Vectorized Online POMDP Planning

Marcus Hoerger, Muhammad Sudrajat, Hanna Kurniawati

***Abstract*— Planning under partial observability is an essential capability of autonomous robots. The Partially Observable Markov Decision Process (POMDP) provides a powerful framework for planning under partial observability problems, capturing the stochastic effects of actions and the limited information available through noisy observations. POMDP solving could benefit tremendously from massive parallelization on today's hardware, but parallelizing POMDP solvers has been challenging. Most of these solvers rely on interleaving numerical optimization over actions with the estimation of their values, which creates dependencies and synchronization bottlenecks between parallel processes that can offset the benefits of parallelization. In this paper, we propose Vectorized Online POMDP Planner (VOPP), a novel parallel online solver that leverages a recent POMDP formulation which analytically solves part of the optimization component, leaving numerical computations to consist of only estimation of expectations. VOPP represents all data structures related to planning as a collection of tensors, and implements all planning steps as fully vectorized computations over this representation. The result is a massively parallel solver with no dependencies or synchronization bottlenecks between parallel processes. Experimental results indicate that VOPP is at least $20\times$ more efficient in computing near-optimal solutions compared to an existing state-of-the-art parallel online solver.**

I. INTRODUCTION

Planning under partial observability is an essential, yet challenging problem for autonomous robots. The Partially Observable Markov Decision Process (POMDP) [1] is a principled framework to solve planning under uncertainty problems. It lifts the planning problem from the robot's state space to its belief space, the space of all probability distributions over the state space. Although solving POMDPs exactly is computationally intractable in general [2], many scalable approximately optimal online solvers have been proposed (reviewed in [3]), and some have been applied to realistic robot applications, such as [4], [5], [6], [7].

However, most POMDP solvers do not exploit massive parallelisation that Graphics Processor Units (GPU) offer. Paralellising POMDP solving is quite involved. POMDP solving requires interleaving of numerical optimisation to find actions with the highest expected total reward and estimation of expected total rewards themselves. When parallelised, this interleaving process creates dependencies that make load balancing difficult. Careful parallelisation strategies, including process synchronization and scheduling for POMDP solving have been proposed [8], [9], [10], [11]. They have significantly improved the scalability of serial approximate POMDP solvers, but come with an overhead that tends to limit the potential benefit of massive parallelisation. In contrast, this paper builds on a recent approach to approximate POMDPs solutions [12] that partially solve the optimisation component analytically, leaving numerical computation only for estimation of expectations.

Specifically, we propose Vectorized Online POMDP Planner (VOPP), a new parallel online POMDP solver based on PORPP [12]. Similar to most online POMDP solvers, VOPP is a tree search-based method: Starting from the current belief, they perform guided belief space sampling to construct a representative belief tree and evaluate different action sequences. However, unlike existing methods, VOPP represents all data structures associated with the belief tree as a collection of tensors. During planning, VOPP iteratively performs guided belief space sampling, followed by backup operations on the belief tree to compute an approximately optimal policy. Crucially, each of these steps is implemented as a sequence of *fully vectorized* computations over the collection of tensors that represent the belief tree. This enables VOPP to fully harness the immense data-parallel throughput of modern GPUs. The result is a massively parallel online POMDP solver – running entirely on the GPU – that uses tens of thousands of parallel simulations to compute a policy, with no explicit synchronization between simulations required.

To the best of our knowledge, VOPP is the first fully vectorized online POMDP solver. Experimental results on three POMDP benchmark problems indicate that VOPP is at least $20\times$ more efficient in computing a near-optimal policy compared to the current state-of-the-art parallel online solver, HyP-DESPOT [11], for problems with large state, action, and observation spaces – for some benchmark, VOPP is more than $100\times$ faster than HyP-DESPOT. VOPP will be released as open source software.

II. BACKGROUND AND RELATED WORK

A. Partially Observable Markov Decision Process (POMDP)

A POMDP provides a general mathematical framework for sequential decision-making under uncertainty. Formally, a POMDP is an 8-tuple $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, T, Z, R, b_0, \gamma \rangle$. Initially, the robot is in a hidden state $s_0 \in \mathcal{S}$. This uncertainty is represented by an initial belief $b_0 \in \mathcal{B}$, a probability distribution on the state space $\mathcal{S}$, where $\mathcal{B}$ is the set of all possible beliefs. At each step $t \geq 0$, the robot executes an action $a_t \in \mathcal{A}$ according to some policy π. Due to stochastic effects of executing actions, it transitions from the current state $s_t \in \mathcal{S}$ to a next state $s_{t+1} \in \mathcal{S}$ according to the

*This work was supported by the ARC Research Hub in Intelligent Robotic Systems for Real-Time Asset Management (IH210100030), in collaboration with the University of Sydney and Nexxis Technology.

The authors are with the School of Computing, Australian National University, Australia {marcus.hoerger, hanna.kurniawati, muhammad.sudrajat}@anu.edu.au

transition model $T(s_t, a_t, s_{t+1}) = p(s_{t+1} \mid s_t, a_t)$, which is a conditional probability function. The robot does not know the state s_{t+1} exactly, but perceives an observation $o_t \in \mathcal{O}$ from the environment according to the observation model $Z(s_{t+1}, a_t, o_t) = p(o_t \mid s_{t+1}, a_t)$. In addition, the robot receives an immediate reward $r_t = R(s_t, a_t) \in \mathbb{R}$. The goal is to find a policy π that maximizes the expected total discounted reward or the *policy value*

$$\mathcal{V}_\pi(b_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, b_0, \pi\right], \tag{1}$$

where the discount factor $0 < \gamma < 1$ ensures that $\mathcal{V}_\pi(b)$ is finite and well-defined.

The robot's decision space is the set Π of policies, defined as mappings from beliefs to actions. In this paper, we consider *stochastic policies*, i.e., π is a belief-dependent distribution over the action space. The POMDP solution is then the optimal policy, denoted as π^* and given by

$$\pi^* = \arg\max_{\pi \in \Pi} \mathcal{V}_\pi(b), \tag{2}$$

for each belief $b \in \mathcal{B}$. A more elaborate explanation is available in [1].

B. Parallel Planning under Uncertainty

Advances in modern compute hardware, including multi-core CPUs and GPUs, have given rise to new approaches in speeding up online planning under uncertainty via parallelization. Most approaches focus on solving MDPs – the fully observable variant of POMDPs – and are based on Monte Carlo Tree Search (MCTS). The works in [13], [14] focus on different MCTS parallelization approaches, including leaf, root, and tree parallelization. Leaf parallelization evaluates leaf nodes using parallel simulations, while root parallelization builds multiple MCTS trees in parallel and merges them at the end of the search. Tree parallelization performs parallel searches in a single tree, but requires heavy use of mutexes to synchronize tree data access from parallel processes.

