APPEARED IN
Proc. Robotics: Science & Systems (RSS), 2018

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*—Planning under uncertainty is critical for robust robot performance in uncertain, dynamic environments, but it 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 near real-time online planning performance for complex tasks with large state, action, and observation spaces. Specifically, we propose Hybrid Parallel DESPOT (HyP-DESPOT), 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 and performs parallel Monte-Carlo simulations at the leaf nodes of the search tree using GPUs. 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.**

I. 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 known exactly. 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 information, environment changes on the belief. To formalize this, a planning algorithm performs look-ahead search in a *belief tree*, in which each tree node represents a belief, and parent and child nodes are connected by action-observation pairs (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 [25] 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 [25], and DESPOT has shown strong performance in various robotic tasks, including autonomous driving [1] and manipulation [12].

Our goal here is to scale up DESPOT further through parallelization and achieve near real-time performance for online planning under uncertainty in complex tasks with large state, action, and observation spaces. 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.

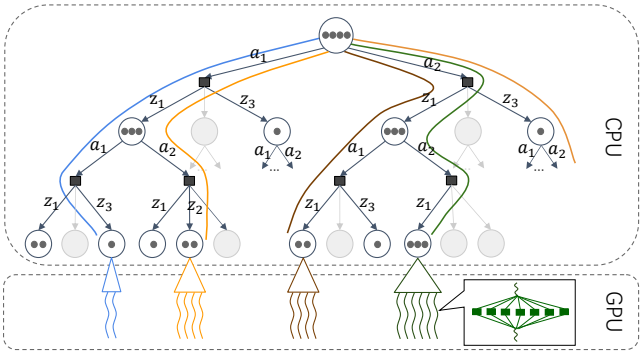


Fig. 1: An overview of HyP-DESPOT. Each node of the belief tree (gray) represents a belief. A parent node and a child node, with associated beliefs b and b' respectively, are connected by an action-observation pair (a, z), indicating that the belief transitions from b to b', when a robot, with initial 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 (inset figure).

First, 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, with minimum communication among the threads.

Second, HyP-DESPOT uses GPUs to perform massively parallel 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 in order to generate sufficient parallel workload and benefit from large-scale GPU parallelization.

To the best of our knowledge, HyP-DESPOT is the first massively parallel algorithm for online planning under uncertainty. Our experiments show that HyP-DESPOT achieves significant speedup and higher-quality solutions, compared with the original DESPOT algorithm.

II. Background

A. Online Planning under Uncertainty

A robot operates in a partially observable stochastic environment. The robot has state space S, action space A, and observation space Z. We model the robot's stochastic dynamics with a probability function $T(s, a, s') = p(s'|s, a)$ for $s, s' \in S$ and $a \in A$. We model the noisy sensors with another probability function $O(s', a, z) = p(z|a, s')$, for $s' \in S$, $a \in A$, and $z \in Z$.

There are two distinct approaches to planning under uncertainty: offline (e.g., [10, 13, 17, 22]) and online (e.g., [19, 21, 25]). Offline planning leverages offline computation to reason about all future contingencies in advance and achieves faster execution time online. In contrast, online planning focuses the computation on the contingency currently encountered and scales up to much more complex tasks.

For online planning under uncertainty, 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 high computational cost, Monte Carlo sampling is a powerful idea to make it efficient in practice. Early examples include the roll-out algorithm [3], sparse sampling [9], hindsight optimization [6], and AEMS [19]. Two state-of-the-art online planning algorithms, POMCP [21] and DESPOT [25], 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 guarantees significantly better worst-case performance theoretically. More importantly, DESPOT offers better opportunities for parallelization, as it generates a large number of Monte Carlo simulations, each corresponding to a sampled scenario, and processes them simultaneously rather sequentially, as POMCP does.

B. Parallel Planning under Uncertainty

Planning under uncertainty can be formalized 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 fully observable [20]. Parallelization is a powerful tool that has been exploited to speed up both MDP planning [2, 5, 8, 18] and offline POMDP planning [11, 24].

