops, because of the reduced opportunity; the larger ε, the fewer expansions.

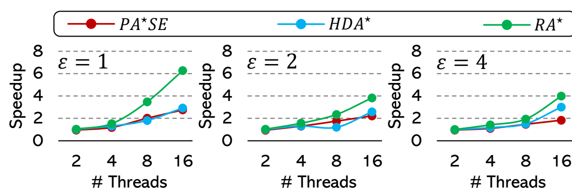


Figure 10: Speedup comparison of methods for *WA**, averaged over the slow-expansion applications.

RA* on Top of Prior Work?

Theoretically, *RA** can be orthogonally used with *PA***SE* and *HDA**, as well as many other prior approaches. *RA** is orthogonal to parallel best-first search algorithms because those algorithms try to expand multiple nodes in parallel; in contrast, *RA** runs ahead of individual expansions and paves the way for accelerating future expansions; they exploit different sources of parallelism.

Nevertheless, we observed that combining *RA** with prior approaches does not improve performance significantly. Not unexpectedly, combining these approaches results in stacking their overheads; unfortunately, the increased parallelism does not outweigh the increased overheads. Particularly, since *HDA** relies on many node-to-node communications, its overheads are entirely stacked with the overheads of *RA** and results in slowdown rather than speedup. We leave to future work the exploration of an efficient combination of *RA** with other parallel algorithms.

Workload Distribution and Thread Utilization

Bars in Figure 11 show the average number of useful (pre-)evaluations (i.e., evaluations and pre-evaluations whose results are ultimately used) per node expansion. The bars are broken down into demand and speculative; demand represents the evaluations done by the baseline algorithm (line `16` in Figure 2), and speculation represents the pre-evaluations of *RA** (line `09` in Figure 2). With increasing the number of threads, the fraction of speculative evaluations increases. This way, more evaluations are lifted from the critical path, reducing the execution time.

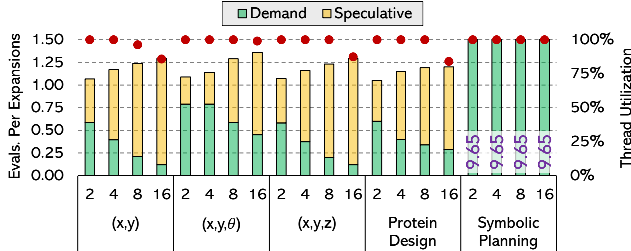


Figure 11: Workload distribution and thread utilization with different numbers of threads.

The solid dots in Figure 11 show the thread utilization. We define thread utilization as the fraction of threads performing evaluations over all threads in non-idle expansions (i.e., expansions with at least one evaluation). When the total number of threads is fewer than 8, the utilization ratio is almost 100%. This shows that with *RA**, there is enough workload to keep all threads busy. With increasing threads, the utilization ratio starts decreasing because of the finite nature of the parallelism even in the presence of speculation. With 16 threads, the utilization ratio is 84%–99% in slow-expansion applications.

Speedup Projection

We develop an analytical model for *RA** to project its speedup beyond the studied applications. We recall that *RA** does not expand any nodes during the speculation phase. Hence, let us assume both the baseline *A** and *RA** expand N nodes in total. Let us assume every node has n neighbors, and the system supports the parallel execution of $t > n$ threads. Consider t_{serial} as the execution time of the serial portion of the algorithm at every expansion (i.e., everything other than neighbor evaluation), $t_{threading}$ as the threading overhead (i.e., the time it takes to manage all threads), and t_{eval} as the evaluation time. The total execution time of the baseline can be formulated as:

$$T_{Baseline} = N \times (t_{serial} + t_{threading} + t_{eval}) \quad (4)$$

Note that we assume all neighbors are evaluated in parallel, as there are more threads available than the number of neighbors (i.e., $t > n$). Now, we formulate the execution time of *RA**. Let us assume P_{cov} is the fraction of correctly predicted expansions out of all expansions, i.e., prediction coverage; and, P_{acc} is the prediction accuracy.

Also, consider l as the prediction latency, and R as the runahead. In non-idle expansions (i.e., expansions with at least one evaluation), *RA** pre-evaluates R nodes; it predicts $\frac{R}{n}$ nodes (line `04` in Figure 2) and pre-evaluates their (up to) n neighbors (lines `06`–`09` in Figure 2). We call $R' = \frac{R}{n}$ *predictor runahead*.

At every expansion, with a probability of P_{cov}, no evaluation is performed (the memoized results are loaded). Consider t_{load} as the time it takes to load the memoized results. We can formulate the execution time of *RA** as:

$$T_{RA^\star} = N \times (t_{serial} + P_{cov} \times t_{load} + (1 - P_{cov}) \times (t_{threading} + t_{eval} + R' \times l)) \quad (5)$$

As mentioned, l is the latency of generating one prediction, and hence, $R' \times l$ is the latency of generating all predictions.