In the context of POMDP solving, several parallel offline solvers have been proposed. The work in [8] proposed an offline solver based on Point-Based Value Iteration (PBVI) [15] implemented on GPUs to accelerate the backup step by exploiting sparsity in belief vectors and optimizing memory access. The work in [9] introduces offline solvers based on Monte Carlo Value Iteration (MCVI) [16] that exploit GPU-only and hybrid CPU–GPU architectures to parallelize action evaluation, belief node value estimation, and expected return computation.

More recently, parallel online solvers have been developed. These solvers aim to parallelize existing tree search-based methods. The work in [10] proposes a parallelized version of POMCP [17], using root parallelization (multiple search trees built in parallel) and pursuing tree parallelization, to speed up the action selection in large POMDPs. The online solver HyP-DESPOT [11] proposes a CPU–GPU hybrid architecture to parallelize the belief tree search on the CPU and Monte Carlo simulations on the GPU, combining them in a hybrid architecture.

Although these online solvers demonstrate remarkable speed-ups compared to their serial counterparts, they require careful synchronization between parallel simulations to ensure consistent updates of belief-tree statistics such as visitation counts and action value estimates. This limits their scalability, as excessive synchronization overhead can quickly offset the benefits of parallelism. Moreover, to ensure that parallel simulations explore the belief tree sufficiently, they rely on auxiliary mechanisms such as virtual losses [11], which complicates their implementation and may bias the search.

In contrast, our VOPP is a fully vectorized online solver running entirely on the GPU. It requires neither synchronization between parallel computations nor any data exchange between CPU and GPU processes. This significantly simplifies the architecture of VOPP and allows us to fully exploit the massive data parallel throughput of modern GPUs.

III. Vectorized Online POMDP Planner

In this Section, we present our fully vectorized solver Vectorized Online POMDP Planner (VOPP). In the context of our solver, vectorization refers to the reformulation of all computational steps as batched operations on tensors, a practice that aligns with the Single Instruction, Multiple Data (SIMD) paradigm of GPUs. For completeness, we first provide a brief overview of PORPP in Section III-A, followed by the description of VOPP in Sections III-B to III-E.

A. PORPP

Partially Observable Reference Policy Programming (PORPP) [12] is a recently proposed online POMDP solver. It builds on the concept of *Reference-Based* POMDPs [18], a reformulation of a POMDP whose *analytical* objective is the POMDP value function, penalized by the Kullback-Leibler (KL) divergence between the maximizing policy and a user-defined *reference policy* π_0. This objective can be compactly written as

$$\mathcal{V}(b) = \frac{1}{\eta} \log\Big[\sum_{a \in \mathcal{A}} \exp[\eta \Psi(b, a)]\Big] := [\mathcal{L}_\eta \Psi](b), \tag{3}$$

where $\eta > 0$ is a temperature parameter, balancing reward maximization with deviation from the reference policy, and $\mathcal{L}_\eta$ is the log-sum-exp operator [19]. The expression Ψ denotes *preferences* over belief-action pairs:

$$\begin{aligned}\Psi(b, a) = \frac{1}{\eta} \log(\pi_0(a \mid b)) + \mathcal{R}(b, a) \\ + \gamma \sum_{o \in \mathcal{O}} p(o \mid b, a)[\mathcal{L}_\eta \Psi](\tau(b, a, o)),\end{aligned} \tag{4}$$

where $\mathcal{R}(b, a)$ is a Monte Carlo estimate of $\int_{s \in \mathcal{S}} R(s, a) b(s) \mathrm{d}s$ and τ is the belief update operator.

For many POMDP problems, it is possible to design a reference policy π_0 that encodes domain knowledge. For instance, for motion planning under uncertainty problems,

the approach in [20] samples (macro)-actions from the reference policy using a fast deterministic sampling-based motion planner [21]. For problems where such domain knowledge is not available, π_0 can be set to be a uniform distribution over the action space, which corresponds to uniformly initialized action preferences. In our experiments in Section IV, we use uniform initial reference policies.

To correct any misspecifications of π_0, PORPP proposes an iterative scheme which gradually deforms π_0 towards an optimal policy π^* for the original POMDP. This is done by iteratively updating the preferences in eq. (4) via

$$\Psi_{k+1}(b,a) = \Psi_k(b,a) - [\mathcal{L}_\eta \Psi_k](b) + \mathcal{R}(b,a) + \gamma \sum_{o \in \mathcal{O}} p(o \mid b,a)[\mathcal{L}_\eta \Psi_k](\tau(b,a,o)), \tag{5}$$

where the policy at iteration k is derived from the preference values via the softmax function

$$\pi_k(b,a) = \frac{\exp[\eta \Psi_k(b,a)]}{\sum_{a' \in \mathcal{A}} \exp[\eta \Psi_k(b,a')]}. \tag{6}$$

In practice, PORPP interleaves belief space sampling with preference backups to approximate the action preference updates in eq. (5). The sampled beliefs are maintained in a belief tree $\mathcal{T}$, consisting of belief and action nodes. Each belief node branches into action nodes, while each action node branches into successor belief nodes based on sampled observations. PORPP constructs $\mathcal{T}$ by incrementing the following steps:

1) Forward search: Starting from the current belief b_0, PORPP samples an *episode*, i.e., a sequence of state–action–observation–reward quadruples up to a maximum depth, and adds new belief nodes along the episode's action–observation history if they do not exist yet. At each visited belief $b \in \mathcal{B}$, PORPP samples an action from the current reference policy, eq. (6), associated with the belief.
2) Preference backup: After sampling an episode, PORPP traverses the sequence of visited beliefs back to the root and, at each visited belief b, updates the preferences of the sampled actions according to eq. (5).

PORPP repeats these steps until the planning budget for the current planning step has been exceeded.

Many online POMDP solvers select actions according to some variant of UCT [22] during the forward search, which requires maximization over action values at each belief node. In contrast, PORPP selects actions by sampling from the current reference policy, which is embarrassingly parallel. Moreover, PORPP computes belief values analytically according to eq. (3). Both operations – embarrassingly parallel action *sampling* and *analytical* belief value computation – open up new avenues for efficient parallelization, which we exploit with VOPP.

B. VOPP Overview

VOPP is an anytime parallel online POMDP solver based on PORPP. Key to VOPP is the representation of all data structures associated with the belief tree $\mathcal{T}$ as tensors.

