APPEARED IN
Int. J. Robotics Research, 2021

HyP-DESPOT: A Hybrid Parallel Algorithm for Online Planning under Uncertainty

Panpan Cai*, Yuanfu Luo, David Hsu and Wee Sun Lee

School of Computing, National University of Singapore, 117417 Singapore

Abstract

Robust planning under uncertainty is critical for robots in uncertain, dynamic environments, but incurs high computational cost. State-of-the-art online search algorithms, such as DESPOT, have vastly improved the computational efficiency of planning under uncertainty and made it a valuable tool for robotics in practice. This work takes one step further by leveraging both CPU and GPU parallelization in order to achieve real-time online planning performance for complex tasks with large state, action, and observation spaces. Specifically, Hybrid Parallel DESPOT (HyP-DESPOT) is a massively parallel online planning algorithm that integrates CPU and GPU parallelism in a multi-level scheme. It performs parallel DESPOT tree search by simultaneously traversing multiple independent paths using multi-core CPUs; it performs parallel Monte-Carlo simulations at the leaf nodes of the search tree using GPUs. HyP-DESPOT provably converges in finite time under moderate conditions and guarantees near-optimality of the solution. Experimental results show that HyP-DESPOT speeds up online planning by up to several hundred times in several challenging robotic tasks in simulation, compared with the original DESPOT algorithm. It also exhibits real-time performance on a robot vehicle navigating among many pedestrians.

Keywords: planning under uncertainty, real-time motion planning, parallel planning.

1 Introduction

As robots move towards uncontrolled natural human environments in our daily life—at home, at work, or on the road—they face a plethora of uncertainties: imperfect robot control, noisy sensors, and fast-changing environments. A key difficulty here is *partial observability*: the system states are not directly revealed. A principled way of handling partial observability is to capture the uncertainties in a *belief*, which is a probability distribution over states, and reason about the effects of robot actions, sensor observations, and change of the environment on the belief. A planning algorithm looks ahead by searching a *belief tree*, in which each tree node represents a belief, and parent and child nodes are connected by action-observation pairs to capture the change in the belief as a result of robot actions and sensor observations (Fig. 1). While the belief tree search is conceptually simple, it is computationally intractable in the worst case, as the number of states or the planning time horizon increases.

DESPOT (Ye et al., 2017) is a state-of-the-art belief tree search algorithm for online planning under uncertainty. To overcome the computational challenge, DESPOT samples a set of "scenarios" and constructs incrementally—via heuristic tree search and Monte Carlo simulation—a *sparse* belief tree, which contains only branches reachable under the sampled scenarios (Fig. 1). The sparse tree is provably near-optimal (Ye et al., 2017), and DESPOT has shown strong performance in various robotic tasks, including autonomous driving (Bai et al., 2015) and object manipulation (Koval et al., 2016; Li et al., 2016).

We seek to scale up DESPOT further using parallelization and enable robots to perform real-time planning under uncertainty for complex tasks with large state, action, and observation spaces, e.g., autonomous driving in a dense crowd of pedestrians. Specifically, we propose *Hybrid Parallel DESPOT* (HyP-DESPOT), which exploits both multi-core CPUs and GPUs to form a multi-level parallelization scheme for DESPOT.

HyP-DESPOT uses multiple CPU threads to perform parallel tree search by simultaneously traversing many paths. The CPU threads provide the flexibility to handle the irregularity of tree search for parallelization. The key issue here is to distribute the threads over a diverse set of tree paths, while preserving the optimality of the search.

HyP-DESPOT uses GPUs to perform massively par-

*Corresponding author. Email: dcscaip@nus.edu.sg

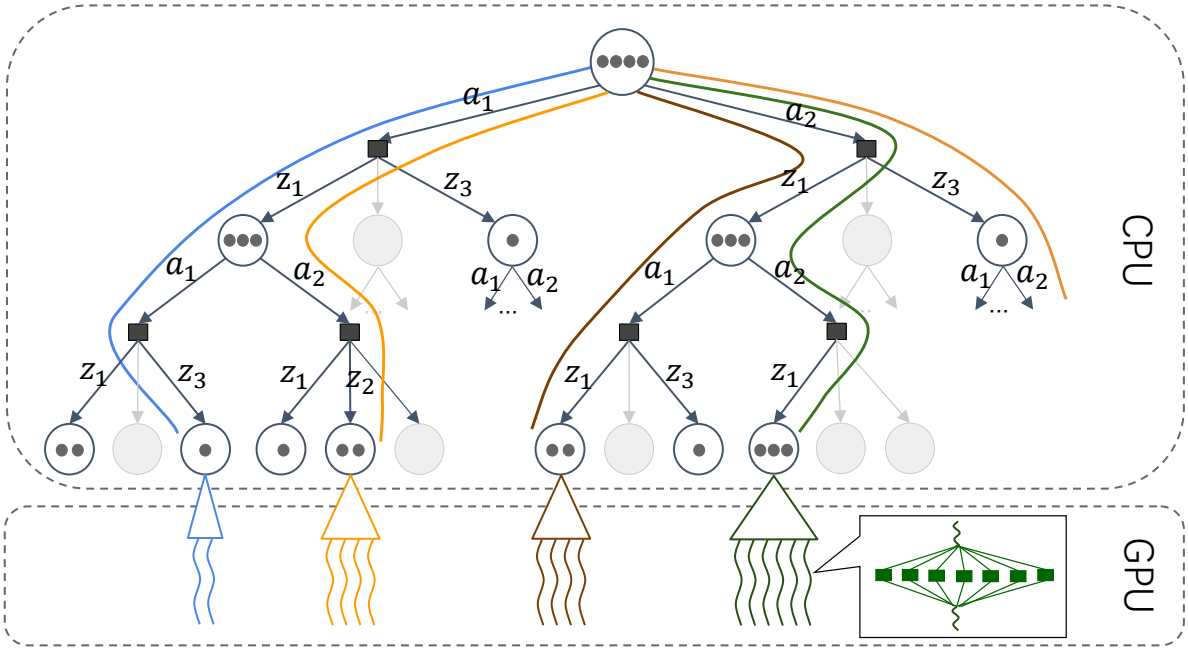


Figure 1: An overview of HyP-DESPOT. Each node of the belief tree (gray) represents a belief. A parent node and a child node, with associated beliefs b and b' respectively, are connected by an action-observation pair (a, z), indicating that the belief transitions from b to b', when a robot, with initial belief b, takes actions a and receives observation z. The DESPOT tree (black) is a sparse subtree of the belief tree and contains only branches reachable under a set of sampled scenarios (black dots). HyP-DESPOT integrates CPU and GPU parallelism: multi-threaded parallel tree search (colored paths) in the CPUs, massively parallel Monte Carlo simulation at the leaf nodes in the GPUs, and fine-grained GPU parallelization within a simulation step by factoring the system dynamics model (inset figure).

allel Monte Carlo simulations at the belief tree node level, the action level, and the scenario level. Further, a complex system often consists of multiple components, e.g., multiple robots or humans in an interactive or collaborative setting. HyP-DESPOT factors the dynamics model and the observation model of such a system in order to extract additional opportunities for GPU parallelization at a fine-grained level. Since the simulations are independent, parallelization is conceptually straightforward. However, GPUs suffer from high memory access latency and low single-thread arithmetic performance. Parallel simulation and parallel tree search must be integrated to generate sufficient parallel workload and benefit from large-scale GPU parallelization. HyP-DESPOT deploys GPU simulations when the parallel workload is high and switches back to CPU simulations on the fly when the parallel workload is low.

To our knowledge, HyP-DESPOT[1] is the first massively parallel algorithm for online robot planning under uncertainty. It provably converges in finite time under moderate conditions and guarantees optimality or near-optimality of the solution. Our experiments show that HyP-DESPOT achieves significant speedup and better solutions, compared with the original DESPOT algorithm, in various online planning tasks.

In the following, Section 2 provides the background on planning under uncertainty and parallel planning. Section 3 presents an overview of HyP-DESPOT. Section 4 and 5 provide technical details on parallel tree search and on parallel Monte Carlo simulation, respectively. Section 6 presents experiments that evaluate the key components and parameters of HyP-DESPOT. It also reports the performance of HyP-DESPOT for real-time control of an autonomous vehicles driving among many pedestrians. Finally, we summarize the main results and point out directions of future work Section 7.

2 Background

2.1 Online Planning under Uncertainty

Consider a robot operating in a partially observable stochastic environment. It has a state space S, an action space A, and an observation space Z. We model the robot's stochastic dynamics with a transition probability function $T(s, a, s') = p(s'|s, a)$ for $s, s' \in S$ and $a \in A$, and model noisy sensing with an observation probability function $O(s', a, z) = p(z|a, s')$, for $s' \in S$,

[1]Code available at https://github.com/AdaCompNUS/HyP-DESPOT

$a \in A$, and $z \in Z$.

There are two general approaches to plan under uncertainty: offline value iteration (e.g., (Pineau et al., 2003; Smith and Simmons, 2004; Kurniawati et al., 2008; Lim et al., 2011)) and online belief tree search (e.g., (Ross and Chaib-Draa, 2007; Silver and Veness, 2010; Ye et al., 2017; He et al., 2011)). Offline planning reason about all future contingencies in advance to achieve faster execution time online. In contrast, online planning focuses on the contingency currently encountered and scales-up to much more complex tasks.

For online planning, a robot computes an action at each time step and interleaves planning and action execution. To determine the best action at the current belief b, we perform lookahead search in a belief tree rooted at b (Fig. 1). The search optimizes the "value" over all policies:

$$\pi^*(b) = \arg\max_{\pi} V_\pi(b). \tag{1}$$

A *policy* π specifies the robot action at every belief, and the *value* of π at a belief b, $V_\pi(b)$, is the expected total discounted reward of executing the policy, with initial belief b:

$$V_\pi(b) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(b_t)) \middle| b_0 = b\right), \tag{2}$$

where $R(s, a)$ is a real-valued reward function designed to capture desirable robot behaviors and γ is a discount factor expressing the preference for immediate rewards over future ones. The robot then executes the action $a = \pi^*(b)$ and receives an observation z. We update the belief by incorporating the information in a and z according to the Bayes' rule:

$$b'(s') = \tau(b, a, z) = \eta O(s', a, z) \sum_{s \in S} T(s, a, s') b(s), \tag{3}$$

where η is a normalization constant. The new belief b' then becomes the entry point of the planning cycle for the next time step.