Considering that in non-idle expansions, *RA** evaluates n demand nodes and $R = R' \times n$ speculative nodes, we can approximately[2] formulate P_{cov} as:

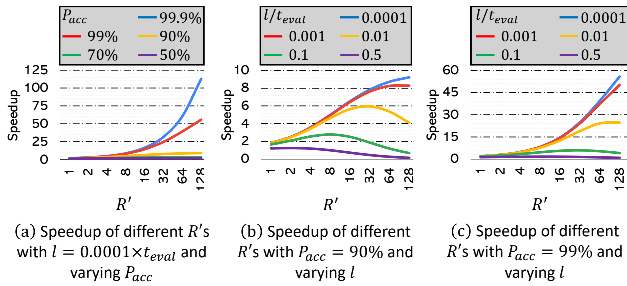
$$P_{cov} \approx \frac{R}{R+n} \times P_{acc} = \frac{R' \times n}{(R'+1)n} \times P_{acc} = \frac{R'}{R'+1} \times P_{acc} \quad (6)$$

In slow-expansion applications, we can assume t_{serial}, $t_{threading}$, $t_{load} \ll t_{eval}$. By eliminating these terms and plugging Eq. 6 into Eq. 5, we can formulate speedup as:

$$Speedup = \frac{T_{Baseline}}{T_{RA^\star}} = \frac{t_{eval}}{(1 - \frac{R'}{R'+1} \times P_{acc}) \times (t_{eval} + R' \times l)} \quad (7)$$

In this formula, P_{acc} depends on the effectiveness of the employed predictor for the application. R' is dependent on how many threads can be executed in parallel on the system. And, l depends on the predictor design. Figure 12 shows how speedup varies with different parameters.

[2] The equation does not take the early-termination cases (i.e., when MLS is infeasible; see Line `06` in Figure 2) into account.



(a) Speedup of different R's with $l = 0.0001 \times t_{eval}$ and varying P_{acc}

(b) Speedup of different R's with $P_{acc} = 90\%$ and varying l

(c) Speedup of different R's with $P_{acc} = 99\%$ and varying l

Figure 12: Projected speedup.

The formula suggests that improving $RA^{\star}$'s speedup for an application can be achieved in the following ways:

- Improving prediction accuracy: the higher P_{acc}, the higher speedup. Particularly, when the prediction accuracy is high, even the slightest incremental improvement could be of significant importance. For example, in Figure 12.a, with $R' = 128$, improving P_{acc} from 90% to 99% results in a $46.6\times$ speedup. In the same configuration, improving the prediction accuracy from 99% to 99.9% brings another $57\times$ speedup. This shows how highly-accurate predictors could greatly accelerate the execution (up to $112.9\times$ in Figure 12.a).

- Increasing runahead: by running further ahead, which is viable by using more threads, $RA^{\star}$ can further parallelize the algorithm, and further enhance the speedup. However, this projection comes with a caveat: increasing runahead results in increasing the cumulative prediction latency (i.e., $R' \times l$). As such, increasing runahead is good so long as the cumulative prediction latency is outweighed. Therefore, when the prediction latency (i.e., l) is small, increasing runahead yields higher speedups. For example, with $l = 0.0001 \times t_{eval}$, increasing R' from 0 to 128 results in $9.2\times$ speedup when $P_{acc} = 90\%$ (Figure 12.b) and $55.9\times$ when $P_{acc} = 99\%$ (Figure 12.c).

- Reducing prediction latency: the faster $RA^{\star}$ generates the predictions, the better. This is particularly true when R' and P_{acc} are high. For example, in Figure 12.c, with $R' = 128$, reducing the prediction latency from $0.001 \times t_{eval}$ to $0.0001 \times t_{eval}$ results in a $5.7\times$ speedup. Also, when the prediction latency is too long ($> 0.1 \times t_{eval}$), $RA^{\star}$ not only does not achieve speedup, it causes slowdown, even when the prediction accuracy is high. This is the behavior we observed with RNNs: although they offer high accuracy, their long prediction latency downgrades their effectiveness. Reducing prediction latency can be achieved by simplifying the prediction method (e.g., *pruning* in the context of deep learning). Likely, it is in conflict with the first approach, i.e., improving prediction accuracy. Typically, there is a trade-off between the performance of a predictor and its overheads; the sophisticated predictors offer a high level of accuracy but impose significant overheads (time) for generating a prediction.

Conclusion

$A^{\star}$ is the backbone of many artificial intelligence applications, and its acceleration can lead to substantial performance improvements in wide-ranging applications. We propose *Runahead* $A^{\star}$, a method that opportunistically parallelizes the search for slow-expansion applications. *Runahead* $A^{\star}$ enhances parallelism by predicting future expansions and pre-computing their expensive operations, thereby reducing execution time. We showed that $RA^{\star}$ is able to reduce the execution time by up to $14.1\times$ using 16 threads, while preserving $A^{\star}$ optimality guarantees.