The main focus of parallel MDP planning is parallel MCTS: leaf parallelization [4], root parallelization [4], and tree parallelization [5]. Leaf parallelization performs multiple roll-outs from leaf tree nodes in parallel. Root parallelization builds multiple trees in parallel to select the best action. Both use multiple CPU threads. One may also combine leaf and root parallelism and exploit large-scale GPU parallelization. Rocki and Suda [18] proposes a block parallelism scheme, which uses GPUs to parallelize roll-out requests from multiple trees. Barriga et al. [2] extends the idea to a multi-block parallelism scheme by additionally expanding the children of leaf nodes. However, increasing the number of trees in root parallelization or the number of roll-outs in leaf parallelization often has limited benefits, because the multiple tree searches are independent without information sharing and the same computation is repeated many times. Tree parallelization addresses this issue by cooperatively searching a shared tree using multiple CPU threads. The challenge here is to minimize the communication overheads. HyP-DESPOT exploits both tree parallelization and leaf parallelization, and integrates them in a CPU-GPU hybrid parallel model for belief tree search.

Offline POMDP planning computes beforehand a policy for all contingencies, thus inducing a huge number of independent tasks for parallelization. gPOMDP [11] parallelizes the Monte Carlo value iteration (MCVI) algorithm [13] by performing Monte Carlo simulations for multiple beliefs, candidate actions, policy graph nodes, etc., in parallel in GPUs. A similar idea [24] is used to parallelize the point-based value iteration (PBVI) algorithm [17].

Offline planning has almost unlimited offline computation time to derive a solution. In contrast, online robot planning under uncertainty is usually given a small fixed amount of time to choose the best action in real time. Parallelism is much more

important for online planning, in order to scale up to complex tasks, but is rarely explored. Our work aims to fill this gap.

III. Hybrid Parallel DESPOT

A. DESPOT

For completeness, we provide a brief summary of the DESPOT algorithm. See [25] for details. 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, $\phi = (s_0, \varphi_1, \varphi_2, ...)$, contains a sampled initial state s_0 and random numbers $\varphi_1, \varphi_2, ...$, that determinize the uncertain outcomes of future actions and observations.

A DESPOT tree is a sparse belief tree conditioned on the sampled scenarios (Fig. 1). Each node of the tree contains a set of scenarios, whose starting states form an approximate representation of a belief. The tree starts from an initial belief. It branches on all actions, but only on observations encountered under the sampled scenarios.

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

1) 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 and a lower bound value, u and l.

2) 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 scenarios visiting b. It then initializes the upper and lower bounds for the new nodes, by performing a large number of Monte Carlo simulations.

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

$$V(b) = \max_{a \in A} \left\{ \frac{1}{|\mathbf{\Phi}_b|} \sum_{\phi \in \mathbf{\Phi}_b} R(s_\phi, a) + \gamma \sum_{z \in Z_{b,a}} \frac{|\mathbf{\Phi}_{b'}|}{|\mathbf{\Phi}_b|} V(b') \right\} \quad (4)$$

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 b_0 is sufficiently small or the maximum time limit is reached.

B. HyP-DESPOT Overview

We want to parallelize all key steps of DESPOT, but they exhibit different 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 communication among the threads. HyP-DESPOT uses a CPU-GPU hybrid parallel model to treat them separately. It uses the more flexible CPU threads to handle the two irregular tree search steps. It uses massively parallel GPU threads to handle the embarrassingly parallel Monte Carlo simulations for leaf node initialization.

GPUs, however, suffer from high memory access latency and low single-thread arithmetic performance, compared with CPUs. The memory latency is 400∼800 clock cycles for GPUs [14], while it is about 15 clock cycles for CPUs [7]. Double-precision arithmetic instructions on GPUs are also several times slower than those on CPUs [7, 16]. Efficient GPU parallelization requires massively parallel tasks to fully utilize GPU threads 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 simultaneously search different paths and discover new leaf nodes. At the same time, It relies on the GPU threads to take over these leaf nodes, expand them, and initialize their children through massively parallel Monte Carlo simulations. Further, HyP-DESPOT factors the dynamics model and the observation model within a single simulation step and simulates the factored elements in parallel, in order to maximally exploit GPU parallelization. The next two subsections present the parallel tree search (Section III-C) and the parallel Monte Carlo simulations (Section III-D) in details.

C. Parallel DESPOT Tree Search

Simply deploying multiple CPU threads for DESPOT tree search fails, because the original DESPOT search heuristics are deterministic and all search threads would end up on the same tree path. A key idea of parallel DESPOT tree search is to effectively distribute the threads across the tree over many different paths. To achieve this, HyP-DESPOT adds exploration bonuses to the search heuristics. For a specific CPU thread, HyP-DESPOT uses a modified PO-UCT algorithm to select an action branch and uses a virtual loss mechanism to select an observation branch.