While belief tree search incurs a high computational cost, Monte Carlo sampling is a powerful idea to make it efficient in practice. Early examples include the roll-out algorithm (Bertsekas and Castanon, 1998), sparse sampling (Kearns et al., 2002), hindsight optimization (Chong et al., 2000), and AEMS (Ross and Chaib-Draa, 2007). Two state-of-the-art belief tree search algorithms, POMCP (Silver and Veness, 2010) and DESPOT (Ye et al., 2017), both make use of Monte Carlo sampling. POMCP performs Monte Carlo tree search (MCTS) on the belief tree and uses the partially observable UCT algorithm (PO-UCT) to trade-off exploration and exploitation. DESPOT performs anytime heuristic search in a sparse belief tree conditioned on a set of sampled scenarios. Both POMCP and DESPOT solve moderately large planning tasks efficiently, while DESPOT provides much better theoretical performance guarantee in the worst case. The UCT search of POMCP is sometimes overly greedy and suffers the worst-case complexity of $\Omega(\exp(\exp(\ldots \exp(1) \ldots)))$[2] to find a sufficiently good action (Coquelin and Munos, 2007). More importantly, DESPOT offers better opportunities for parallelization, as it generates a large number of Monte Carlo simulations, each corresponding to a sampled scenario, that can be processed simultaneously, rather than sequentially as POMCP does.

An alternative for planning under uncertainty is to combine traditional deterministic motion planning and stochastic optimal control. Earlier work handles stochastic dynamics and noisy sensing by integrating local control using linear-quadratic Gaussian (LQG) with global planning using probabilistic roadmaps (PRM) (Prentice and Roy, 2009) or rapid exploring random trees (RRT) (Van Den Berg et al., 2011). These methods, however, are based on Gaussian noise assumptions. In contrast, HyP-DESPOT does not assume the form of uncertainty.

2.2 Parallel Planning

Modern computer hardware, such as multi-core CPUs and GPUs, significantly boost the performance of planning algorithms and scale them up to complex robot tasks.

Parallelization has been exploited extensively in classic motion planning on expensive subroutines such as free space construction (Lozano-Prez and O'Donnell, 1991) and online collision checking (Bialkowski et al., 2011), on the planning algorithm itself (Challou et al., 1993; Plaku et al., 2005; Jacobs et al., 2013), and on both planning and collision checking (Cai et al., 2018a). These algorithms, however, do not deal with uncertainty, which is an crucial challenge in real-world planning.

Planning under uncertainty is usually formulated as a *Markov decision process* (MDP) if the system state is fully observable, or as a *partially observable Markov decision process* (POMDP) if the system state is not (Russell and Norvig, 2002). Parallelization has been exploited to speed up both MDP planning (Chaslot et al., 2008; Rocki and Suda, 2011; Barriga et al., 2014; Johnson et al., 2016) and offline POMDP planning (Lee and

[2]Composition of $D - 1$ exponential functions for search depth D.

Kim, 2016; Wray and Zilberstein, 2015).

Parallel MDP planning has focused on parallel *Monte Carlo tree search* (MCTS). Three main schemes exist: leaf parallelization (Cazenave and Jouandeau, 2007), root parallelization (Cazenave and Jouandeau, 2007), and tree parallelization (Chaslot et al., 2008). Leaf parallelization performs multiple roll-outs at a leaf node simultaneously. Root parallelization builds multiple trees in parallel. Both schemes only use CPU threads but can be combined to exploit massive GPU parallelization. Block parallelization (Rocki and Suda, 2011) uses GPUs to process roll-out requests from multiple trees. Multi-block parallelization (Barriga et al., 2014) further batch roll-outs for the children of leaf nodes. Unfortunately, simply increasing the number of trees or the number of roll-outs only offer limited benefits, because the trees do not share information, which could result in repeated efforts. Tree parallelization addresses this issue by cooperatively searching a shared tree. The challenge is to minimize the communication overheads for CPU threads. HyP-DESPOT integrates existing schemes in a novel form to achieve massively-parallel belief tree search. It performs both tree parallelization and leaf parallelization in a CPU-GPU hybrid scheme.

Offline POMDP planning computes beforehand a policy for all contingencies, thus inducing a huge number of independent tasks for parallelization. gPOMDP (Lee and Kim, 2016) parallelizes Monte Carlo value iteration (MCVI) (Lim et al., 2011) by batching Monte Carlo simulations for multiple beliefs, action candidates, and policy graph nodes in GPUs. A similar idea (Wray and Zilberstein, 2015) is used to parallelize point-based value iteration (PBVI) (Pineau et al., 2003).

Offline planning has almost unlimited computation time to derive a solution. In contrast, online planning is usually given a small fixed amount of time to choose the best action in real-time. Parallelism is thus much more important for online planning to scale-up to complex tasks, but is rarely explored. Our work aims to fill this gap.

This paper extends our earlier work on parallel belief tree search (Cai et al., 2018b) with new algorithmic components for optimistic trials (Section 4.4) and hybrid expansions (Section 5.4). We also provide new theoretical analysis on the convergence and optimality of the algorithm (Theorem 1). Additional experiments are conducted for the extended algorithm, including a real robot experiment (Section 6.5).

3 Overview

HyP-DESPOT is a belief tree search algorithm that leverages a CPU-GPU hybrid parallel model for online planning under uncertainty. It parallelizes the DESPOT algorithm (Ye et al., 2017) and retains its theoretical performance guarantee. For completeness, we provide a brief summary of the DESPOT algorithm (Section 3.1), followed by the HyP-DESPOT algorithm (Section 3.2).

3.1 DESPOT

To overcome the computational challenge of online planning under uncertainty, DESPOT samples a small finite set of K scenarios as representatives of the future. Each scenario, $\boldsymbol{\phi} = (s_0, \varphi_1, \varphi_2, ...)$, contains a state s_0 sampled from the initial belief and random numbers $\varphi_1, \varphi_2, ...$ that determinize the uncertain outcomes of future actions and observations. DESPOT then applies a *deterministic simulative model* $g\colon S \times A \times R \to S \times R$ to perform Monte Carlo simulations:

$$(s', z') = g(s, a, \varphi) \tag{4}$$

Simulating this model for an action sequence $(a_1, a_2, a_3, ...)$ under a scenario $(s_0, \varphi_1, \varphi_2, ...)$ generates a simulation trajectory $(s_0, a_1, s_1, z_1, a_2, s_2, z_2, ...)$. The collection of all trajectories form a *DESPOT tree* (Fig. 1).

A DESPOT tree is a sparse belief tree conditioned on the sampled scenarios. Each node of the tree contains a set of scenarios, whose starting states form an approximate representation of a belief. The tree starts with an initial belief. It branches on all actions available, but only on observations encountered under the sampled scenarios.

The DESPOT algorithm performs anytime heuristic search and constructs the tree incrementally by iterating on three key steps as follows.

Forward search. DESPOT starts from the root node b_0 and searches a single path down to expand the tree. At each node along the path, DESPOT chooses an action branch and an observation branch optimistically according to the heuristics defined by an upper bound u and a lower bound value l.

Leaf node initialization. Upon reaching a leaf node b, DESPOT fully expands it for one level using all actions and the observations encountered under the sampled scenarios. It then initializes the upper and lower bounds for the new nodes by performing a large number of Monte Carlo simulations.

Backup. After creating the new nodes, the algorithm traverses back to the root and updates the upper and lower bounds for all nodes along the path, according to the Bellman's principle:

$$V(b) = \max_{a \in A} \left\{ \frac{1}{|\mathbf{\Phi}_b|} \sum_{\phi \in \mathbf{\Phi}_b} R(s_\phi, a) + \gamma \sum_{z \in Z_{b,a}} \frac{|\mathbf{\Phi}_{b'}|}{|\mathbf{\Phi}_b|} V(b') \right\} \quad (5)$$

where $\mathbf{\Phi}_b$ is the set of scenarios visiting a node b, V stands for both the upper bound and lower bound values, and $b' = \tau(b, a, z)$ represents a child node of b.

DESPOT repeats the three steps until the gap between the upper and lower bounds at the root, $\epsilon(b_0)$, is sufficiently small, or reaching a maximum time limit. See (Ye et al., 2017) for details.

3.2 HyP-DESPOT

We want to parallelize all key steps of DESPOT to maximize performance gain. These steps, however, differ in their structural properties for parallelization. The two tree search steps, forward search and back-up, are irregular. In contrast, leaf node initialization, which consists of many identical Monte Carlo simulations with different initial states, is regular and "embarrassingly parallel", meaning that the simulations can be easily divided and dispatched to parallel threads with no data sharing or inter-thread communication. HyP-DESPOT leverages CPU and GPU parallelism to treat them separately. It uses the more flexible CPU threads to handle the two irregular tree search steps and uses massively parallel GPU threads to handle the embarrassingly parallel Monte Carlo simulations at leaf nodes.

Compared with CPUs, GPUs suffer from high memory access latency and low single-thread arithmetic performance. GPU main memory latency is usually 400–800 clock cycles (Luitjens, 2011), while CPU main memory latency is much shorter at approximately 15 clock cycles (Intel Corporation, 2018). Double-precision arithmetic instructions on GPUs are also several times slower than those on CPUs (NVIDIA Corporation, 2017; Intel Corporation, 2018). CPU-GPU communication also costs significant time. Effective GPU parallelization requires massively parallel tasks to utilize GPU threads fully and amortize latency penalties.

HyP-DESPOT integrates CPU-based parallel tree search and GPU-based parallel Monte Carlo simulations in a multi-level scheme (Fig. 1). Specifically, HyP-DESPOT launches multiple CPU threads to traverse different paths simultaneously. It uses explorative heuristics to distribute parallel threads across the tree. In the meantime, it guarantees optimality by launching optimistic trials periodically. Concurrent with the parallel tree search, HyP-DESPOT relies on the GPU threads to take over new leaf nodes, expand them, and initialize their children through massively parallel Monte Carlo simulations. To maximally exploit GPU parallelization, HyP-DESPOT further factors the system dynamics model and the observation model, then process the factored elements in parallel within a single simulation step. Finally, when the parallel workload becomes too low at the tail of a search path, HyP-DESPOT switches back to CPU simulation to avoid the overhead of GPU computation.

