Parallelizing POMCP - Research comps report

Semanti Basu

Abstract— **Planning under uncertainty is an active area of research in robotics. Partially-Observable Markov Decision Process (POMDP) provides a way to model such planning problems where system states are not fully known. Partially-Observable Monte Carlo Planning (POMCP) is a widely used online planning algorithm that scales well to complex POMDPs and large domains. POMCP is based on building a search tree by iteratively running a large number of simulations to get acceptable accuracy — a process that becomes challenging and expensive over large domains.The POMCP search is based on Monte Carlo Tree Search (MCTS), where each simulation depends on the updates from the previous one.In this paper we introduce two different ways to parallelize POMCP: (1) Tree Parallel POMCP (2) Lockfree Tree parallel PO-UCT. Tree parallel POMCP extends a commonly used parallelization scheme for MCTS to partially observable POMDP planning. Lockfree Tree parallel PO-UCT is an attempt to accelerate POMCP by decoupling the search from the belief update, which lets us perform parallel search without involving locks.For the tree parallel schemes, we explore several techniques to deploy threads strategically for an effective multi-threaded search. We also explore how different parameters such as the number of threads used and the domain size of the POMDP problem being solved, affect speedup.After extensive experiments on different domains, we show that parallel POMCP can achieve a speedup of upto 5x over it's serial counterpart on a 6 core machine.**

I. INTRODUCTION

This report builds upon and extends the work presented at a Robotics Science and Systems workshop [1].

To operate in the real world, agents must be able to deal with uncertainties in the environment. Sensor readings are error-prone. There could be occlusion and incomplete information available to the agent. To handle such situations where the system state is only partially observable, the "Partially-Observable Markov Decision Process" or POMDP architecture [2] is popularly used to model robot planning problems under uncertainty. The objective of solving a POMDP is to produce an optimal policy that maximizes the expectation of discounted returns, an intractable feat for many real-world POMDPs. By solving a POMDP problem, we can obtain the optimal policy at each time step. Online planning methods can be used to solve POMDP problems. One such planning algorithm is "Partially Observable Monte Carlo Planning" or POMCP [3]. POMCP is based on Monte Carlo planning and depends on running several simulations iteratively to generate an optimal action at each step.Its performance depends on the number of simulations we can run, and improves with a larger number of simulations. POMCP has been widely successful and has proven to be robust across a variety of domains [4][5]. As such, the algorithm is inherently sequential since the trajectory of one simulation depends on the tree built and updated from all previous simulations. So, each stage of building the search tree depends on knowledge from the previous stage. The greater the number of simulations we run, the more-detailed the search tree is, and more accurate the action predicted by the tree is. However, running numerous simulations on a large domain can be computationally expensive to simulate. This makes POMCP prohibitively slow across larger domains, and prevents it from scaling.

In this paper we make the following contributions:

1) We scale up the POMCP algorithm by upto 5x on a 6 core machine by parallelization it.
2) We present several schemes for effective parallelization of the Tree Search involved in POMCP, include strategic locking as well as a lockfree approach.
3) We augment POMCP by exploring different strategies such as virtual loss, keeping track of incomplete simulations [6],and a combination of both, to spread out the CPU threads effectively during planning.

II. BACKGROUND

A. POMDP architecture

A POMDP[2] problem can be defined by seven parameters: $\langle S, A, \Omega, O, R, T, \gamma \rangle$. S represents the state, A represents the set of possible actions,Ω is the set of possible observations,O is the observation model which is essentially the probability distribution over all possible observations, R defines the reward function, T is the transition model from one state to the next, and γ is the discount factor.

History is defined as the sequence of action-observation pairs upto that point in time such that $h_t = (a_0, o_0, a_1, o_1, \cdots, a_t, o_t)$.The states are not fully observable to the agent, however the agent can receive an observation from the environment which helps it to refine its "belief state". A belief state (B(h)), in POMDP, is the probability distribution over the system states, given history, $B(h) = Pr(s_t = s|h)$.The transition model determines the next state, given current state and action, $T = Pr(s_{t+1} = s'|s_t = s, a_t = a)$.The observation model defines the probability of receiving an observation given next state and action, $O = Pr(o_{t+1} = o|s_{t+1} = s, a_t = a)$. The reward function determines the expected reward, given state and action at time l, $R = E[r_{l+1}|s_l, a_l]$.

The probability distribution over actions, given history is known as the *policy* π. $\pi_t(h, a) = P(a_{t+1} = a|h_t = h)$. The return $R_t = \sum_{l=t}^{\infty} \gamma^{l-t} r_l$ is the sum of discounted reward obtained at each time step, from time t onwards .The expected return on executing a policy is given by the value function $V^{\pi}(h) = E[R_t|h_t = h]$ .The policy that maximizes the value function gives us the action at that time step.