1) Heuristics in DESPOT: For the completeness of this paper, we first describe the original heuristics used in DESPOT. At each node b, DESPOT always traverse the action branch with the maximum upper bound value:

$$a^* = \arg\max_{a \in A} u(b, a) \quad (5)$$

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

$$z^* = \arg\max_{z \in Z_{b,a^*}} E(b') \quad (6)$$

$$= \arg\max_{z \in Z_{b,a^*}} \left\{ \epsilon(b') - \frac{|\mathbf{\Phi}_{b'}|}{K} \cdot \xi \epsilon(b_0) \right\} \quad (7)$$

Here $\epsilon(b) = u(b) - l(b)$ represents the gap between the upper and lower bounds in node b. Intuitively, the WEU value $E(b')$ captures the amount of uncertainty contained in node b' with

reference to that in the root node b_0. DESPOT terminates an exploration path if E becomes zero at the current node. The constant ξ controls the target level of uncertainty to be achieved by the search.

2) Scenario-based PO-UCT for Action Branches: The PO-UCT algorithm [21] is originally designed for trading off exploitation and exploration during the serial belief tree search. It augments the value of an action branch with an exploration bonus that captures its frequency of been tried, such that the search not only exploits the known promising directions, but also reduces the uncertainty in less-explored branches.

We reformulate the PO-UCT algorithm to distribute parallel CPU threads across action branches under HyP-DESPOT nodes. The new algorithm, *scenario-based PO-UCT*, respects that a belief node b in DESPOT is always traversed by a set of scenarios, $\mathbf{\Phi}_b$. It 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 $|\mathbf{\Phi}_b|N(b,a)$, then uses the following augmented upper bound to select an action branch for a thread:

$$u^+(b,a) = u(b,a) + c_a\sqrt{\frac{log(|\mathbf{\Phi}_b|N(b))}{|\mathbf{\Phi}_b|N(b,a)}} \tag{8}$$

The last term in Eqn. (8) is the exploration bonus, which is updated immediately when each CPU thread visits b. The scaling factor c_a controls the desired level of exploration for CPU threads among action branches, and can be tuned offline using hyper-parameter selection algorithms like Bayesian optimization [15].

3) Virtual Loss for Observation Branches: Observation branches under an action captures possible outcomes of the action under different scenarios. It is beneficial to have CPU threads explore multiple of them simultaneously. To achieve this, HyP-DESPOT appends a virtual loss ζ to the WEU value of an observation branch, once it is being traversed by a thread:

$$E^+(b') = E(b') - \zeta(b') \tag{9}$$

This virtual loss discourages following threads to traverse the same branch, until the current thread leaves the branch and releases it.

In effect, the first thread will always traverse the maximum-WEU observation branch. Later peer threads tend to explore other promising branches. As a simple implementation, $\zeta(b')$ can be set proportional to the initial gap of the root node, written as $c_o\epsilon(b_0)$, where c_o controls the level of exploration among observation branches and can be also tuned offline.

D. Parallel Monte Carlo Simulation

During the search, HyP-DESPOT uses the GPU to continually take over leaf nodes, and perform parallel Monte Carlo simulations to expand them and initialize their children.

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

$$s', z = g(s, a, \boldsymbol{\phi}), \forall \boldsymbol{\phi} \ \in \mathbf{\Phi}_b, \ a \in A \tag{10}$$

We then calculate *in parallel* the initial upper bound and lower bound values for all children belief nodes $\{b'\}$:

$$b' = \tau(b, a, z), \ a \in A, z \in Z_{b,a} \tag{11}$$

The upper bound is calculated using a heuristic function $u(\boldsymbol{\phi})$, and the lower bound is calculated by simulating a default policy π_0 from the current depth $\Delta_{b'}$:

$$u_0(b') = \frac{1}{|\mathbf{\Phi}_{b'}|}\sum_{\boldsymbol{\phi}\in\mathbf{\Phi}_{b'}} u(\boldsymbol{\phi}) \tag{12}$$

$$l_0(b') = \frac{1}{|\mathbf{\Phi}_{b'}|}\sum_{\boldsymbol{\phi}\in\mathbf{\Phi}_{b'}}\sum_{t=\Delta_{b'}}^{\infty} \gamma^{t-\Delta_{b'}} R(s^t_{\boldsymbol{\phi}}(\pi_0), a^t_{\pi_0}) \tag{13}$$

where $s^t_{\boldsymbol{\phi}}(\pi_0)$ represents the state at time step t, updated using the step function g, and determined by scenario $\boldsymbol{\phi}$ and the sequence of actions $\{a^t_{\pi_0}\}$. In practice, we only perform the simulation until a maximum depth D, after which the future value is estimated by a heuristic function $l(\boldsymbol{\phi})$.