Algorithm 1: HyP-DESPOT

```
1  Sample K scenarios Φ_{b_0} from the current belief;
2  Create the root belief node b_0;
3  Initialize u(b_0) and l(b_0) in the GPU (line 13&14);
4  ε(b_0) ← u(b_0) − l(b_0);
5  for N threads in parallel (CPU) do
6      while ε(b_0) > 0 and elapsed time < T do
7          b ← b_0;
8          while b is not a leaf node do
9              b ← SelectBestChild(b) (Eqn. 6-10);
10         end
11         for a ∈ A and φ ∈ Φ_b in parallel (GPU) do
12             (s', z) ← g(s_φ, a, φ_φ);
13             l_0(s') ← Rollout(s') (Eqn. 14);
14             u_0(s') ← DefaultUB(s') (Eqn. 13);
15         end
16         Create new nodes for all a and z (Eqn. 12);
17         Backup (Eqn. 5) from b to the root node b_0;
18     end
19 end
20 return a* = arg max_{a∈A} l(b_0, a);
```

Algorithm 1 summaries the basic version of HyP-DESPOT. Details are described in the next two sections.

4 Optimal Parallel Belief Tree Search

The key to parallel belief tree search is the effective distribution of CPU threads over different promising paths. Simply deploying multiple CPU threads for DESPOT tree search does not work well, as the original DESPOT search heuristics are deterministic and all search threads end up on the same tree path. To achieve effective parallelization, HyP-DESPOT introduces exploration bonuses in search heuristics. When a CPU thread traverses a node, HyP-DESPOT uses a modified PO-UCT algorithm to select an action branch and uses a virtual loss mechanism to select an observation branch.

4.1 Search Heuristics for DESPOT

We first describe the original heuristics used in DESPOT. At each node b, DESPOT always traverses the action branch with the maximum upper bound value:

$$a^* = \arg\max_{a \in A} u(b, a) \tag{6}$$

and selects the observation branch leading to a child node b' with the maximum weighted excess uncertainty (WEU):

$$z^* = \arg\max_{z \in Z_{b,a^*}} E(b') \tag{7}$$

$$= \arg\max_{z \in Z_{b,a^*}} \left\{ \epsilon(b') - \frac{|\mathbf{\Phi}_{b'}|}{K} \cdot \xi \epsilon(b_0) \right\} \tag{8}$$

Here $\epsilon(b) = u(b) - l(b)$ represents the gap between the upper and lower bounds in node b. Intuitively, the WEU captures the amount of uncertainty remained in node b' with reference to that in the root node b_0. DESPOT terminates a path if E becomes zero at the current node. The constant ξ controls the target level of uncertainty.

4.2 Scenario-based PO-UCT for Action Selection

PO-UCT (Silver and Veness, 2010) is originally designed to trade off exploitation and exploration for serial belief tree search. It augments the estimated value of an action branch with an exploration bonus capturing the frequency of trying the action. The augmented heuristics thus perform two tasks simultaneously: exploit the known promising actions, and explore less-visited branches to improve the value estimation.

We reformulate PO-UCT to encourage parallel HyP-DESPOT threads to explore different action branches. The new algorithm, *scenario-based PO-UCT*, records a scenario-wise visitation count for each node b, written as $|\mathbf{\Phi}_b|N(b)$, and for each action branch under b, written as $|\mathbf{\Phi}_b|N(b, a)$. $\mathbf{\Phi}_b$ is the set of scenarios encountered at b, and $N(\cdot)$ is the number of times that parallel threads visit the node or the branch.

To select an action branch, the scenario-based PO-UCT augments the estimated upper bound with an exploration bonus:

$$u^+(b, a) = u(b, a) + c_a \sqrt{\frac{log(|\mathbf{\Phi}_b|N(b))}{|\mathbf{\Phi}_b|N(b, a)}} \tag{9}$$

The bonus (the last item) decreases immediately when a CPU thread visits action a under b. Later threads thus tend to explore different, less-visited actions. The scaling factor c_a controls the desired level of exploration. It can be tuned offline using hyper-parameter selection algorithms like Bayesian optimization (Mockus, 1989). We will also show in Section 6.4.5 that HyP-DESPOT is robust to the choice of c_a.

4.3 Virtual Loss for Observation Selection

Execution of an action a at node b can produce different observations in different scenarios. HyP-DESPOT traverses these observation branches in parallel using a virtual loss mechanism.

The first thread visiting the branch (b, a) always selects the maximum-WEU observation and traverses the corresponding child node b'. In the meantime, it appends a virtual loss ζ to the WEU value of b':

$$E^+(b') = E(b') - \zeta(b') \tag{10}$$

Later threads visiting (b, a) are thus encouraged to traverse other observations, until the former thread leaves the branch and releases the virtual loss. As a simple implementation, $\zeta(b')$ can be set proportional to the initial gap of the root node, written as $c_o \epsilon(b_0)$. Again, c_o controls the level of exploration among observation branches and can be tuned offline.

4.4 Optimistic Trials and Optimality

Explorative heuristics are critical for distributing parallel threads across the search tree, but may violate the optimality of the original DESPOT algorithm. To retain optimality, HyP-DESPOT launches an *optimistic trial* for every P trials. The special trial applies the optimistic DESPOT heuristics (Section 4.1), performing only exploitation along the traversed path, while other threads still use the explorative heuristics (Section 4.2 and 4.3). The following theorem states that the optimistic trials guarantee the convergence and the optimality of HyP-DESPOT:

Theorem 1. *Suppose that ϵ_0 is the target gap at the root b_0 to be achieved by the algorithm, and δ is the approximation error of the upper bound (Ye et al., 2017). HyP-DESPOT will converge in finite time. The policy reported by HyP-DESPOT is (1) near-optimal when $\epsilon_0 > 0$, and (2) optimal when $\epsilon_0 = 0, \delta = 0$ and the regularization constant $\lambda > 0$.*

The convergence is not naturally guaranteed by applying optimistic DESPOT heuristics. An optimistic trial may encounter inconsistent information produced by other parallel threads updating the same nodes. The behavior of the optimistic trial can thus differ from DESPOT. Our proof in Appendix A shows that HyP-DESPOT still converges despite this inconsistency of information.

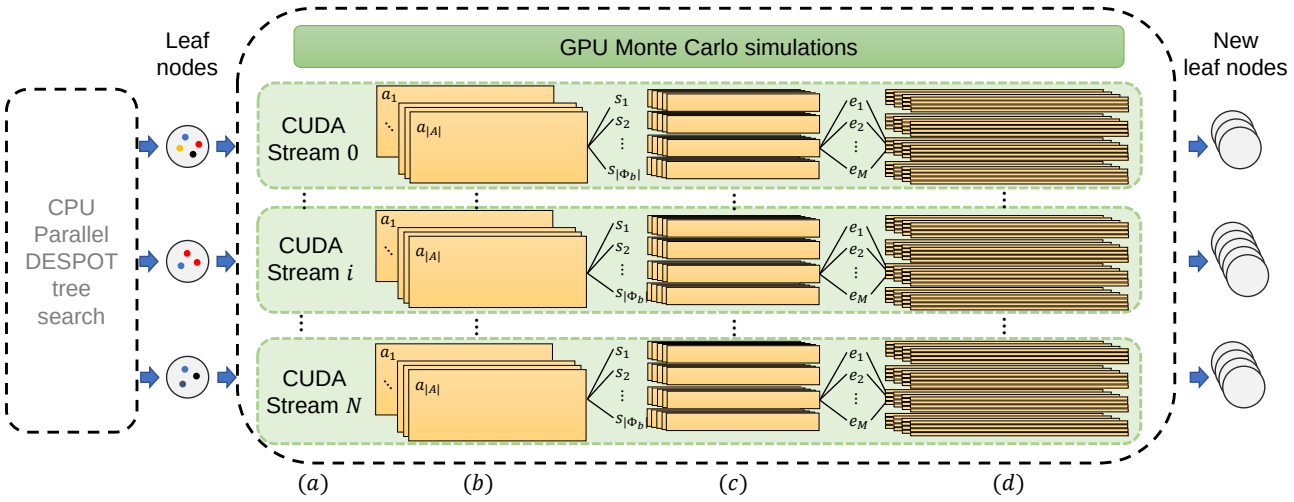


Figure 2: Multi-level parallelization scheme for Monte Carlo simulations in HyP-DESPOT. (a) Node-level parallelism. (b) Action-level parallelism. (c) Scenario-level parallelism. (d) Fine-grained simulation-step level parallelism.

5 Parallel Monte Carlo Simulations

Concurrent with the parallel tree search, HyP-DESPOT passes new leaf nodes to the GPU, expands them and initializes their children by performing parallel Monte Carlo simulations.

HyP-DESPOT expands multiple leaf nodes simultaneously. Each leaf node b is expanded by simulating all possible actions in A using all scenarios in $\mathbf{\Phi}_b$ *in parallel* for one step forward, using the deterministic step function:

$$(s', z) = g(s, a, \phi), \forall \phi \ \in \mathbf{\Phi}_b, \ a \in A \qquad (11)$$

Children belief nodes are created according to the new observations $\{z\}$:

$$b' = \tau(b, a, z), \ a \in A, z \in Z_{b,a} \qquad (12)$$

Scenarios are also updated using the new states $\{s'\}$ and split into children nodes under different observation branches.