Algorithm 1 VOPP

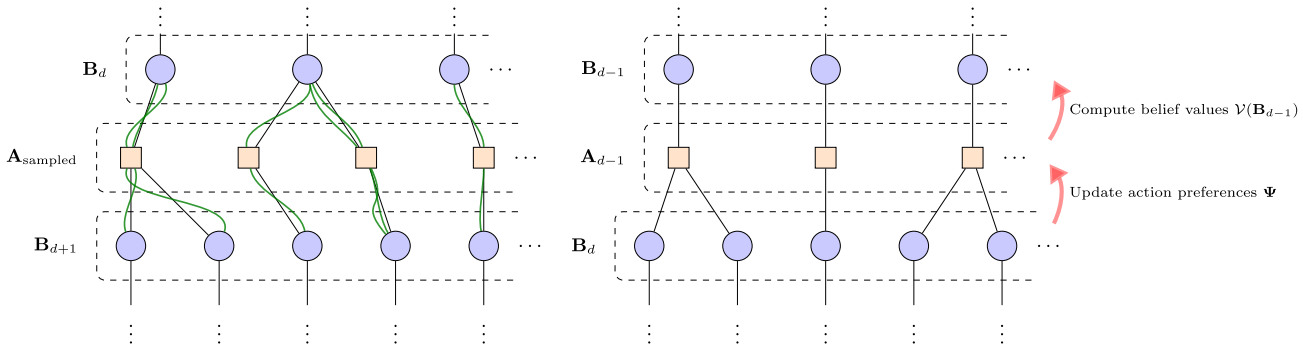
Require: Initial belief b, Num. parallel simulations n_p, Temperature η

1: **while** Problem not terminated **do**
2: $\quad\mathcal{T} \leftarrow$ Initialize tensors $\mathbf{B}, \mathbf{A}, \mathbf{\Psi}$
3: $\quad D_{\max} \leftarrow 1$
4: $\quad$**while** planning budget not exceeded **do**
5: $\quad\quad$*depth* $\leftarrow 0$
6: $\quad\quad\mathbf{S} \leftarrow$ SAMPLESTATESFROMBELIEF(b, n_p)
7: $\quad\quad\mathbf{B}_{\text{curr}} \leftarrow$ Root node indices of size $|\mathbf{S}|$
8: $\quad\quad(\mathbf{B}_{\text{leaf}}, \mathbf{H}) \leftarrow$ SEARCH$(\mathcal{T}, \mathbf{B}_0, \mathbf{S}, \textit{depth}, D_{\max})$ ▷ Alg. 2
9: $\quad\quad\mathcal{T} \leftarrow$ BACKUP$(\mathcal{T}, \mathbf{B}_{\text{leaf}}, \mathbf{H}, D_{\max})$ ▷ Alg. 3
10: $\quad\quad D_{\max} \leftarrow D_{\max} + 1$
11: $\quad$**end while**
12: $\quad\mathbf{B}_0 \leftarrow$ Root node in $\mathcal{T}$
13: $\quad a \leftarrow \arg\max_a \mathbf{\Psi}(\mathbf{B}_0, a)$
14: $\quad$Execute action a and perceive observation o
15: $\quad b \leftarrow \tau(b,a,o)$
16: **end while**

This allows VOPP to implement PORPP's key steps – *forward search* and *preference backup* – as a sequence of fully vectorized computations that manipulate the tensor data structures of $\mathcal{T}$. In contrast to existing parallel online POMDP solvers, this design requires no synchronization between concurrent computations, enabling VOPP to achieve a significantly higher computational throughput on massively parallel hardware, such as GPUs.

Suppose the POMDP to be solved is $\mathcal{P} = \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, T, Z, R, b_0, \gamma \rangle$. The state $\mathcal{S}$, action $\mathcal{A}$ and observation $\mathcal{O}$ spaces can be discrete, continuous, or hybrid. For continuous action/observation spaces, we represent the space with a fixed, yet representative set of sampled actions/observations with finite size, which is selected a priori. We also assume to have access to a stochastic generative model $G : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S} \times \mathcal{O} \times \mathbb{R}$ to simulate the transition, observation, and reward models. That is, for a given state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, the model G produces a next state $s' \in \mathcal{S}$, observation $o \in \mathcal{O}$ and reward $r \in \mathbb{R}$, such that (s', o) is distributed according to $p(o, s' \mid s, a) = T(s,a,s')Z(s',a,o)$, and $r = R(s,a)$. We further assume that G is implemented as a vectorized model, i.e., for a tensor of states and actions, it produces a tensor of next states, observations, and rewards. In the remainder of this paper, we use **BOLD** upper-case letters to denote tensors.

The key steps of VOPP are presented in Algorithm 1. At each planning loop iteration, (lines 4 to 11), VOPP performs a vectorized forward search to expand the belief tree by one level, followed by a vectorized backup operation (line 9) to update the action preferences stored in $\mathcal{T}$. Details on the vectorized forward search and backup operations are provided in Section III-D and Section III-E, respectively. Figure 1 provides an illustration of both steps. These steps are repeated until the planning budget for the current plan-



(a) Vectorized forward search (b) Vectorized preference backup

Fig. 1. Illustration of the two vectorized main operations – *forward search* (a) and *preference backup* (b) – of VOPP. Blue circles represent belief nodes, while yellow squares represent action nodes. The green lines represent sampled episodes. (a) *Vectorized forward search*: VOPP samples an action for each episode from the belief nodes $\mathbf{B}_d$ at depth d in parallel and collects the sampled actions in the action tensor $\mathbf{A}_{\text{sampled}}$. It then performs a vectorized forward simulation of the episodes from one step using the generative model G and $\mathbf{A}_{\text{sampled}}$. For the resulting observations, VOPP appends new belief nodes to $\mathbf{B}$ if they do not exist yet. The search then continues from the belief nodes at depth $d+1$ that the episodes visit. (b) *Vectorized preference backup*: For all belief nodes $\mathbf{B}_d$ at depth d, VOPP updates the preference values $\mathbf{\Psi}$ of their parent actions in single vectorized step. The updated preference values are then used to compute the belief values of all beliefs $\mathbf{B}_{d-1}$ at depth $d-1$ in one vectorized step, before the backup continues from $d-1$.

ning step has been exceeded. Finally, we select the action with the highest preference value at the root node (line 13), execute the action in the environment (line 14) and update the belief, based on the action executed and observation perceived (line 15). To update the belief, we use a Sequential Importance Resampling (SIR) particle filter [23]. The next section details our belief tree tensor data structures.