HyP-DESPOT parallelizes all computations in Eqn. (10), (12) and (13) in the GPU, but creates the new nodes (Eqn. (11)) in the CPU.

Modern GPUs have a hierarchical computational architecture, CUDA [16]. GPU functions are launched as "kernels" and are executed by a pool of parallel GPU threads. The thread pool is organized into multiple thread blocks that are further partitioned into "warps" of 32 threads executing in lock-step.

Following this architecture, we also parallelize the Monte Carlo simulations in hierarchical levels (Fig. 2), including the node-, action-, scenario-, and factored-model- level. The node-level parallelism handles concurrently multiple leaf nodes passed by CPU threads. The action-level and the scenario-level parallelisms perform Monte Carlo simulations for different expansion actions and scenarios simultaneously. Finally, the factored-model level parallelism parallelizes the factored dynamics or observation models (if available) within a simulation step g in a fine-grained level.

1) Node-level Parallelism and Kernel Concurrency: When a CPU thread reaches a leaf node, HyP-DESPOT launches a GPU kernel, *MC_simulation*, to perform the computations defined in Eqn. (10), (12) and (13). HyP-DESPOT associates each CPU thread with a CUDA stream [16], such that *MC_simulation* kernels for leaf nodes execute independently and concurrently in the GPU. Fig. 2(a) shows the node-level parallelism.

2) Action-level and Scenario-level Parallelisms: The *MC_simulation* kernel for a leaf node b performs several tasks—*update*, *expansion*, and *roll-out*—in parallel, with independent actions in A and scenarios in $\mathbf{\Phi}_b$ assigned to individual thread blocks and threads in the GPU (Fig. 2(b)-(c)) respectively. The kernel first gathers scenarios corresponding to b from its parent, and *updates* them to the current search

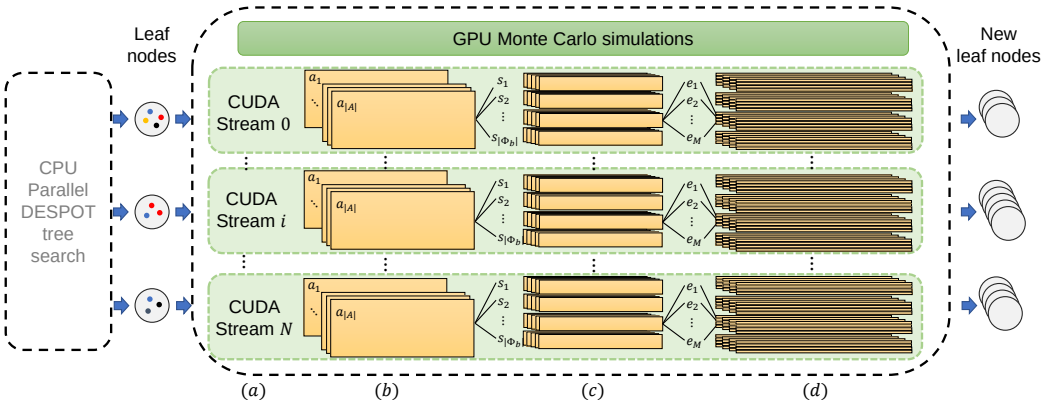


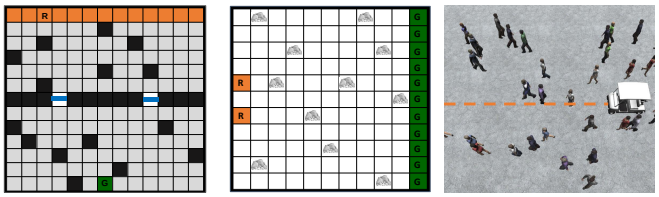
Fig. 2: Multi-level parallelization scheme for Monte Carlo simulations in HyP-DESPOT. (a) Node-level parallelism. (b) Action-level parallelism. (c) Scenario-level parallelism. (d) Fine-grained factored-model level parallelism.

depth by applying the last action in the history. The leaf node is then fully expanded by one-level following Eqn. (10), producing information for its children nodes including scenarios, rewards, and observation labels. For these children, the kernel further computes their upper bounds using Eqn. (12), and initializes their lower bounds by performing *roll-outs* (Eqn. (13)). Finally, the kernel returns the scenario-based observations, step rewards, and initial bounds to the corresponding CPU thread, in order to branch scenarios and prepare children nodes (Eqn. (11)). Once the new nodes are ready, the CPU thread resumes back to the tree search.