The algorithm then calculates the initial upper bound and lower bound for all children nodes $\{b'\}$ *in parallel*. The upper bound u_0 is calculated using a heuristic function $u(\phi)$, and the lower bound l_0 is calculated by simulating a default policy π_0 from the current depth $\Delta_{b'}$:

$$u_0(b') = \frac{1}{|\mathbf{\Phi}_{b'}|} \sum_{\phi \in \mathbf{\Phi}_{b'}} u(\phi) \qquad (13)$$

$$l_0(b') = \frac{1}{|\mathbf{\Phi}_{b'}|} \sum_{\phi \in \mathbf{\Phi}_{b'}} \sum_{t=\Delta_{b'}}^{\infty} \gamma^{t-\Delta_{b'}} R(s_\phi^t(\pi_0), a_{\pi_0}^t) \qquad (14)$$

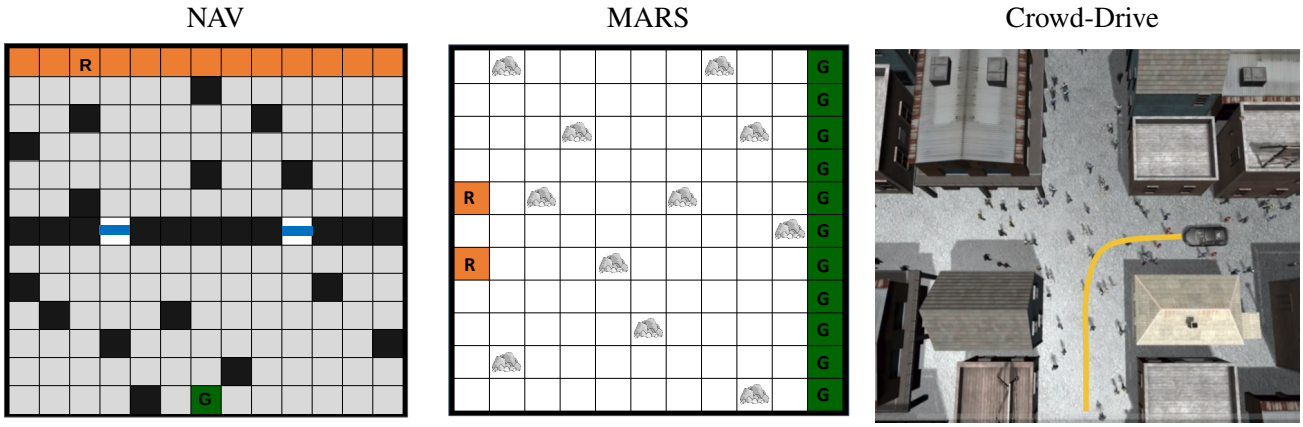
where $s_\phi^t(\pi_0)$ represents the state at time step t encountered under scenario ϕ after applying the sequence of actions $\{a_{\pi_0}^t\}$ determined by the default policy π_0. In practice, we only perform the simulation until a maximum depth D, after which the future value is estimated by a heuristic function $l(\phi)$.

HyP-DESPOT parallelizes all computations in Eqn. (11), (13) and (14) in the GPU. Modern GPUs have a hierarchical computational architecture, e.g., CUDA (NVIDIA Corporation, 2017). GPU functions are launched as "kernels" and are executed by a pool of parallel GPU threads. The thread pool consists of multiple thread blocks further partitioned into "warps" of 32 threads executing in lock-step.

Following the CUDA architecture, we also parallelize the Monte Carlo simulations in hierarchical levels (Fig. 2), processing in parallel leaf nodes, actions, scenarios, and elements within a single simulation step. At the node level, HyP-DESPOT processes multiple leaf nodes concurrently. In the action and the scenario level, HyP-DESPOT performs Monte Carlo simulations for different expansion actions and scenarios simultaneously. Finally, within a simulation step g, HyP-DESPOT parallelizes the factored dynamics or observation models (if available) at a fine-grained level.

5.1 Node-Level Parallelism and Kernel Concurrency

Fig. 2a illustrates the node-level parallelism. HyP-DESPOT associates each CPU thread with a CUDA stream (NVIDIA Corporation, 2017). When the thread reaches a leaf node, it launches a GPU kernel, *MC_simulation*, to perform the computations defined in



(a) (b) (c)

Figure 3: Evaluation tasks.

Eqn. (11), (13) and (14). The *MC_simulation* kernels launched by multiple CPU threads execute independently and concurrently in the GPU.

5.2 Action-Level and Scenario-Level Parallelism

The *MC_simulation* kernel assigns simulations for independent actions in A and scenarios in $\mathbf{\Phi}_b$ to GPU thread blocks and threads, respectively. For each leaf node b, the kernel performs *update*, *expansion*, and *roll-out* in parallel. It first gathers scenarios in b from its parent, and *updates* them to the current search depth by applying the last action in the history. The leaf node is fully *expanded* by considering all actions and simulating all scenarios (Eqn. (11)), producing information for its children nodes such as updated scenarios, rewards, and observations. Then, the kernel computes the upper bounds (Eqn. (13)) and lower bounds by performing *roll-outs* (Eqn. (14)) using the new scenarios. All computed information is finally returned to the corresponding CPU thread to construct children nodes (Eqn. (12)). Once the new nodes are ready, the CPU thread resumes back to the tree search. Fig. 2b-2c summarize the action- and scenario- level parallelism.

5.3 Parallelism within a Simulation Step

The dynamics or observation models in large-scale problems often have multiple independent elements. For example, an environment may have multiple robots or objects moving independently. We can thus factor the models in the step function g into fine-grained parallel tasks (Fig. 2(d)) such as transitions of a vehicle and pedestrians in Crowd-Drive. These factored tasks can be heterogeneous to each other, causing serialized executions of GPU threads. To avoid serialization, HyP-DESPOT assigns each task to an independent thread warp. This fine-grained parallelism enables higher GPU utilization. It also reduces the memory usages in GPU blocks, as each block needs to process fewer scenarios.

5.4 Hybrid Expansion

Despite the hierarchical and fine-grained parallelization, the benefit of GPU simulations vanishes with the search depth. As scenarios diverge to different observation branches along a search path, leaf nodes deep down the tree may contain only few scenarios. In consequence, GPU simulation kernels will be dominated by communication overheads. The problem becomes more prominent when the simulation is computationally simple or when a task has a large observation space that makes scenarios diverge quickly.

HyP-DESPOT overcomes the vanishing benefit problem by switching back to CPU expansions when necessary. When the number of scenarios in a node b falls below a threshold (set to 2 empirically), the corresponding thread copies GPU scenarios back to the host memory and switches to CPU expansions for the rest of the trial. CPU and GPU expansions in different trials are executed simultaneously by parallel threads. The process is thus referred to as "hybrid expansions".

6 Experimental Results

We evaluated HyP-DESPOT on three robot planning tasks under uncertainty (Fig. 3): navigation with a partially known map (NAV), multi-agent rock sample (MARS), and autonomous driving in a crowd (Crowd-Drive). NAV has an enormous state space of size $|S| = 169 \times 2^{124}$ because the map is unknown. MARS has 625 actions, producing a huge tree to be searched. Finally, Crowd-Drive has an enormous observation space

with more than 10^{112} observations and a complex dynamics model, and we evaluated HyP-DESPOT both in simulation and on a real robot vehicle.

We compare HyP-DESPOT and its variations with the original DESPOT algorithm and GPU-DESPOT that performs GPU parallelization only. Our results show that HyP-DESPOT speeds up DESPOT by up to several hundred times. GPU parallelization provides significant performance gain and integration with CPU parallelization offers additional benefits.

Results also suggest that algorithmic components such as explorative heuristics, optimistic trials, and hybrid expansions play significant roles. Explorative heuristics improves both the parallelism of the search and the quality of the solution. Optimistic trials not only guarantee the convergence of HyP-DESPOT, but also improve the practical performance. Hybrid expansions can bring additional speedup when a problem has simple roll-out policies or large observation spaces.

The performance benefits of HyP-DESPOT also depend on the inherent parallelism that a task affords. Our results suggest that generally, large state and action spaces have a positive effect on parallelization, and large observation space has a negative effect.

Details are presented in the subsections below.

6.1 Evaluation Tasks

6.1.1 Navigation with a Partially Known Map

A robot starts from a random position at the top border of a 13×13 map, and travels to its goal in the bottom via one of the two alternatively open gates on the middle wall (colored in blue in Fig. 3a). The map is only partially-known to the robot. The known grids (black grids in Fig. 3a) help the robot localize itself, but they look identical to each other. Other grids (grey in Fig. 3a) are unknown to the robot and have 0.1 probability of being occupied.

In each step, the robot can stay or move to its eight neighboring grids. Moving of the robot can fail with a small probability 0.03, while the observation of each neighboring grid (OCCUPIED or FREE) can be wrong with 0.03 probability. The robot receives a small motion cost (-0.1) for each step it moves. Staying still is discouraged by a penalty of -0.2. If the robot hits an obstacle, it receives a crash penalty (-1). When the goal is reached, the robot receives a goal reward (+20), and the world terminates.

NAV has an huge state space $|S| = 169 \times 2^{124}$. To navigate successfully, the robot has to reason about both localization and map uncertainties, and plan for a sufficiently long horizon to precisely pass the open gate and reach the goal.

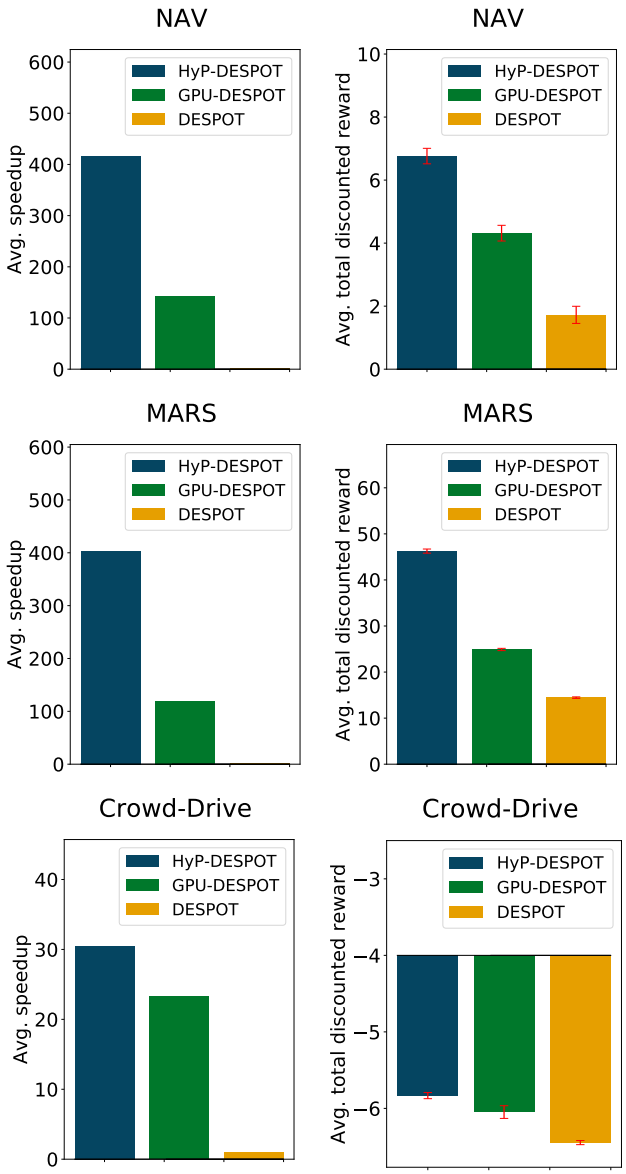


Figure 4: Performance of HyP-DESPOT and GPU-DESPOT, compared with DESPOT in the tree evaluation tasks: average speedup (left column) and average total discounted reward (right column).