In robotics, POMDP plays an important role in problem formulation and has been used in several applications [7] [8] [9].In Doshi and Roy [9], dialog management on a robotic wheelchair is considered. Dialog managers based on the POMDP architecture usually assume apriori knowledge of the user model. In [9], a bayesian approach to learning the user model, while at the same time fine-tuning the dialog-manager's policy,is introduced. In [8], planning in partially observable environment is explored in the context of a grasping/robot manipulation problem. In [8], a POMDP is designed such that the states are the regions formed by partitioning the configuration space. Roy and Thrun [7] addresses the problem of minimizing uncertainty while reaching a goal state.In [7], the authors discuss how POMDPs become intractable over large state spaces and to work around that, the positional uncertainty of the mobile robot's location is embedded as a state variable in the solution they propose.

B. Monte Carlo Sampling based online planning

Monte Carlo sampling based planning methods are popularly used to solve POMDP problems. POMCP [3] and DESPOT [10] are commonly used online planning algorithms based on Monte Carlo Sampling, which are used to solve POMDP problems.POMCP simulataneously performs Monte Carlo Tree Search along with Monte carlo belief state update by using the same set of simulations for both. DESPOT uses a sparse belief tree for planning. It samples a fixed set of scenarios and creates a sparse belief tree based on that. Both POMCP and DESPOT iteratively construct the tree by running a large number of simulations.

Monte Carlo Tree Search is a search algorithm in which each tree node represents a state s .Each node stores a value and a visitation count. The value of each action, Q(s,a), is determined by the average of all simulations in which that action a was selected from state s, and the visitation count N(s,a) is the sum of the number of times action a was selected when in node s. MCTS is comprised of four major steps as detailed in Fig 1: Selection, Expansion, Simulation and Backpropagation. In the Selection step, the most appropriate child node is selected until the end of the tree is reached. In the Expansion step, a single leaf node is added to the tree. In the Simulation step, simulations are run from the leaf node until termination. Finally, during Backpropagation, the values of the nodes are updated with the results of the simulations.

POMCP is based on MCTS described above. However, in POMCP the search is conducted in history space, rather than state space as in MCTS. Moreover, POMCP updates the belief state together with the tree search. POMCP[3] uses a modified MCTS and Monte Carlo belief state updates. Each node of the tree contains $(N(h), V(h), B(h))$, where $N(h)$ is the visitation count for history h, $V(h)$ is the value of the node, and $B(h)$ is the set of particles representing belief state at that history. A look ahead search tree is built from the current history. A state is sampled from the current belief and simulations are run from that state, in accordance with the Partially Observable UCT algorithm. If child nodes are available, then the UCB1 formula is used to select a node:

$$V_{aug} = V(ha) + c\sqrt{\frac{\log(N(h))}{N(ha)}} \tag{1}$$

The action which maximizes V_{aug} is chosen. If child nodes are not available, the rollout policy is used to add a single node to the tree. The rollout policy can be determined by the user and could be as simple as randomly sampling an action. The belief state for each node is updated to include the simulation state for the different $(action, observation)$ pairs encountered while running simulations. After the search tree is built, the best action a_{best} is selected and executed and an observation o_{new} is received from the environment. This helps in pruning the tree, and $(N(h, a_{best}, o_{new}), V(h, a_{best}, o_{new}), B(h, a_{best}, o_{new}))$ becomes the new root.

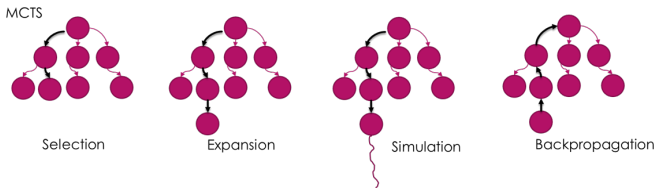

Fig. 1. Visual depiction of the stages of a Monte-Carlo Tree Search (MCTS).

There have been several studies on parallelizing Monte Carlo Tree Search (MCTS) and MCTS-based algorithms. In [11], three different ways to parallelize MCTS are introduced: root parallelization, tree parallelization and leaf Parallelization.In root parallel MCTS (Fig. 2), several search trees are built in parallel, each carrying out a fraction of the overall number of simulations. There is no shared information across the trees, and each one is built by a separate thread. After completion, the results from the trees are merged to obtain a final output. In tree parallel POMCP (Fig. 3), several threads carry out all four stages of MCTS simultaneously on the same search tree. In [12] the author

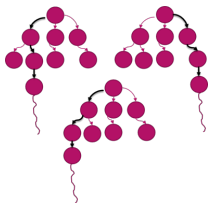

Fig. 2. Root parallel MCTS

looks into the limitations of parallel MCTS, and analyzes how well MCTS can scale. In [6] , a novel approach to parallelizing MCTS is introduced, by using a new set of statistics which keep track of the simulations in progress, to better parallize MCTS. Two different parallelized versions of UCT (Upper Confidence bounds for Trees) leveraging GPUs are introduced in [13].In [14], a lockfree approach to Tree-parallel MCTS is introduced.