C. Belief Tree Tensor Data Structure

To enable a fully vectorized implementation of VOPP, we represent all internal data structures associated with the belief tree $\mathcal{T}$ as a collection of three tensors, $\mathbf{B}$, $\mathbf{A}$, and $\mathbf{\Psi}$. The tensors $\mathbf{B}$ and $\mathbf{A}$ represent the belief and action nodes, including their parent-child relations. Specifically, $\mathbf{B}$ is a 2D tensor, where each row corresponds to a belief node and contains two entries: the first stores the row index of the parent action node in $\mathbf{A}$, while the second stores the parent observation. For the root node, stored in the first row of $\mathbf{B}$, both entries are set to `NULL`. The tensor $\mathbf{A}$ is a 2D tensor in which each row represents an action node and contains four entries: The first entry stores the row index in $\mathbf{B}$ of the node's parent belief, while the second stores the action associated with the node. The final two entries store the action node's cumulative immediate reward and visitation count, respectively, which are updated during the forward search. Finally, $\mathbf{\Psi}$ is a 2D tensor with $1+|\mathcal{A}|$ columns. Each row stores the current action preference values (defined in eq. (4)) for a specific belief. The first column stores the row index in $\mathbf{B}$ corresponding to that belief, while the remaining $|\mathcal{A}|$ columns store the preference value for each action.

Together, these tensors form a compact representation of the belief tree, $\mathcal{T} = \{\mathbf{B}, \mathbf{A}, \mathbf{\Psi}\}$, which enables fully vectorized forward search and backup operations, as detailed in the next two sections.

D. Vectorized Forward Search

Algorithm 2 presents the pseudocode for VOPP's vectorized forward search. At each planning loop iteration, we first sample a batch of size n_p of initial states from the current belief and store them in a state tensor $\mathbf{S}$ (line 6 in Algorithm 1). The parameter n_p determines the number of episodes that are sampled in parallel during the vectorized forward search (in our experiments, we use up to $60,000$ parallel episodes). We also construct a matching index tensor $\mathbf{B}_{\text{curr}}$ of the same batch size (line 7), where each entry points to the root node in $\mathbf{B}$, thereby associating each sampled state in $\mathbf{S}$ with the initial belief node in the tree. We then recursively sample a batch of episodes, i.e., sequences of state–action–observation–reward quadruples, starting from the initial states in $\mathbf{S}$, as follows:

Algorithm 2 SEARCH$(\mathcal{T}, \mathbf{B}_{\text{curr}}, \mathbf{S}, \mathit{depth}, D_{\max})$

1: **if** $\mathit{depth} > D_{\max}$ **then**
2: **return** $(\mathbf{B}_{\text{curr}}, \text{VALUEHEURISTIC}(\mathbf{S}))$
3: **end if**
4: $\boldsymbol{\pi}(\mathbf{B}_{\text{curr}}, \cdot) \leftarrow \text{SOFTMAX}[\eta \cdot \mathbf{\Psi}(\mathbf{B}_{\text{curr}}, \cdot)]$
5: $\mathbf{A}_{\text{sampled}} \sim \boldsymbol{\pi}(\mathbf{B}_{\text{curr}}, \cdot)$
6: $(\mathbf{S}', \mathbf{O}, \mathbf{R}) \leftarrow G(\mathbf{S}, \mathbf{A}_{\text{sampled}})$ ▷ Generative model
7: $(\mathbf{A}_{\text{node}}, \mathcal{T}) \leftarrow \text{APPENDACTIONS}(\mathcal{T}, \mathbf{B}_{\text{curr}}, \mathbf{A}_{\text{sampled}})$
8: $(\mathbf{B}_{\text{next}}, \mathcal{T}) \leftarrow \text{APPENDBELIEFS}(\mathcal{T}, \mathbf{A}_{\text{node}}, \mathbf{O})$
9: **return** SEARCH$(\mathcal{T}, \mathbf{B}_{\text{next}}, \mathbf{S}', \mathit{depth}+1, D_{\max})$

For each belief indexed by $\mathbf{B}_{\text{curr}}$, we first construct a softmax policy π (line 4) using the corresponding action preferences in $\mathbf{\Psi}(\mathbf{B}_{\text{curr}}, \cdot)$ according to eq. (6). We then sample an action for each belief from this newly constructed policy and store the results in the action tensor $\mathbf{A}_{\text{sampled}}$ (line 5). Note that both the policy construction and action sampling steps are fully vectorized over all entries in $\mathbf{B}_{\text{curr}}$. The batch of states in $\mathbf{S}$ and corresponding actions in $\mathbf{A}_{\text{sampled}}$ are then simulated forward in a single vectorized step using the generative model G, yielding the next state tensor $\mathbf{S}'$, the observation tensor $\mathbf{O}$, and the reward tensor $\mathbf{R}$ (line 6).

Following the forward simulation step, the belief tree $\mathcal{T}$ is expanded with the sampled actions and observations (lines 7 to 8). To ensure that no action or belief nodes are added more than once, we use the following vectorized expansion

process: First, we pair each belief index in $\mathbf{B}_{\text{curr}}$ with the corresponding action in $\mathbf{A}_{\text{sampled}}$ to form a batch of belief-action pairs and identify all unique pairs. Using a fast hash-based matching algorithm, we efficiently determine which action nodes corresponding to these unique pairs already exist in $\mathbf{A}$. New, previously unseen pairs are concatenated to the action tensor $\mathbf{A}$, and a single index tensor $\mathbf{A}_{\text{node}}$ is returned. This tensor provides the corresponding action node index in $\mathbf{A}$ (either existing or newly created) for each entry in the sampled action tensor $\mathbf{A}_{\text{sampled}}$. During this process, the cumulative rewards and visit counts stored in $\mathbf{A}$ are updated for all affected action nodes in a single vectorized step.

To append new belief nodes to $\mathbf{B}$, we use a similar vectorized procedure. We pair each action node index in $\mathbf{A}_{\text{node}}$ with the corresponding observation in $\mathbf{O}$ and identify all unique parent-observation pairs. These pairs are then matched against existing belief nodes in $\mathbf{B}$, with new nodes being created as needed. This process returns an index tensor, $\mathbf{B}_{\text{next}}$, that points to the belief nodes for the subsequent simulation step (line 9).

This recursive forward search continues until depth $D_{\max}$. For the resulting belief nodes at depth $D_{\max}$, a state-based, problem-dependent heuristic function estimates their values from $\mathbf{S}$ (line 2). These estimates serve as the initial values for the subsequent backup phase.