6.1.2 Multi-Agent Rock Sample

To test the performance of HyP-DESPOT on tasks with many actions, we modify *Rock Sample*, a well-established benchmark, to *multi-agent Rock Sample* (Fig. 3b) which requires centralized planning. In multi-agent Rock Sample(n, m), two robots cooperate to explore a $n \times n$ map and sample m rocks distributed across the map. The robots aim to sample as many GOOD rocks as possible together and leave the map via the east border. The robots are mounted with noisy sensors to detect

whether a rock is GOOD or BAD, with accuracy decreasing exponentially with the sensing distance. In each step, each robot can either move to the four neighboring grids, or SENSE a specific rock. If a robot reaches a rock, it can SAMPLE it, and receive a +10 reward if the rock is GOOD, or a -10 reward if the rock is BAD. Finally, a robot receives a +10 reward upon reaching the east border. The world terminates when both robots exit the map.

We test HyP-DESPOT on multi-agent Rock Sample(20,20). It has a large action space containing 625 actions, producing a belief tree with a very high branching factor.

6.1.3 Autonomous Driving in a Crowd

We also evaluate HyP-DESPOT on a real-world robotic task: an autonomous vehicle driving through a dense crowd (Fig. 3c). We first conduct a quantitative study in simulation (Fig. 3c), then provide qualitative demonstrations on a real robot vehicle in Section 6.5.

The simulation is extended from the task in (Bai et al., 2015). A vehicle drives among a crowd of pedestrians (Fig. 3c), trying to reach its goal within 200 time steps while taking care of 20 nearest pedestrians. To achieve high fidelity simulation, we model both attentive and distracted pedestrians in the environment. Attentive pedestrians move according to a state-of-the-art motion model, PORCA (Luo et al., 2018), which assumes that pedestrians cooperatively avoid collision with other while optimizing its navigation efficiency. In contrast, distracted pedestrians take straight-line paths towards their goals and don't cooperate with others. Gaussian noise is added to pedestrians' walking directions to simulate noisy transition and sensing. The simulated scene contains roughly 30% of attentive pedestrians and 70% of distracted pedestrians. The vehicle can observe positions and velocities of itself and all pedestrians around it, but cannot directly know the goals of individual pedestrians, which has to be inferred from past observations.

We let the vehicle drive along a pre-planned path, and control its speed online using HyP-DESPOT. In each time step, the vehicle can choose to ACCELERATE, DECELERATE, or MAINTAIN its speed, so that it avoids collision with pedestrians and drives efficiently and smoothly. However, both ACCELERATE and DECELERATE of the vehicle can fail with a small probability (0.01). Rewards in this task follow the setting in (Bai et al., 2015).

Crowd-Drive requires HyP-DESPOT to hedge against a variety of uncertainties: the vehicle's noisy control, hidden intentions of pedestrians, and their noisy transitions.

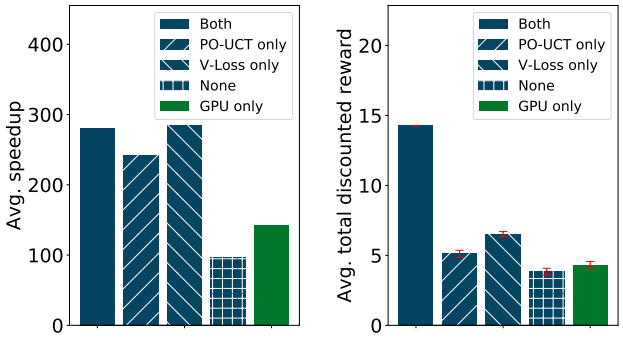


Figure 5: The effect of disabling tree search heuristics of HyP-DESPOT in NAV. "Both" indicates using both the scenario-based PO-UCT and the virtual loss; "None" means using neither of the them.

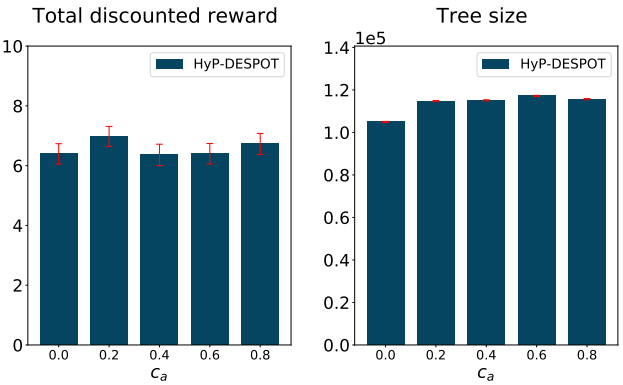


Figure 6: The effect of changing the exploration factor, c_a, on the performance of HyP-DESPOT in NAV.

6.2 Performance Evaluation

To study the computational efficiency, we measure the speedup of HyP-DESPOT and GPU-DESPOT over the serial DESPOT algorithm. The speedup measures the ratio between the tree sizes been constructed within a given planning time. If any of the algorithms overuse the planning time (when expanding the root node), we further normalize the tree sizes by the actual planning time. Our results (Fig. 4) show that HyP-DESPOT achieved high speedup in all evaluation tasks. By constructing larger belief trees, HyP-DESPOT also generates higher quality solutions evaluated with the average total discounted reward collected by the robot(s).

All experiments were conducted on a server with two Intel(R) Xeon(R) Gold 6126 CPUs running at 2.60GHz, a GeForce GTX 1080Ti GPU (11 GB VRAM), and 256 GB main memory. NAV and MARS are solved using 1 second planning time, as in standard online planning set-

Table 1: Detailed performance measurements of DESPOT (K=100), GPU-DESPOT (K=1000), and HyP-DESPOT (K=1000) on Crowd-Drive. The efficiency of a drive is calculated by T_{max}/T where T_{max} is the time limit and T is the traveling time to reach the goal. If the vehicle fails to reach the goal in a specific run, the efficiency is treated as zero.

	Total discounted reward	Efficiency	Collision rate	# Decelerations
DESPOT	-6.4 $\pm$ 0.006	0.0144	0.0 $\pm$ 0.0	12.7 $\pm$ 0.03
GPU-DESPOT	-6.05 $\pm$ 0.083	1.848	2.4e-4 $\pm$ 5.2e-5	12.6 $\pm$ 0.05
HyP-DESPOT	-5.83 $\pm$ 0.040	1.812	1.0e-4 $\pm$ 3.0e-5	12.7 $\pm$ 0.04

ting. For Crowd-Drive, we use 10 Hz control frequency (0.1 second planning time).

In NAV, HyP-DESPOT and GPU-DESPOT achieve 416.3 and 142.6 times speedup over DESPOT, respectively. As a result, HyP-DESPOT and GPU-DESPOT achieve 292.2% and 150.3% higher total discounted rewards than DESPOT, respectively.

For MARS, HyP-DESPOT and GPU-DESPOT achieve 403.2 and 119.2 times speedup over DESPOT, and bring up to 220.1% and 72.3% of improvements on the total discounted rewards, respectively.

The Crowd-Drive task affords a limited level of parallelism, primarily because of the huge observation space, $Z > 10^{112}$, causing scenarios to diverge along the search paths. Fig. 4 shows that pure GPU parallelization still brought 23.2x speedup over DESPOT benefiting from the fine-grained parallelism within each simulation step. With additional CPU parallelization, HyP-DESPOT achieved 30.5x speedup and significantly higher rewards. Detailed measurements in Table 1 show that HyP-DESPOT significantly improved the navigation efficiency from DESPOT. It also reduced the collision rate from GPU-DESPOT, thus delivered safer drives.

6.3 Benefits of Parallelization

In this section, we examine the benefit of three key components in HyP-DESPOT that enables us to integrate CPU and GPU parallelization: the explorative heuristics, the optimistic trials, and the hybrid expansions. The following results show that they are all critical for the efficiency of the algorithm.

6.3.1 Explorative Heuristics

We illustrate the importance of the explorative heuristics using ablation analysis with the NAV task. Fig. 5 shows that launching multiple CPU threads without explorative heuristics (*None*) did not bring any speedup over GPU-DESPOT (*GPU*) because the threads tend to traverse identical paths. Enabling either the PO-UCT or the virtual loss (*PO-UCT* only and *V-Loss* only) brought significant performance gain over GPU-DESPOT. When both heuristics were functioning (*Both*), HyP-DESPOT achieved the best performance.

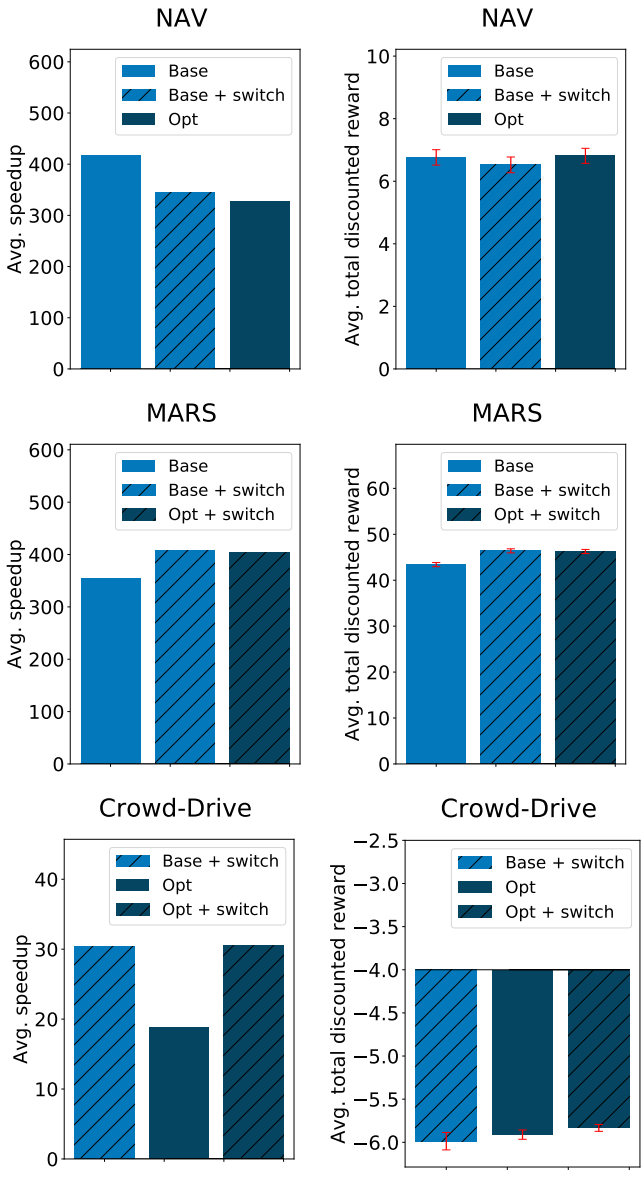


Figure 7: The effect of using or not using optimistic trials (with *opt* or *base* flags) and hybrid expansions (with or without the *switch* flag) on the performance of HyP-DESPOT in the evaluation tasks.