3) Factored-model level Parallelism: The dynamics or observation models in large-scale problems often have multiple independent elements. For example, an environment may have multiple robots or obstacles moving independently. We can thus factor the models defined in the step function g into fine-grained parallel tasks (Fig. 2(d)). HyP-DESPOT dispatches these tasks to GPU thread warps, in order to avoid serialization problem caused by heterogeneous tasks, e.g., transitions of a vehicle and pedestrians in an autonomous driving task. By applying the factored-model level parallelism, HyP-DESPOT achieves higher GPU utilization, and consumes less per-block memory as each GPU block needs to process less scenarios.

IV. Experimental Results

We evaluated HyP-DESPOT in simulation on three large-scale planning tasks under uncertainty: navigation with a partially known map, multi-agent rock sample, and autonomous driving in a crowd. The navigation task has an enormous state space of size $|S| = 169 \times 2^{124}$, because of map uncertainty. The multi-agent rock sample task has 625 actions, requiring HyP-DESPOT to search a very large tree. Finally, the autonomous driving task has a huge 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 with the original DESPOT algorithm and GPU-DESPOT, which 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 provides additional benefits.



(a) (b) (c)

Fig. 3: Large-scale planning tasks for evaluating HyP-DESPOT. (a) Navigation with a partially known map. (b) Multi-agent rock sample. (c) Autonomous driving in a crowd.

The performance benefits of HyP-DESPOT 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.

A. Evaluation Tasks

1) Navigation with a Partially-Known Map (Navigation): 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. 3(a)). The map is only partially-known to the robot. The known grids (black grids in Fig. 3(a)) help the robot localize itself, but they look identical to each other. Other grids (grey in Fig. 3(a)) are unknown to the robot and have 0.1 probability to be occupied.

In each step, the robot can stay or move to its eight neighboring positions. 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. Staying still is discouraged by a small penalty (-0.2). The robot receives a small motion cost (-0.1) for each step it moves. 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.

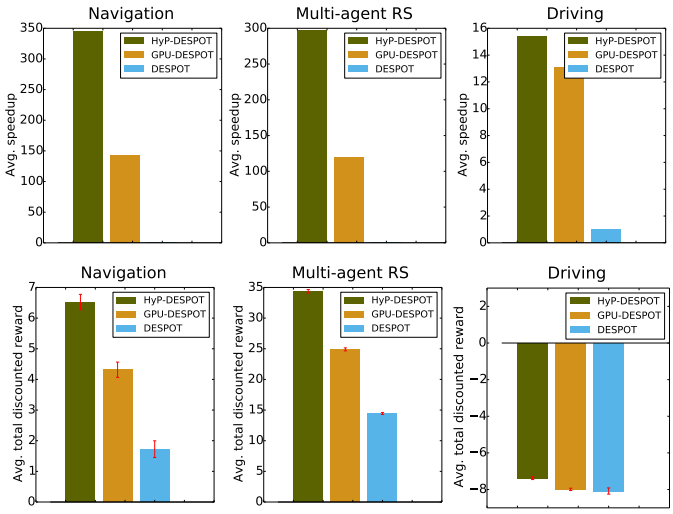


Fig. 4: Performance of HyP-DESPOT and GPU-DESPOT, compared with DESPOT: average speedup (top row) and average total discounted reward (bottom row).

This navigation task 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 sufficiently long horizon to precisely pass the open gate.

2) Multi-Agent Rock Sample (MARS): To test the performance of HyP-DESPOT on tasks with many actions, we modify *Rock Sample*, a well-established benchmark, to the *multi-agent Rock Sample* problem (Fig. 3(b)) 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 in total 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 exist the map.

We test HyP-DESPOT on multi-agent Rock Sample(20,20). It has a large action space containing 625 actions, requiring HyP-DESPOT to explore a wide belief tree.

3) Autonomous Driving in a Crowd (Driving): We also evaluate HyP-DESPOT on a real-world robotic task: an autonomous vehicle driving through a dense crowd (Fig. 3(c)). We first conduct a quantitative study in a simulation environment (Fig. 3(c)), then demonstrate the application on a real robot vehicle in Section IV-H.

In this task [1], a vehicle drives along its planed path among a crowd of pedestrians (Fig. 3(c)), with its speed controlled by a POMDP planner. The vehicle tries to reach its goal within 200 time steps, while taking care of 20 nearest pedestrians.