Acknowledgments

This work was supported in part by National Science Foundation grant CCF-2028949, by a VMware University Research Fund Award, and by the Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma). Mohammad Bakhshalipour was supported by the Apple CMU ECE Ph.D. Fellowship in Integrated Systems. We would like to thank the anonymous reviewers for their valuable comments.

References

Bakhshalipour, M.; Ehsani, B.; Qadri, M.; Guri, D.; Likhachev, M.; and Gibbons, P. B. 2022. RACOD: Algorithm/Hardware Co-Design for Mobile Robot Path Planning. In *International Symposium in Computer Architecture (ISCA)*, 597–609.

Bakhshalipour, M.; Likhachev, M.; and Gibbons, P. B. 2022. RTRBench: A Benchmark Suite for Real-Time Robotics. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 175–186.

Boroujerdian, B.; Genc, H.; Krishnan, S.; Cui, W.; Faust, A.; and Reddi, V. 2018. MAVBench: Micro Aerial Vehicle Benchmarking. In *International Symposium on Microarchitecture (MICRO)*, 894–907.

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.

Darabi, S.; Sadrosadati, M.; Akbarzadeh, N.; Lindegger, J.; Hosseini, M.; Park, J.; Gómez-Luna, J.; Mutlu, O.; and Sarbazi-Azad, H. 2022. Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources. In *International Symposium on Microarchitecture (MICRO)*, 228–244.

Gupta, M.; and Agrawal, P. 2022. Compression of Deep Learning Models for Text: A Survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 16(4): 1–55.

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.

Hornung, A.; Wurm, K. M.; Bennewitz, M.; Stachniss, C.; and Burgard, W. 2013. OctoMap: An Efficient Probabilistic

3D Mapping Framework Based on Octrees. *Auton. Robots*, 34(3): 189–206.

Irani, K.; and Shih, Y.-F. 1986. Parallel A* and AO* Algorithms- An Optimality Criterion and Performance Evaluation. In *International Conference on Parallel Processing (ICPP)*, 274–277.

Jinnai, Y.; and Fukunaga, A. 2016. Abstract Zobrist Hashing: An Efficient Work Distribution Method for Parallel Best-First Search. In *AAAI Conference on Artificial Intelligence*, 717–723.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 201–208.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a Simple, Scalable, Parallel Best-First Search Strategy. *Artificial Intelligence*, 195: 222–248.

Leach, A. R.; and Lemon, A. P. 1998. Exploring the Conformational Space of Protein Side Chains Using Dead-End Elimination and the A* Algorithm. *Proteins: Structure, Function, and Bioinformatics*, 33(2): 227–239.

Lee, J.; Jung, C.; Lim, D.; and Solihin, Y. 2009. Prefetching with Helper Threads for Loosely Coupled Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 20(9): 1309–1324.

Nevill-Manning, C. G.; and Witten, I. H. 1997. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research*, 7: 67–82.

Phillips, M.; Likhachev, M.; and Koenig, S. 2014. PA*SE: Parallel A* for Slow Expansions. In *International Conference on Automated Planning and Scheduling (ICAPS)*, volume 24, 208–216.

Pohl, I. 1970. Heuristic Search Viewed As Path Finding in a Graph. *Artificial Intelligence*, 1(3-4): 193–204.

Raoufi, M.; Zhang, Y.; and Yang, J. 2022. IR-ORAM: Path Access Type Based Memory Intensity Reduction for Path-ORAM. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 360–372.

Shoshany, B. 2021. A C++17 Thread Pool for High-Performance Scientific Computing. *arXiv e-prints*, arXiv:2105.00613.

Somogyi, S.; Wenisch, T. F.; Ailamaki, A.; and Falsafi, B. 2009. Spatio-Temporal Memory Streaming. In *International Symposium in Computer Architecture (ISCA)*, 69–80.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, 4(2): 144–148.

Weinstock, A.; and Holladay, R. 2017. Parallel A* Graph Search. Project Report, School of Computer Science, Carnegie Mellon University, available in https://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf. Accessed: 2023-03-29.

Wenisch, T. F. 2007. *Temporal Memory Streaming*. Ph.D. thesis, Carnegie Mellon University.

Zhou, R.; and Hansen, E. A. 2007. Parallel Structured Duplicate Detection. In *AAAI Conference On Artificial Intelligence*, 1217–1224.

Zhou, Y. 2015. Open Source Protein REdesign for You on a GPU. https://github.com/zhou13/gOSPREY. Accessed: 2023-03-29.

Zhou, Y.; and Zeng, J. 2015. Massively Parallel A* Search on a GPU. In *AAAI Conference On Artificial Intelligence*, volume 29, 1248–1255.