We further test the robustness of HyP-DESPOT with respect to the exploration factor c_a in Eqn. 6. Fig. 6 shows the performance of HyP-DESPOT with different c_a applied to action branch selection. The results indicate that HyP-DESPOT is robust to the value of c_a and it can be conveniently chosen for a range of similar tasks.

6.3.2 Optimistic Trials

In this section, we analyze the practical effect of optimistic trials, beyond its contribution to the theoretical guarantee. We first compare the performance of HyP-DESPOT with and without optimistic trials. Then, we conduct a fine-grained analysis by varying the period P of launching optimistic trials.

Our results (Fig. 7) show that optimistic trials can improve the solution quality without building larger trees. This is because the optimistic trials better exploit existing information in the tree. HyP-DESPOT with optimistic trials (with *opt* tags) generally has lower speedup than variants without them (with *base* tags), but their performance still match up with or exceed that of the *base* variants. For example, in NAV, the *opt* variant gave much lower speedup than the *base* variant, but achieved similar performance. In Crowd-Drive, *opt*+*switch* had similar speedup as *base*+*switch*, but improved the rewards significantly.

On the other hand, optimistic trials can also harm the efficiency HyP-DESPOT due to reduced parallelism, particularly when they are launched too frequently. Fig. 8 illustrates the effect of using different periods P (a low P means launching optimistic trials more frequently). As an extreme case, when P is 0, parallel trials are all optimistic, and they end up traversing the same paths, making the search highly inefficient. The choice of P is a trade-off between exploration and exploitation in the context of parallel tree search. Our analyses (Fig. 8) indicate that $P \approx 5$ is the optimal setting for the NAV task.

6.3.3 Hybrid Expansions

The benefit of GPU expansions depends on the complexity of the roll-outs and the size of the observation space. When the roll-outs are simple, GPU computation time will be dominated by thread synchronization and memory latency; When the observation space is large, scenarios diverge fast along search paths, leaving most of leaf nodes with a small scenario set to be parallelized. Our evaluation tasks cover both cases. MARS has a very simple default policy that blindly moves the robots to the east border. In Crowd-Drive, the huge observation space comprises the information of all involved agents. Our results show that variants using hybrid expansions (with *switch* tags) outperformed other variants significantly in both problems (Fig. 7). In MARS, *base* + *switch* achieved 14.9% higher speedup than *base*, improving 7% on the rewards. In Crowd-Drive, *opt* + *switch* improved the speedup by 62.3% and achieved the best driving performance.

6.4 Effects of Key Parameters

The performance of HyP-DESPOT also relies on the choice of parameters, such as the number of sampled scenarios and the planning time, as well as the underlying property of the task, such as the size of the action space and the number of elements in the step function. This section studies the effect of algorithm parameters and task properties on two algorithms: HyP-DESPOT and HyP-DESPOT-Base. The former uses optimistic trials in the search while the latter does not. Hybrid expansions are used when they are beneficial.

6.4.1 Number of Scenarios

Generally, problems with large state spaces can benefit significantly from our parallelization. Large $|S|$-problems require more scenarios to cover the state space and representative outcomes of actions. These scenarios create many independent Monte Carlo simulations, thus increase the parallelism of the algorithm. We used NAV (Section 6.1.1), with $|S| = 169 \times 2^{124}$, as an example to study this effect. We varied K from 100 to 5000 for HyP-DESPOT while keeping the planning time unchanged. Fig. 9 shows the high scalability of HyP-DESPOT with respect to K. In contrast to the decaying performance of DESPOT, the performance of HyP-DESPOT increased when sampling more scenarios by offering higher speedup and searching larger trees. The search depth of HyP-DESPOT, on the other hand, decreased with K, indicating that it searches a wider tree to produce robust decisions. Noticeably, HyP-DESPOT with optimistic trials consistently outperformed HyP-DESPOT-Base with all K's while always building smaller trees.

6.4.2 Planning Time

Now we fix K in NAV and vary the planning time per step T. We ran HyP-DESPOT for $T = 0.25s$, and set

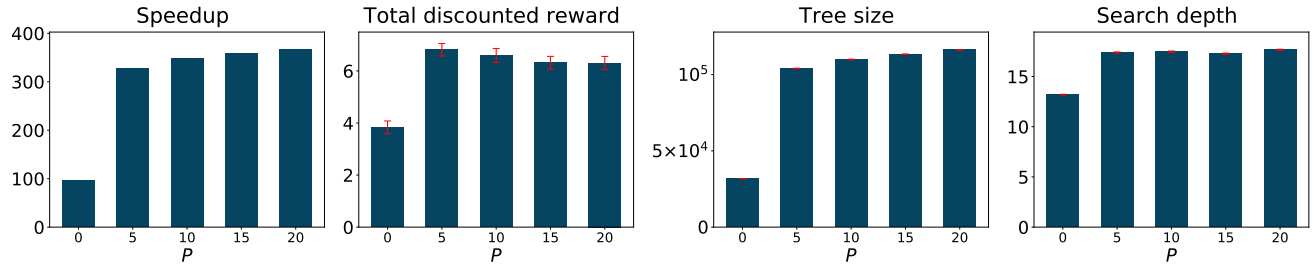


Figure 8: The effect of the period of launching optimistic trials, P, on the performance of HyP-DESPOT in NAV.

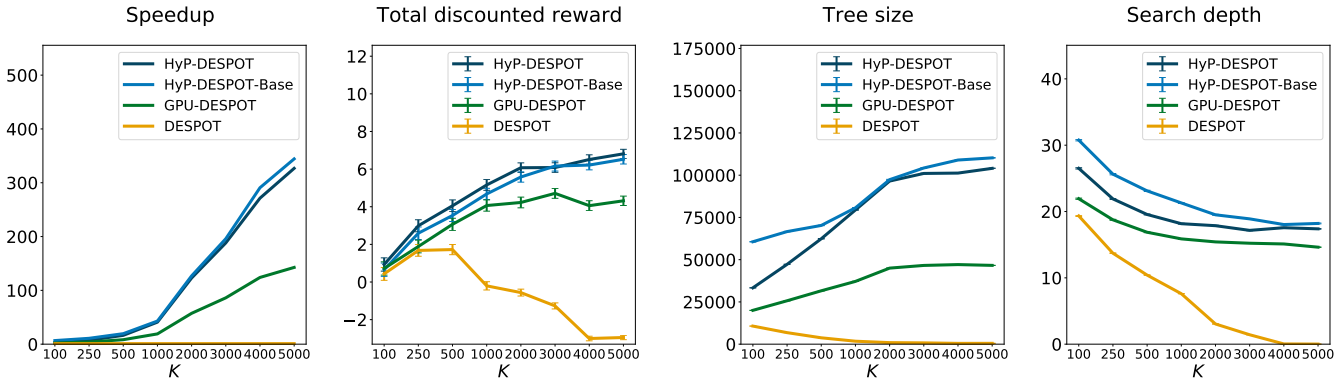


Figure 9: The effect of the number of scenarios, K, on the performance of HyP-DESPOT in NAV.

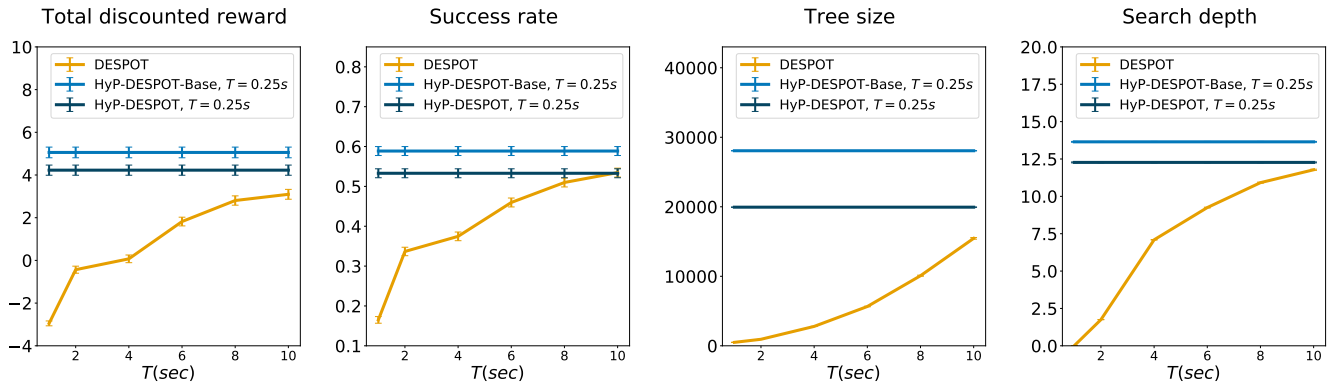


Figure 10: The effect of the planning time T. In this experiment, DESPOT used a sequence of increasing planning time and HyP-DESPOT used $T = 0.25$ sec in NAV. The success rate is defined as the proportion of trials where the robot reaches the goal within 60 steps.