E. Vectorized Preference Backup

Algorithm 3 BACKUP($\mathcal{T}, \mathbf{B}_{\text{leaf}}, \mathbf{H}, D_{\max}$)

1: $\mathbf{N}(\mathbf{B}_{\text{leaf}}) \leftarrow$ AGGREGATEBELIEFVISITS($\mathbf{B}_{\text{leaf}}$)
2: $\mathbf{C}(\mathbf{B}_{\text{leaf}}) \leftarrow$ AGGREGATEHEURISTICS($\mathbf{B}_{\text{leaf}}, \mathbf{H}$)
3: $\mathcal{V}(\mathbf{B}_{\text{leaf}}) \leftarrow \mathbf{C}(\mathbf{B}_{\text{leaf}})/\mathbf{N}(\mathbf{B}_{\text{leaf}})$
4: **for** $d = D_{\max}, D_{\max}-1, \ldots, 1$ **do**
5: Let $\mathbf{B}_d$ be the tensor of belief nodes at depth d in $\mathcal{T}$
6: Let $\mathbf{A}_{d-1}$ be the tensor of parent actions of $\mathbf{B}_d$ in $\mathcal{T}$
7: Let $\mathbf{B}_{d-1}$ be the tensor of parent beliefs of $\mathbf{A}_{d-1}$ in $\mathcal{T}$
8: $\mathbf{R}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1}) \leftarrow \frac{\mathbf{C}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1})}{\mathbf{N}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1})}$
9: $\mathbf{W}(\mathbf{A}_{d-1}) \leftarrow$ WEIGHTEDSUM($\mathcal{V}(\mathbf{B}_d), N(\mathbf{B}_d)$)
10: $\mathbf{Q}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1}) \leftarrow \mathbf{R}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1}) + \gamma \mathbf{W}(\mathbf{A}_{d-1})$
11: $\mathbf{N}(\mathbf{B}_{d-1}) \leftarrow$ SUMACTIONVISITS($\mathbf{N}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1})$)
12: $\mathcal{V}_{\text{curr}}(\mathbf{B}_{d-1}) \leftarrow [\mathcal{L}_\eta \mathbf{\Psi}](\mathbf{B}_{d-1})$
13: $\mathbf{\Psi}(\mathbf{B}_{d-1}, \cdot) \leftarrow \mathbf{\Psi}(\mathbf{B}_{d-1}, \cdot) - \mathcal{V}_{\text{curr}}(\mathbf{B}_{d-1}) + \mathbf{Q}(\mathbf{B}_{d-1}, \mathbf{A}_{d-1})$
14: $\mathcal{V}(\mathbf{B}_{d-1}) \leftarrow [\mathcal{L}_\eta \mathbf{\Psi}](\mathbf{B}_{d-1})$
15: **end for**

After sampling a batch of episodes as described in the previous section, we perform a sequence of vectorized backup operations to update the action preferences values at the sampled beliefs, as detailed in Algorithm 3.

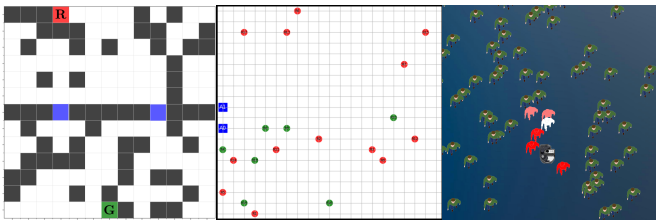
The backup process begins at the leaf nodes reached by the sampled episodes. We first perform vectorized aggregation operations to initialize their values based on the heuristic estimates in $\mathbf{H}$. The function AGGREGATEBELIEFVISITS (line 1 in Algorithm 3) performs a batched count of each unique belief node index in the leaf node tensor $\mathbf{B}_{\text{leaf}}$ to determine their visit counts. Similarly, AGGREGATEHEURISTICS (line 2) performs a batched sum of the heuristic values $\mathbf{H}$ for each of these unique leaf nodes. The final value estimate $\mathcal{V}(b)$ for each leaf is then computed by dividing the summed heuristic value by the visit count (line 3).

Subsequently, the backup proceeds iteratively from the leaf nodes $d = D_{\max}$ to the root. In each iteration, we perform a series of vectorized computations on *all* nodes at a given depth to update the corresponding action preferences $\mathbf{\Psi}$ according to eq. (5).

First, we compute a tensor of Q values for all action nodes $\mathbf{A}_{d-1}$ (lines 8 to 10). These Q values combine the average immediate reward $\mathbf{R}$, computed from the cumulative rewards $\mathbf{C}(\mathbf{B}_{d-1}, \cdot)$ and the visit counts $\mathbf{N}(\mathbf{B}_{d-1}, \cdot)$ stored in the global action tensor $\mathbf{A}$ (line 8), with the expected future value $\mathbf{W}$. This future value is computed by the WEIGHTEDSUM function (line 9) as the weighted average of the values of all child belief nodes in $\mathbf{B}_d$, where each child's value is weighted by its relative visit count.

With the Q values computed for all actions at depth $d-1$, we update their action preference values and the values of their parent belief nodes in $\mathbf{B}_{d-1}$. The visit counts for these parent nodes are computed by aggregating their corresponding child action visits (line 11). Next, the action preferences $\mathbf{\Psi}$ are updated using the newly computed Q values. To do this, we compute the current values $\mathcal{V}_{\text{curr}}$ of the beliefs in $\mathbf{B}_{d-1}$ using the existing preference values and the log-sum-exp operator (line 12). With the current belief values and the computed Q values, we update the action preferences $\mathbf{\Psi}$ (line 13). Finally, the value $\mathcal{V}(b)$ for each belief node in $\mathbf{B}_{d-1}$ is recalculated from these updated preferences (line 14), before the backup progresses with the next iteration.

IV. EXPERIMENTS AND RESULTS



(a) Navigation (b) MARS (c) CrowdNav

Fig. 2. The problem scenarios used to evaluate VOPP.

We tested VOPP on three planning under uncertainty benchmark problems, detailed below.

A. Experimental Scenarios

Navigation in a partially known map (Navigation) [11]: In this problem, shown in Figure 2(a), a robot (red square) starts from a random position at the top border of a map with 13×13 cells, consisting of randomly placed obstacles (black squares), and must reach a goal area (green square) at the bottom of the map by passing one of the two gates

in the middle wall (blue squares), while avoiding collisions with the obstacles. The obstacle locations and which of the two gates is open are only partially known to the robot, but it has access to a noisy sensor that provides information on which of the eight neighboring grid cells around the robot are occupied by an obstacle (resulting in $|\mathcal{O}| = 8$). In each step, the robot can move to one of its eight neighboring grid cells (resulting in $|\mathcal{A}| = 8$) or remain in its current cell. Reaching the goal yields a reward of 20, colliding with an obstacle and standing still yield a penalty of -1 and -0.2, respectively. Additionally, for every step, the robot receives a small penalty of -0.1. The problem terminates when the robot has reached the goal cell or a after a maximum of 60 planning steps. The discount factor is $\gamma = 0.983$. This problem has a very large state space of size $|\mathcal{S}| = 169 \times 2^{124}$, which includes the robot position and the occupancy of unknown cells.