TABLE I: Performance comparisons of DESPOT (K=100), GPU-DESPOT (K=1000), and HyP-DESPOT (K=1000) on the autonomous driving task.

	Collision rate	Traveled distance	Decelerations
DESPOT	0.00177 $\pm$ 0.0002	12.493 $\pm$ 0.1	8.175 $\pm$ 0.07
GPU-DESPOT	0.000496 $\pm$ 0.00008	9.131 $\pm$ 0.07	6.744 $\pm$ 0.05
HyP-DESPOT	0.000612 $\pm$ 0.00008	10.034 $\pm$ 0.08	6.045 $\pm$ 0.05

We assume that pedestrians move towards their goals with a uniform speed and Gaussian noises on their heading directions. The vehicle can fully observe positions and velocities of itself and all pedestrians around it, but cannot directly know the goals of individual pedestrians, which information has to be inferred from past observations. 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 follows the setting in [1].

The autonomous driving task has a huge state space and requires to hedge against uncertainties in both the vehicle's motion and the intended navigation goals of pedestrians.

B. Performance Comparison

To illustrate the computational efficiency of HyP-DESPOT and GPU-DESPOT, we measure their speedup over DESPOT, defined as the ratio between the sizes of the belief trees been constructed within a given planning time. If any of the algorithms over-use the planning time (when expanding the root node), we further normalize the tree sizes by the true planning time. Our experimental results show that, constructing a larger belief tree leads to higher solution quality as measured by the total discounted reward.

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. The navigation task and multi-agent RS are solved using 1 second planning time, as in standard online planning setting. For the driving task, we use 10 Hz control frequency (0.1 second planning time).

For navigation with a partially-known map, HyP-DESPOT and GPU-DESPOT achieve 344.3 and 142.6 times speed-up over DESPOT (Fig. 4(a)), respectively. As a result, HyP-DESPOT and GPU-DESPOT achieve 278% and 150% higher total discounted rewards than DESPOT, respectively.

For multi-agent Rock Sample, HyP-DESPOT and GPU-DESPOT achieve 297.4 times and 119.2 times speedup over DESPOT, and bring up to 137.9% and 72.3% of improvements on the total discounted rewards (Fig. 4(b)), respectively.

The autonomous driving task affords limited level of parallelism, primarily because of the huge observation space, $Z > 10^{112}$, causing scenarios to diverge along the exploration

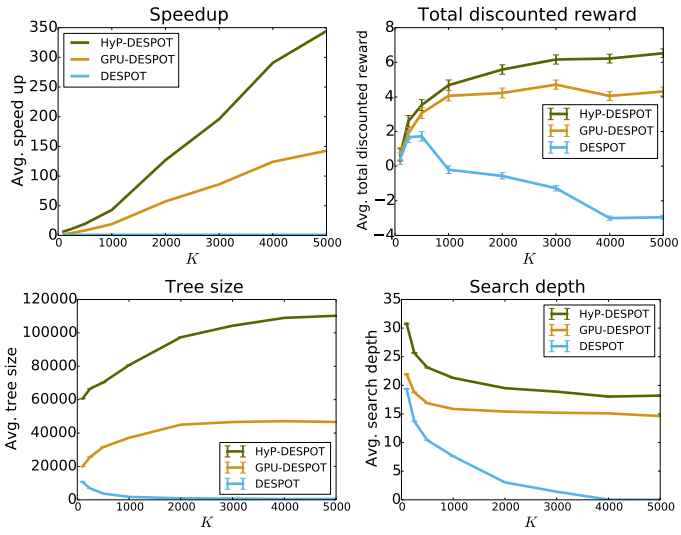


Fig. 5: The effect of the number of scenarios, K, on the planning performance in the Navigation task.

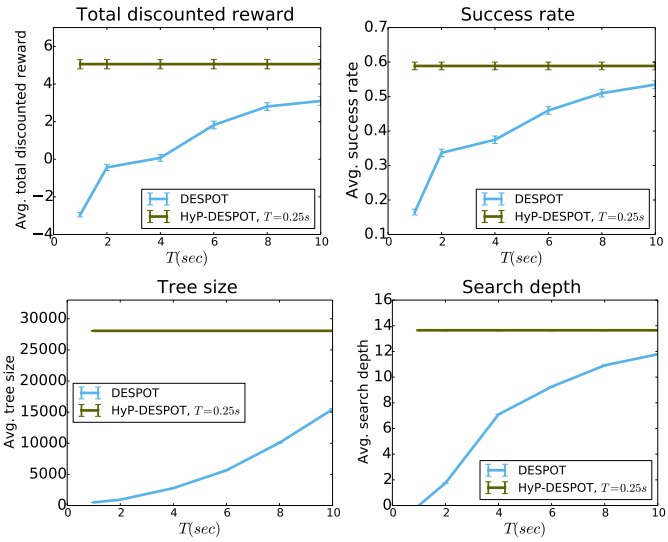


Fig. 6: Performance comparison of DESPOT with increasing planning time T and Hyp-DESPOT with $T = 0.25$ sec in the Navigation task.