$T = 1 \sim 10s$ for DESPOT. Fig. 10 shows that DESPOT takes much more time ($>$ 40x) to reach a comparable performance with HyP-DESPOT. HyP-DESPOT can construct much larger and deeper trees, even when the latter uses 10s planning time. The performance gap decreases when DESPOT uses more time, but the convergence trend becomes slow after $T = 10s$. Here, HyP-DESPOT-Base with no optimistic trials performed better than HyP-DESPOT when $T = 0.25s$. This is because $0.25s$ is insufficient for obtaining reliable value estimations. Thus it is more important to explore (using HyP-DESPOT heuristics) than to exploit (using DESPOT heuristics).

6.4.3 Size of the Action Space

HyP-DESPOT favors large action spaces, as they provides high parallelism to leverage. We evaluated HyP-DESPOT on MARS (11,11), (15,15), and (20,20), with $|A|$ to be 256, 400, and 625, respectively. We fixed K and T for all the tests. Fig. 11 shows that HyP-DESPOT provided higher speedup when $|A|$ increases, and achieved much higher rewards than DESPOT for all $|A|$'s.

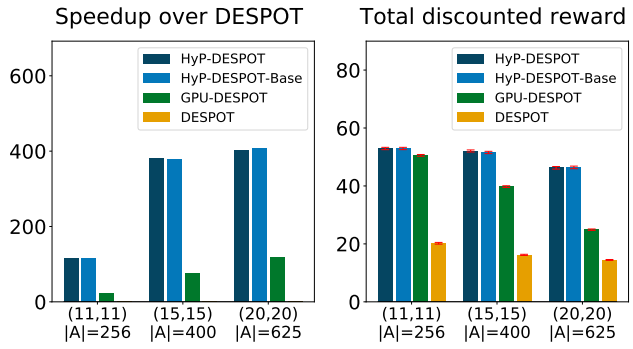


Figure 11: The effect of the action space size on the performances of HyP-DESPOT, GPU-DESPOT, and DESPOT. The experiment is conducted on three MARS tasks with 256, 400, and 625 actions.

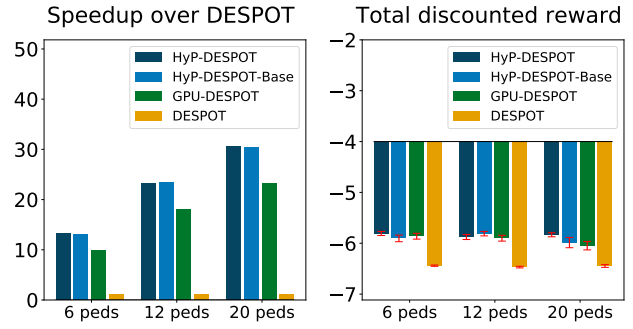


Figure 12: The effect of factored elements in the simulation model on the performances of HyP-DESPOT, GPU-DESPOT, and DESPOT. The experiment is conducted on three Crowd-Drive tasks considering 6, 12 and 20 nearby pedestrians.

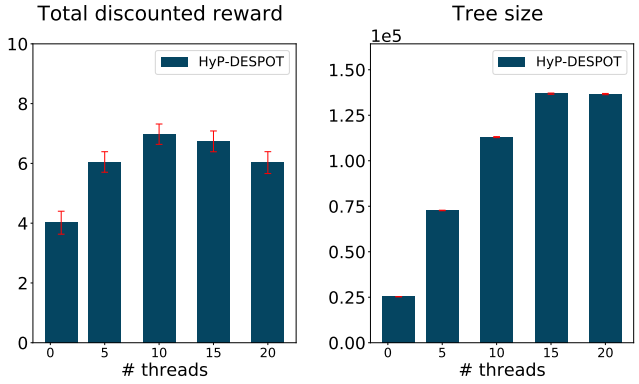


Figure 13: The effect of the number of parallel CPU threads on the performance of HyP-DESPOT in NAV.

6.4.4 Number of Elements in the Factored Model

Large observation spaces, Z, however, usually restrict the speedup of HyP-DESPOT by diverging scenarios into observation branches. Fortunately, many large-Z problems can still leverage the fine-grained parallelism within a simulation step. For example, our results in Crowd-Drive (Fig. 12) show that HyP-DESPOT achieved higher speedups when simulating more pedestrians in planning. Consequently, HyP-DESPOT improved driving performance significantly in all cases.

6.4.5 Number of CPU threads

Fig. 13 shows how the performance of HyP-DESPOT scales with the numbers of CPU threads. In summary, using more CPU threads for search increases the tree size consistently, but the benefit diminishes once the GPU cores are saturated. Further, larger search trees do not necessarily guarantee higher rewards. The parallel search threads are designed to be explorative. More threads generally lead to wider trees, instead of deeper ones. Thus too many CPU threads can harm the performance for tasks such as NAV, which requires long-horizon planning.

6.5 Experiments on an Autonomous Vehicle

We implemented HyP-DESPOT on a robot vehicle to drive among pedestrians on a campus plaza (Fig. 14). Sensors on the vehicle include two LIDARs, an inertia measurement unit (IMU), and wheel encoders. We use a SICK LMS151 LIDAR, mounted on top of the vehicle, for pedestrian detection, and a SICK TiM551 LIDAR, mounted at the front, for localization. The maximum vehicle speed is 1 m/s. HyP-DESPOT runs on an Ethernet-connected laptop with an Intel Core i7-4770R CPU running at 3.90 GHz, a GeForce GTX 1050M GPU (4 GB VRAM), and 16 GB main memory.

We apply a two-level approach to control the vehicle (Bai et al., 2015). At the high level, we use the Hybrid A* algorithm (Stanley, 2006) to plan a path. At the low level, we run HyP-DESPOT to compute the vehicle speed along the planned path. The maximum planning time is 0.3s. So the system re-plans both the path and the speed at approximately 3 Hz.

Our experiments on a campus plaza show that the autonomous vehicle can drive safely, efficiently, and smoothly, among many pedestrians. Running on only a middle-end laptop, HyP-DESPOT is able to handle 20 real pedestrians around the vehicle, enabling it to drive in a much denser crowd than demonstrated in previous work.

Fig. 14 shows the vehicle interacting with crowds of pedestrians walking towards different goals. The driving trajectories are generated by Hybrid A* according to the free spaces around pedestrians. In the meantime, HyP-DESPOT infers the intentions and the future motion of pedestrians to plan for optimal speed control. For example, in Fig. 14a, the vehicle encounters an approaching

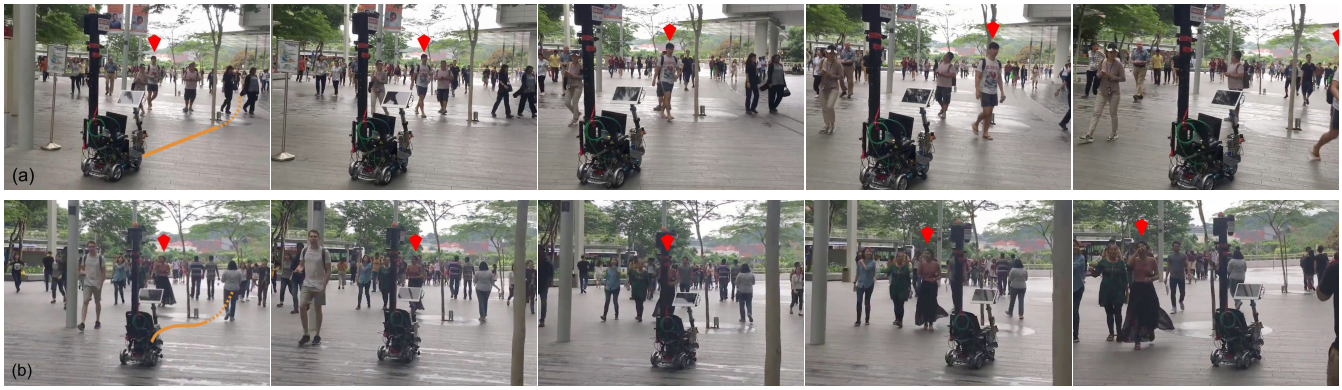


Figure 14: The robot vehicle drives among pedestrians on a campus plaza. See also the accompanying video at https://youtu.be/YaIpHGZOCsE.

pedestrian intending to cut through the vehicle's planned path. To avoid collisions, the vehicle decelerates and gives way to the pedestrian. In another case (Fig. 14b), a pedestrian walks in the opposite direction towards the vehicle. The vehicle maneuvers to leverage the potential free space on the right and maintains its driving speed.

7 Conclusion

This paper presents HyP-DESPOT, a massively parallel algorithm for online planning under uncertainty. HyP-DESPOT performs parallel DESPOT tree search in multi-core CPUs and massively parallel Monte Carlo simulations in GPUs. To achieve effective parallelization, it uses explorative heuristics to distribution parallel trials. The optimality of the search is preserved by launching optimistic trials periodically. When possible, HyP-DESPOT factors a complex system model and performs fine-grained parallelization to achieve further performance gain. By integrating CPU and GPU parallelism in a hybrid and multi-level scheme, HyP-DESPOT achieves hundreds of times speedup over DESPOT on several large-scale planning tasks under uncertainty.

The parallelization scheme underlying HyP-DESPOT can be easily generalized to other belief tree search algorithms, e.g., POMCP. HyP-DESPOT can also be combined with importance sampling (Luo et al., 2019) to further improve performance. These are some directions that we plan to explore in the near future.

Acknowledgement

This research is partially supported by the NUS AcRF Tier 1 grant R-252-000-A87-114 and the ONR Global and AFRL grant N62909-18-1-2023.

References

H. Bai, S. Cai, N. Ye, D. Hsu, and W. S. Lee. Intention-aware online pomdp planning for autonomous driving in a crowd. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 454–460, May 2015.

N. A. Barriga, M. Stanescu, and M. Buro. Parallel uct search on gpus. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–7, Aug 2014.

D. P. Bertsekas and D. A. Castanon. Rollout algorithms for stochastic scheduling problems. In *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, volume 2, pages 2143–2148 vol.2, Dec 1998.

J. Bialkowski, S. Karaman, and E. Frazzoli. Massively parallelizing the rrt and the rrt. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3513–3518, Sept 2011.

P. Cai, I. Chandrasekaran, J. Zheng, and Y. Cai. Automatic Path Planning for Dual-Crane Lifting in Complex Environments Using a Prioritized Multiobjective PGA. *IEEE Transactions on Industrial Informatics*, 14(3):829–845, 2018a.