Multi-Agent Rocksample (MARS) [11]: MARS(n, m), shown in Figure 2(b) is an extension of the popular Rocksample benchmark problem, in which two agents (blue squares) operate in a $n \times n$ map populated by m randomly placed rocks that are either `GOOD` (green rocks) or `BAD` (red rocks), resulting in $|\mathcal{S}| = n^4 \times 2^m$. The agents do not know the rock states initially, but they are equipped with a noisy sensor to detect the state of a rock ($|\mathcal{O}| = 9$, including a `NULL` observation, when the sensor is not used). If an agent is on top of a rock, it can `SAMPLE` it (resuling in an action space of size $|\mathcal{A}| = (5 + m)^2$), which yields a reward of 10 for good rocks and -10 for bad rocks. Good rocks turn bad after sampling. The agents must work cooperatively to sample as many good rocks as possible, before leaving the map on the right-hand side, which yields a reward of 10. The problem terminates when both agents leave the map or a maximum of 90 planning steps has been reached. The discount factor in this problem is $\gamma = 0.983$.

Crowd Navigation (CrowdNav): We propose CrowdNav (Figure 2(c)), a problem in which a Stretch 3 mobile robot navigates in a conference hall of size 50×40m densely populated by 300 randomly placed people. While the robot can fully observe the location of the people, their behavior is determined by an initially unknown character trait: each person is either curious with probability p_{curious} or shy with probability $1 - p_{\text{curious}}$. The state space consists of the robot's 2D position and the 2D positions and character traits of the n-nearest people to the robot (resulting in the state space $\mathcal{S} = \mathbb{R}^2 \times \mathbb{R}^{2n} \times \{\texttt{CURIOUS}, \texttt{SHY}\}^n$). The motion of the people is stochastic. At each time step, the movement of all people is perturbed by zero-mean Gaussian noise with a standard deviation of $\sigma_p = 0.05$. Additionally, with a probability of 0.9, people within a radius r_{nearby} of the robot also react based on their trait: curious ones move toward the robot with velocity v_{curious}, while shy ones move away with velocity v_{shy}. To navigate, the robot can choose from four directional actions—`NORTH`, `EAST`, `SOUTH`, `WEST`—each moving it one meter. It also has a `YELL` action (resulting in $|\mathcal{A}| = 5$), which causes all nearby people to rapidly back away with velocity v_{back}. The robot's observation at each step encodes whether each of the n-nearest people decreased or increased their distance from its position (resulting in $|\mathcal{O}| = 2^n$). Since the crowd behavior is stochastic, this observation provides only an imperfect signal of their hidden traits. The robot's objective is to travel from the southern to the northern border of the hall, receiving a reward of $1,000$ for success. Bumping into a person incurs a penalty of -200. Since using the `YELL` action might disturb nearby people, it incurs a penalty of -25. Additionally, each step incurs a small step penalty of -1. The problem terminates once the robot leaves the hall, or after maximum of 200 planning steps. In our experiments, we set $r_{\text{nearby}} = 4m$, $v_{\text{curious}} = 0.3m/s$, $v_{\text{shy}} = 0.8m/s$, $v_{\text{back}} = 2m/s$, and $n = 6$. The discount factor is $\gamma = 0.97$.

B. Experimental Setup

The purpose of our experiments is two-fold: The first is to compare VOPP with a state-of-the-art parallel online POMDP solver HyP-DESPOT [11] in the Navigation and MARS problem scenarios. To do this, we implemented VOPP and the problem scenarios in Python. We use PyTorch [24] as the backbone of VOPP's tensor data structures and vectorized computations due to its maturity, simplicity, and rich API, though other libraries such as JAX [25] or Taichi [26] are possible, too. For VOPP, we first ran a set of systematic trials for both problem scenarios to determine the best parameters, that is, the temperature parameter η in eq. (3) and the number n_p of episodes that are sampled in parallel during our vectorized forward search (Section III-D). Based on these trials, we set $\eta = 2.0$ for both Navigation and MARS, and $n_p = 50,000$ and $n_p = 60,000$ for Navigation and MARS respectively. For HyP-DESPOT, we use the implementation[1] and the parameters for both problem scenarios provided by the authors [11]. The results of these experiments are presented in Section IV-C.

The second purpose is to demonstrate the efficacy of VOPP in a challenging robotics planning under uncertainty problem. In particular, we tested VOPP on the CrowdNav problem, where we investigated VOPP's robustness to different crowd behaviors. Specifically, we varied the curiosity probability, p_{curious}, across five scenarios, where we used $p_{\text{curious}} \in \{0, 0.25, 0.5, 0.75, 1\}$. We then analyzed the robot trajectories computed by VOPP in terms of length and safety. The results are presented in Section IV-D.

All experiments were carried out on a laptop with one Intel Core i7-13850HX CPU with 32GB of RAM and a Nvidia RTX 3500 ADA GPU with 12GB of VRAM. VOPP uses only the GPU, while HyP-DESPOT uses both the CPU and GPU.

C. Comparison with HyP-DESPOT

To compare the performance of VOPP with HyP-DESPOT, we first tested both solvers on MARS$(20, 20)$, a variant of MARS with a map of size 20, randomly populated by 20 rocks. This variant has an action space of 625 actions. We ran