paths. HyP-DESPOT still achieves 15.4x speed-up and higher quality solutions (Fig. 4(c)) over DESPOT, because of both the parallel tree search and the fine-grained factored-model level parallelism. Detailed measurements in Table I show that HyP-DESPOT significantly reduces the collision rate from DESPOT, which drives the vehicle over-aggressively, and enables the vehicle to driver faster and smoother than GPU-DESPOT.

C. Effect of the Number of Scenarios

Generally, problems with large $|S|$'s can benefit significantly from the efficiency of HyP-DESPOT. Large $|S|$-problems require more scenarios to cover the states space and representative outcomes of actions and observations for robustness, creating many independent Monte Carlo simulations, and thus increasing the parallelism. To study this effect, we vary K from 100 to 5000 for HyP-DESPOT when solving the Navigation task (Section IV-A1), with $|S| = 169 \times 2^{124}$, while keeping the planning time unchanged. Fig. 5 shows the high scalability of HyP-DESPOT with respect to K: In contrast to the decaying performance of DESPOT, HyP-DESPOT achieves higher speed-up when sampling more scenarios, searching larger trees, and thus generating higher quality solutions. The search depth, on the other hand, decreases with K, indicating that HyP-DESPOT searches a wider tree to produce robust decisions.

D. Effect of the Planning Time

Moreover, we fix K in the Navigation task and vary the planning time per step, T, and show that DESPOT takes much more time (> 40x) to reach a comparable performance with HyP-DESPOT. We run HyP-DESPOT for $T = 0.25s$, and set $T = 1 \sim 10s$ for DESPOT. Performance of the algorithms is measured using both the total discounted reward and the success rate of the robot reaching the goal within 60 steps.

TABLE II: The effect of HyP-DESPOT tree search heuristics in the Navigation task.

	Speedup	Total discounted reward
HyP-DESPOT	344.3	6.52±0.25
HyP-DESPOT_VL	320.3	5.97±0.26
HyP-DESPOT_PO-UCT	297.0	4.89±0.25
GPU-DESPOT	142.6	4.32±0.25

Fig. 6 shows that HyP-DESPOT significantly outperforms DESPOT, by searching a larger and deeper tree, even when the latter uses 10s planning time. The performance gap decreases when DESPOT uses more time, but the trend becomes slow after $T = 10s$.

E. Effect of Heuristics

We further study the effect of the heuristics in HyP-DESPOT, by deactivating one or both of the exploration schemes. We tested the following algorithms: the full HyP-DESPOT, "HyP-DESPOT_VL" with only virtual losses, "HyP-DESPOT_PO-UCT" with only the scenario-based PO-UCT, and GPU-DESPOT with both schemes disabled. Concluding from the speedup and solution quality shown in Table II, both scenario-based PO-UCT and virtual loss contribute to the efficiency of HyP-DESPOT. It brings significant performance gain when enabling one of the schemes, and brings further improvements when both are functioning.

F. Effect of the Size of the Action Space

Large $|A|$ improves the parallelism in Monte Carlo simulations like large K's do. To illustrate this, we evaluated HyP-DESPOT on MARS (11,11), (15,15), and (20,20), with $|A|$ to be 256, 400, and 625, respectively, with K and T fixed

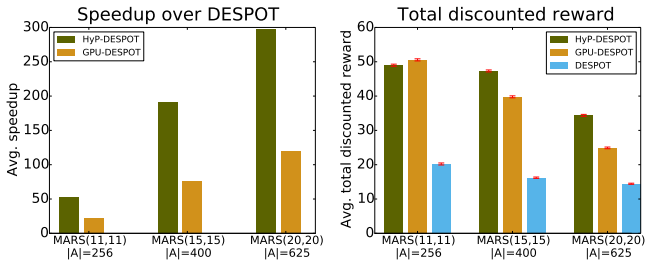


Fig. 7: The performances of HyP-DESPOT, GPU-DESPOT, and DESPOT in the MARS task with increasing action space sizes.

for all the tests. Results in Fig. 7 illustrate that HyP-DESPOT achieves higher speedup when $|A|$ increases, and generates significantly higher quality solutions than DESPOT in all three tasks.