P. Cai, Y. Luo, D. Hsu, and W. S. Lee. Hyp-despot: A hybrid parallel algorithm for online planning under uncertainty. In *Robotics: Science and Systems XIV*, 2018b.

T. Cazenave and N. Jouandeau. On the parallelization of uct. In *Proceedings of the Computer Games Workshop*, pages 93–101, 2007.

D. J. Challou, M. Gini, and V. Kumar. Parallel search algorithms for robot motion planning. In *IEEE International Conference on Robotics and Automation*, pages 46–51. IEEE, 1993.

G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008.

E. Chong, R. Givan, and H. Soo Chang. A framework for simulation-based network control via hindsight optimization. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 2, pages 1433 – 1438, Dec 2000.

P.-A. Coquelin and R. Munos. Bandit algorithms for tree search. *arXiv preprint cs/0703062*, 2007.

R. He, E. Brunskill, and N. Roy. Efficient planning under uncertainty with macro-actions. *J. Artif. Int. Res.*, 40(1):523–570, Jan. 2011.

Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2018. URL `https://intel.ly/2lgN4rc`.

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

C. Johnson, L. Barford, S. M. Dascalu, and F. C. Harris. CUDA implementation of computer go game tree search. In *Information Technology: New Generations: 13th International Conference on Information Technology*, pages 339–350, Cham, 2016.

M. Kearns, Y. Mansour, and A. Y. Ng. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Mach. Learn.*, 49(2-3): 193–208, Nov. 2002.

M. Koval, D. Hsu, N. Pollard, and S. Srinivasa. Configuration lattices for planar contact manipulation under uncertainty. In *Algorithmic Foundations of Robotics XII—Proc. Int. Workshop on the Algorithmic Foundations of Robotics (WAFR)*. 2016.

H. Kurniawati, D. Hsu, and W. S. Lee. Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces. In *In Proc. Robotics: Science and Systems*, 2008.

T. Lee and Y. J. Kim. Massively parallel motion planning algorithms under uncertainty using pomdp. *Int. J. Rob. Res.*, 35(8):928–942, July 2016.

J. K. Li, D. Hsu, and W. S. Lee. Act to see and see to act: Pomdp planning for objects search in clutter. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5701–5707, Oct 2016.

Z. Lim, L. Sun, and D. Hsu. Monte carlo value iteration with macro-actions. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1287–1295. 2011.

T. Lozano-Prez and P. A. O'Donnell. Parallel robot motion planning. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1000–1007, April 1991.

J. Luitjens. GPU computing webinar, 2011. URL https://bit.ly/2YUvefG.

Y. Luo, P. Cai, A. Bera, D. Hsu, W. S. Lee, and D. Manocha. Porca: Modeling and planning for autonomous driving among many pedestrians. *IEEE Robotics and Automation Letters*, 3(4):3418–3425, 2018.

Y. Luo, H. Bai, D. Hsu, and W. S. Lee. Importance sampling for online planning under uncertainty. *The International Journal of Robotics Research*, 38(2-3): 162–181, 2019.

J. Mockus. *Bayesian approach to global optimization: theory and applications*. Springer, 1989.

NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. URL https://bit.ly/1IyiYCS. Version 8.0.

J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for pomdps. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1025–1030, 2003.

E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki. Sampling-based roadmap of trees for parallel motion planning. *IEEE Transactions on Robotics*, 21(4):597–608, 2005.

S. Prentice and N. Roy. The belief roadmap: Efficient planning in belief space by factoring the covariance. *The International Journal of Robotics Research*, 28 (11-12):1448–1465, 2009.

K. Rocki and R. Suda. Large-scale parallel monte carlo tree search on gpu. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 2034–2037, May 2011.

S. Ross and B. Chaib-Draa. Aems: An anytime online search algorithm for approximate policy refinement in large pomdps. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, pages 2592–2598, 2007.

S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall series in artificial intelligence. Prentice Hall, 2002.

D. Silver and J. Veness. Monte-carlo planning in large pomdps. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2164–2172. 2010.

T. Smith and R. Simmons. Heuristic search value iteration for pomdps. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, pages 520–527, 2004.

T. S. Stanley. The robot that won the darpa grand challenge: research articles. *Journal Robotics System*, 23 (9):661–692, 2006.

J. Van Den Berg, P. Abbeel, and K. Goldberg. Lqg-mp: Optimized path planning for robots with motion uncertainty and imperfect state information. *The International Journal of Robotics Research*, 30(7):895–913, 2011.

K. H. Wray and S. Zilberstein. A parallel point-based pomdp algorithm leveraging gpus. In *AAAI Fall Symposium on Sequential Decision Making for Intelligent Agents (SDMIA)*, pages 95–96, 2015.

N. Ye, A. Somani, D. Hsu, and W. S. Lee. Despot: Online pomdp planning with regularization. *Journal of Artificial Intelligence Research*, 58:231–266, 2017.

Appendices

A Convergence Proof of HyP-DESPOT

A.1 Definitions

Definition 1 (HyP-DESPOT tree)**.** *The belief tree that HyP-DESPOT constructed is called a "HyP-DESPOT tree", denoted as $\mathcal{H}$.*

Definition 2 (Consistency of a HyP-DESPOT tree)**.** *A HyP-DESPOT tree $\mathcal{H}$ is called "consistent" if for all node b in $\mathcal{H}$, the upper bound value and the lower bound value satisfy the Bellman's equation:*

$$V(b) = \max_{a \in A} \left\{ \frac{1}{|\mathbf{\Phi}_b|} \sum_{\phi \in \mathbf{\Phi}_b} R(s_\phi, a) + \gamma \sum_{z \in Z_{b,a}} \frac{|\mathbf{\Phi}_{b'}|}{|\mathbf{\Phi}_b|} V(b') \right\} \quad (15)$$

where $\mathbf{\Phi}_b$ is the set of scenarios visiting a node b, V stands for both the upper bound and lower bound values, and $b' = \tau(b, a, z)$ represents a child node of b.

Definition 3 (Optimistic trial)**.** *An "optimistic trial" is a trial launched every P_{th} exploration trials that uses the DESPOT heuristics described in (Ye et al., 2017) rather that the HyP-DESPOT heuristics described in Section 4.2 and 4.3. For the simplicity of notations, we let $P = N$ in the proof. Other choices of P work similarly.*

A.2 Proof

Lemma 2. *The duration for an exploration trial ψ is bounded by a constant δT.*

Proof. If only a single thread is searching the tree, the duration will be bounded by $D \max\{\delta T_e, \delta T_f, \delta T_b\}$, where D is the maximum search depth of the tree, δT_e, δT_f, and δT_b are the maximum duration for expanding a node, forward traversing a node, and performing backup at a node, respectively.

Now suppose that N threads are simultaneously traversing the tree. We consider one of the N threads, ψ. For an arbitrary node b along its traversed path, the time spent on visiting b path can only be increased by waiting for other threads traversing, expanding or backing-up the same node. Note that there are at most $N-1$ threads blocking the ψ at b. Thus the duration for ψ visiting node b is bounded by $N \max\{\delta T_e, \delta T_f, \delta T_b\}$. Thus, the total exploration time is bounded by $\delta T = DN \max\{\delta T_e, \delta T_f, \delta T_b\}$. □

Lemma 3. *If a HyP-DESPOT tree $\mathcal{H}$ is inconsistent at time t, a working trial (a trial that is currently not visiting the root node) have expanded at least one node within duration $[t - \delta T, t]$*

Proof. New information in the tree can only be created by expanding a new node. Additionally, after each trial finishes the backup step and returns to the root, information along the path will be consistent again. All working trials at time t were started within $[t - \delta T, t]$ (Lemma 2). If none of them had expanded a single node, information in the tree should be consistent, a contradiction. Thus we conclude that a working trial should have expanded one node within duration $[t - \delta T, t]$. □

Lemma 4. *Consider an optimistic trial $\tilde{\psi}$. Denote as t_0 the time when $\tilde{\psi}$ started from the root of the tree. The algorithm either has expanded or closed a gap of a node during $[t_0 - \delta T, t_0]$, or will expand or close a gap of a node within $[t_0, t_0 + \delta T]$.*

Proof. Additionally denote the time point when $\tilde{\psi}$ reach a leaf node of the HyP-DESPOT tree as t_n. There exist only three possible cases:

1. The HyP-DESPOT tree at time step t_0 is inconsistent;
2. The HyP-DESPOT tree at time step t_0 is consistent, but it has been updated during $[t_0, t_n]$;
3. The HyP-DESPOT tree at time step t_0 is consistent, and it has not been updated during $[t_0, t_n]$;

For case (1), since the tree is not consistent, by applying Lemma 3, we conclude that a node should have been expanded during $[t_0 - \delta T, t_0]$.

For case (2), the tree has been updated during $[t_0, t_n]$, meaning that at least a node has been expanded during $[t_0, t_n] \in [t_0, t_0 + \delta T]$.

Lastly, for case (3), the HyP-DESPOT tree should always be consistent during the duration $[t_0, t_n]$. Therefore, trial $\tilde{\psi}$ behaves exactly like in the serial DESPOT algorithm, and Theorem 4.1-4.2 in (Ye et al., 2017) applies, implying that $\tilde{\psi}$ will at least expand one node or close the gap of one node along the current path (which happens during $[t_0, t_0 + \delta T]$). □

A.2.1 Proof of Theorem 1

Proof. We first show that HyP-DESOPT with N parallel threads expands or closes the gap of at least one node after every $\delta T'$, where $\delta T'$ is some fix amount of time. Note that the difference between the starting time of an optimistic trial $\tilde{\psi}$ and its previous optimistic trial $\tilde{\psi}'$ is at

most $N\delta T$. Applying Lemma 4, HyP-DESPOT should expand or close the gap of at least one node for at least every $\delta T' = (N + 2)\delta T$ time duration. Thus, the proof of Theorem 4.2 in (Ye et al., 2017) and the conclusion apply here, meaning that HyP-DESPOT will expand all useful belief nodes and terminate in finite time. Upon termination, HyP-DESPOT constructs the full DESPOT tree, achieving near-optimal or optimal policy in case (1) and (2), respectively. The same regret bound given in Theorem 3.2 for DESPOT (Ye et al., 2017) holds here. □