[1] `https://github.com/AdaCompNUS/hyp-despot`

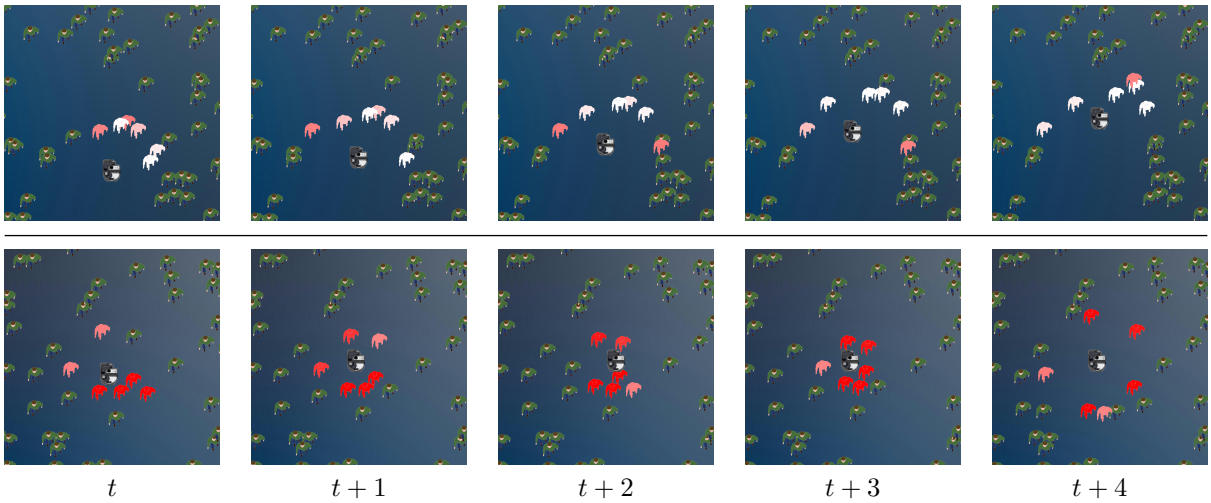


Fig. 3. Two partial trajectories of the Stretch 3 mobile robot in the CrowdNav scenario with $p_{\text{curious}} = 0.0$ (top) and $p_{curious} = 1.0$ (bottom) at different time steps. Nearby people are colored according to their inferred character trait. Darker red tones indicate a higher probability of a person being curious.

TABLE I

AVERAGE TOTAL DISCOUNTED REWARDS AND 95% CONFIDENCE INTERVALS OF VOPP AND HYP-DESPOT ON THE MARS$(20, 20)$ AND NAVIGATION PROBLEMS. THE AVERAGE IS TAKEN OVER 200 SIMULATION RUNS PER SOLVER, PROBLEM AND PLANNING TIME / STEP.

	Planning time / step (s)	Avg. total discounted reward
MARS$(20, 20)$		
HyP-DESPOT	1.0	47.9 ± 1.6
VOPP (Ours)	1.0	58.8 ± 2.1
VOPP (Ours)	0.1	53.3 ± 2.1
VOPP (Ours)	0.05	50.0 ± 1.9
VOPP (Ours)	0.01	31.1 ± 2.6
MARS$(50, 50)$		
HyP-DESPOT	–	–
VOPP (Ours)	1.0	45.1 ± 2.0
Navigation		
HyP-DESPOT	1.0	9.3 ± 1.3
VOPP (Ours)	1.0	11.9 ± 0.6
VOPP (Ours)	0.1	11.7 ± 0.7
VOPP (Ours)	0.05	10.7 ± 0.8
VOPP (Ours)	0.01	8.8 ± 1.0

both solvers for 200 simulation runs with 1 second planning time per step. Table I shows the average total discounted rewards achieved by both solvers. VOPP clearly outperforms HyP-DESPOT in this scenario by a significant margin. VOPP computes strategies for which both agents tend to sample more good rocks before leaving the map. Initially, the environment consists of 10 good rocks on average, and VOPP sampled, on average, 9 good rocks and 0.1 bad rocks. In contrast, HyP-DESPOT sampled, on average, only 6 good rocks (and 0.12 bad rocks) before the agents leave the map.

To further test VOPP's performance on the MARS$(20, 20)$ problem, we ran an additional set of experiments in which we reduced the maximum planning time per step to 0.1, 0.05 and and 0.01 seconds respectively. The results in Table I indicate that VOPP is at least $20\times$ more efficient than HyP-DESPOT in this problem: For a planning time of 0.01s per step (a hundredth of what was used for HyP-DESPOT), VOPP already achieves an average total discounted reward of $\sim 64\%$ of what HyP-DESPOT achieves. As we increase the planning time per step to 0.1s, VOPP achieves a better result than HyP-DESPOT. In fact, the policies generated by VOPP with 0.05s planning time per step are better than what HyP-DESPOT can generate with 1s of planning time per step.

To test the scalability of VOPP further, we ran 200 simulation runs on the MARS$(50, 50)$ problem, a variant of MARS with 3025 actions. The results in Table I indicate that VOPP handles this problem well, achieving an average total discounted reward of ~ 45.1. On average, the environment contains 25 good rocks. VOPP sampled an average of 21 good rocks and 1.32 bad rocks per run. In contrast to many existing online solvers (including HyP-DESPOT), VOPP does not require an exhaustive enumeration of all actions, which makes it much more suitable for solving POMDP problems with large action spaces. Unfortunately, we were unable to test HyP-DESPOT on MARS$(50, 50)$, since the implementation provided by the authors crashed for variants larger than MARS$(36, 36)$.

For the Navigation problem, we tested both VOPP and HyP-DESPOT using 200 simulation runs with a maximum planning time of 1s per planning step. The results are shown in Table I. Similarly to MARS, VOPP significantly outperforms HyP-DESPOT in this problem.

D. Navigation in a crowd

We tested VOPP on the CrowdNav problem using 50 simulation runs for each curiosity probability $p_{\text{curious}} \in \{0, 0.25, 0.5, 0.75, 1\}$ with a maximum planning time of 1s per step.

Table II shows the average path length, the average number of times the robot bumped into people, and the average number of times the robot used the `YELL` action for each value of p_{curious}. In all simulation runs, the robot reached its goal at the northern border of the hall. With increasing p_{curious}, the crowd consists of more curious people, causing the robot to take detours on its way to the goal in order to avoid bumping into the people. This is reflected in the

TABLE II

THIS TABLE SHOWS THE AVERAGE PATH LENGTH, NUMBER OF TIMES THE ROBOT BUMPED INTO PEOPLE AND NUMBER OF TIMES THE ROBOT USED THE YELL ACTION AND 95% CONFIDENCE INTERVALS IN THE CROWDNAV SCENARIO FOR DIFFERENT VALUES OF p_{CURIOUS}. THE AVERAGE IS TAKEN OVER 50 SIMULATION RUNS FOR EACH p_{CURIOUS}.

Curiosity prob. p_{curious}	0.0	0.25	0.5	0.75	1.0
Avg. path length	77.0 ± 2.7	75 ± 3.1	90.7 ± 5.6	108 ± 6.5	123 ± 7.8
Avg. num. bumps	0.0 ± 0.0	0.2 ± 0.1	0.4 ± 0.1	0.5 ± 0.2	0.8 ± 0.2
Avg. num. yells	2.6 ± 0.9	2.4 ± 0.5	5.0 ± 0.8	8.1 ± 1.0	12.3 ± 1.4

higher average path length and the only marginally increasing average number of times the robot bumps into people.