G. Effect of the Number of Elements in Step Function

Large observation spaces, Z, however, usually restrict the performance gain from HyP-DESPOT by diverging scenarios into observation branches. Fortunately, many large-Z problems, e.g., driving among pedestrians (Section IV-A3), can still leverage the fine-grained factored-model level parallelism to improve the performance. By varying the number of pedestrians in the driving task (Fig. 8), we illustrate that HyP-DESPOT benefits from problems with more independent elements within a simulation step. It achieves higher speedups when considering more pedestrians in planning, and achieves significant improvements on the solution quality.

H. Experiments on an Autonomous Vehicle

We implemented HyP-DESPOT on a robot vehicle for autonomous driving among pedestrians on a campus plaza (Fig. 9). The main vehicle sensors consist of two LIDARs, an inertia measurement unit (IMU), and wheel encoders. We use the SICK LMS151 LIDAR, mounted on top of the vehicle, for pedestrian detection, and the SICK TiM551 LIDAR, mounted at the front, for localization. The maximum vehicle speed is 1 m/s. HyP-DESPOT runs on an Ethernet-connected computer 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 [1]. At the high level, we use the Hybrid A* algorithm [23] to plan a path. At the low level, we run HyP-DESPOT to compute the vehicle speed along the planned path using the POMDP model described in Section IV-A3. The maximum planning time for HyP-DESPOT is 0.3s. So it 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.

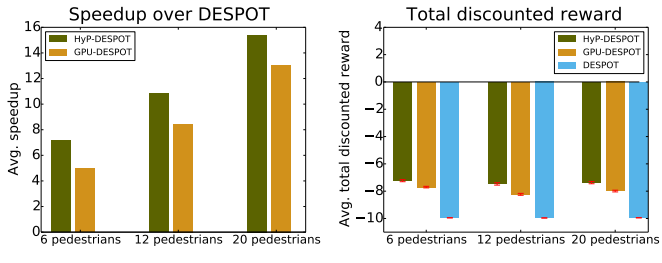


Fig. 8: The performances of HyP-DESPOT, GPU-DESPOT, and DESPOT in the Driving task, with increasing number of pedestrians.

Fig. 9: The robot vehicle drives among pedestrians on a campus plaza.

V. Conclusions

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. When possible, it factors a complex system model and extracts fine-grained parallelism for further performance gain. By integrating CPU and GPU parallelism in a multi-level scheme, HyP-DESPOT achieves significant speedup over DESPOT on several large-scale planning tasks under uncertainty. The parallelization ideas underlying Hyp-DESPOT can be generalized to other belief tree search algorithms, for instance, POMCP.

HyP-DESPOT has two main performance bottlenecks for: communication overhead among CPU threads and unbalanced workload of Monte Carlo simulations in GPU threads. Our next steps include implementing lock-free trees to minimize communication among CPU threads and designing load balancing schemes for GPU parallelization.

Acknowledgment

We thank the anonymous reviewers for their comments and suggestions that have helped to improve the paper. This work is supported by the MoE AcRF grant 2016-T2-2-068 and National Research Foundation Singapore through the SMART IRG program.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik. *Parallel Monte-Carlo Tree Search*, pages 60–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[6] 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.

[7] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2018. URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[8] C. Johnson, L. Barford, S. M. Dascalu, and F. C. Harris. *CUDA Implementation of Computer Go Game Tree Search*, pages 339–350. Springer International Publishing, 2016.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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. Curran Associates, Inc., 2011.

[14] J. Luitjens. GPU computing webinar, 2011. URL https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GlobalMemory.pdf.

[15] J. Mockus. *Bayesian approach to global optimization: theory and applications*, volume 37. Springer Science & Business Media, 2012.

[16] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2017. URL http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#abstract. Version 8.0.

[17] 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*, IJCAI'03, pages 1025–1030, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[18] 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.

[19] 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*, IJCAI'07, pages 2592–2598. Morgan Kaufmann Publishers Inc., 2007.

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

[21] 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. Curran Associates, Inc., 2010.

[22] T. Smith and R. Simmons. Heuristic search value iteration for pomdps. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 520–527, Arlington, Virginia, United States, 2004. AUAI Press.

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

[24] 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.

[25] 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.