More interestingly, different crowd behaviors (in terms of the curiosity probability p_{curious}) exhibit vastly different strategies employed by the robot, based on the inferred character traits of nearby people. For smaller p_{curious}, the crowd consists mostly of shy people, i.e., they tend to avoid the robot. Over time, the robot infers this character trait and adapts its strategy accordingly by taking more direct actions towards the goal. An example is shown in Figure 3 (top row), where the people around the robot are inferred to be shy with high probability. As a result, the robot dashes towards the goal, even if the direct path is currently blocked. On the other hand, for larger values of p_{curious}, the crowd consists of more curious people. This causes the robot to take detours in order to avoid bumping into people (as reflected by the increased time to reach the goal). In addition, the robot is often surrounded by curious people. In such situations, the robot tends to use the YELL action as a strategy to avoid bumping into people, which causes nearby people to temporarily retreat from the robot. An example of this behavior is shown in Figure 3 (bottom row). The nearby people are inferred to be curious with high probability and surround the robot at time step $t + 3$. In the same time step, the robot uses the YELL action, helping it avoid bumping into people and creating more space for maneuvering.

V. CONCLUSION

We propose a novel parallel online POMDP solver, called VOPP, the first fully vectorized online solver. VOPP builds on a recent POMDP formulation that analytically solves value functions and only leaves the estimation of expectations for numerical computations, thereby removing any requirements for explicit synchronization between parallel computations. VOPP represents all data structures related to the belief tree as tensors and formulates planning as a sequence of fully vectorized operations over this representation, with no dependencies between parallel computations. This allows VOPP to fully leverage the massive data parallel throughput of modern GPUs. Experimental results indicate that VOPP is at least $20\times$ more efficient than a current state-of-the-art parallel online POMDP solver.

REFERENCES

[1] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[2] C. H. Papadimitriou and J. N. Tsitsiklis, "The complexity of Markov decision processes," *Mathematics of Operations Research*, vol. 12, no. 3, pp. 441–450, 1987.

[3] H. Kurniawati, "Partially Observable Markov Decision Processes and Robotics," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 5, pp. 253–277, 2022.

[4] M. Hoerger, J. Song, H. Kurniawati, and A. Elfes, "POMDP-based Candy Server: Lessons Learned from a Seven Day Demo," in *ICAPS*, 2019, pp. 698–706.

[5] H. Bai and D. Hsu, "Unmanned aircraft collision avoidance using continuous-state pomdps," *Robotics: Science and Systems VII*, vol. 1, pp. 1–8, 2012.

[6] M. S. Saleem, R. Veerapaneni, and M. Likhachev, "A pomdp-based hierarchical planning framework for manipulation under pose uncertainty," *arXiv preprint arXiv:2409.18775*, 2024.

[7] M. Lauri, D. Hsu, and J. Pajarinen, "Partially Observable Markov Decision Processes in Robotics: A survey," *IEEE Trans. on Robotics*, vol. 39, no. 1, pp. 21–40, 2022.

[8] K. H. Wray and S. Zilberstein, "A parallel point-based pomdp algorithm leveraging gpus." in *AAAI Fall Symposia*, 2015, pp. 95–96.

[9] T. Lee and Y. J. Kim, "Massively parallel motion planning algorithms under uncertainty using pomdp," *The International Journal of Robotics Research*, vol. 35, no. 8, pp. 928–942, 2016.

[10] S. Basu, S. Rajesh, K. Zheng, S. Tellex, and R. I. Bahar, "Parallelizing pomcp to solve complex pomdps," in *RSS workshop on software tools for real-time optimal control*, 2021.

[11] P. Cai, Y. Luo, D. Hsu, and W. S. Lee, "Hyp-despot: A hybrid parallel algorithm for online planning under uncertainty," *The International Journal of Robotics Research*, vol. 40, no. 2-3, pp. 558–573, 2021. [Online]. Available: https://doi.org/10.1177/0278364920937074

[12] E. Kim and H. Kurniawati, "Partially Observable Reference Policy Programming: Approximately Solving POMDPs Sans Numerical Optimisation," in *IJCAI*, 2025.

[13] T. Cazenave and N. Jouandeau, "On the parallelization of uct," in *Computer games workshop*, 2007.

[14] G. M.-B. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel monte-carlo tree search," in *International Conference on Computers and Games*. Springer, 2008, pp. 60–71.

[15] J. Pineau, G. J. Gordon, and S. Thrun, "Point-based value iteration: An anytime algorithm for POMDPs," in *IJCAI*, G. Gottlob and T. Walsh, Eds. Acapulco, Mexico: Morgan Kaufmann, 2003, pp. 1025–1032.

[16] H. Bai, D. Hsu, W. S. Lee, and V. A. Ngo, "Monte carlo value iteration for continuous-state pomdps," in *algorithmic foundations of robotics IX: selected contributions of the ninth international workshop on the algorithmic foundations of robotics*. Springer, 2010, pp. 175–191.

[17] D. Silver and J. Veness, "Monte Carlo planning in large POMDPs," in *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, ser. NIPS'10, vol. 2. Red Hook, New York: Curran Associates Inc., 2010, p. 2164–2172.

[18] E. Kim, Y. Karunanayake, and H. Kurniawati, "Reference-based pomdps," *Advances in Neural Information Processing Systems*, vol. 36, pp. 40 659–40 675, 2023.

[19] P. Blanchard, D. J. Higham, and N. J. Higham, "Accurately computing the log-sum-exp and softmax functions," *IMA Journal of Numerical Analysis*, vol. 41, no. 4, pp. 2311–2330, 2021.

[20] Y. Liang, E. Kim, W. Thomason, Z. Kingston, H. Kurniawati, and L. E. Kavraki, "Scaling long-horizon online pomdp planning via rapid state space sampling," *arXiv preprint arXiv:2411.07032*, 2024.

[21] W. Thomason, Z. Kingston, and L. E. Kavraki, "Motions in microseconds via vectorized sampling-based planning," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2024, pp. 8749–8756.

[22] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[23] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking,"

IEEE Transactions on Signal Processing, vol. 50, no. 2, pp. 174–188, 2002.

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[25] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, "Jax: composable transformations of python+numpy programs," *GitHub repository*, 2018. [Online]. Available: http://github.com/google/jax

[26] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: a language for high-performance computation on spatially sparse data structures," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–16, 2019.