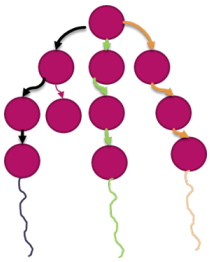

Fig. 3. Tree parallel MCTS

We consider variations of Root and Tree parallelization in the context of planning under uncertainty. Since POMCP operates on the history space, apart from updating the values and visitation counts in each node, we also have to keep track of the history while parallelizing the search. Each simulation updates the history with its own unique action observation pair and multiple simulations on multiple threads can easily lead to desynchronized search and eventual collapse. Moreover, each node also maintains its own set of beliefs unlike MCTS which doesn't involve a belief update. So, in addition to values and counts, updating the belief states also need to be synchronized.

HyP-DESPOT [15] proposes a parallelized version of DESPOT using a combination of CPU and GPUs.HyP-DESPOT uses Tree parallelization in the context of DESPOT tree search. Since DESPOT uses a sparse belief tree conditioned on sampled scenarios prior to search, it is more structurally suitable for a parallel implementation since several branches can be explored simultaneously.The major challenge to parallelizing POMCP is the inherently sequential nature of the algorithm. As detailed above, each simulation depends on the update from the previous simulation. Parallelizing POMCP, at it's core, involves running simulations in parallel instead of iteratively, which means not all simulations will be run on updates from previous ones. Therefore, parallelizing POMCP to cut down on the run time might affect the reward. Our goal is to parallelize it to reduce the runtime without affecting the reward. To the best of our knowledge, this is the first effort to parallelize POMCP for faster online planning under uncertainty.

III. METHOD

A. Root Parallel POMCP

This is the simplest version of parallel POMCP. Similar to Root parallel MCTS, instead of building a single look ahead search tree in POMCP, we build multiple trees in parallel and merge the results. If the serial version built a search tree on n simulations, the root parallel built t trees each using $\frac{n}{t}$ number of simulations.

Each tree reports an optimal action. A vote is taken on the t actions reported, and the most popular one is chosen to be executed in the real world. Each tree maintains its own separate belief state. These belief states were pruned using the voted action and observation received. The overall process is shown in Fig. 4. However, there are several limitations to this method. Using fewer number of simulations per tree results in less accurate predictions. Similar computations have to be repeated across all trees, therefore increasing redundancy. Moreover, one tree might take significantly longer than other due to the random nature of rollouts at the leaf node. We would have to wait for the tree that takes the longest in order to proceed. The fact that we have to maintain different belief states for the same problem is also unintuitive and might lead to a lot of extra computation that doesn't contribution anything towards the solution. However maintaining the same belief state would requiring sharing information across trees or merging the trees, which is computationally expensive and will forego any speedup we would receive from running fewer simulations per tree. For all these reasons, we decided that a Tree Parallel approach is much more suitable to parallelizing POMCP.

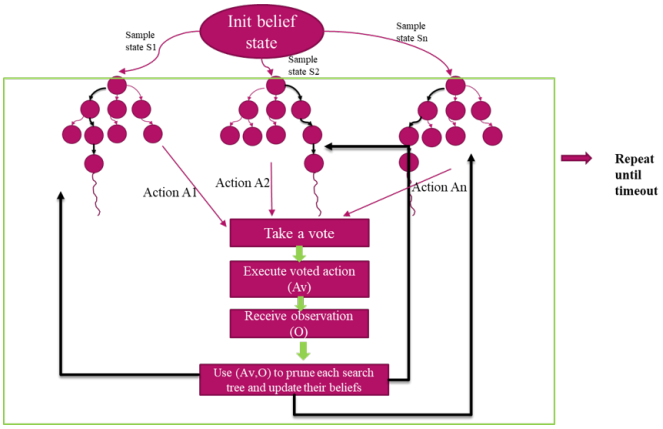

Fig. 4. Root Parallel POMCP

B. Tree Parallel POMCP

In this version of POMCP, we deploy multiple threads at once to build the look ahead search tree simultaneously. So, the selection, expansion, simulation and backpropagation steps can all occur simultaneously in different parts of the tree. One thread could be in the simulation phase, while the other could be in the backpropagation phase on different branches of the tree. The threads each sample a random state from the belief state of the Root, given its history, and then proceed to build the tree independent of each other.

We segregate "history" as defined in original POMCP into two different kinds to allow for parallelization: (1) Real History (2) Simulated History. Real History is a sequence of Real Actions implemented by the agent and real observations received from the environment. Simulated History, on the other hand, is private to each thread and instantiated to the real History at the beginning of simulations. The simulated history repeatedly updates itself during the course of the Monte Carlo simulations, with simulated actions and observations.Segregating the simulated history from the real history lets the threads execute independent of each others route.

Building the search tree requires cooperation between the threads to prevent race conditions. We propose two different

architectures for synchronized search - one using locks and the other without using locks.

1) ***Tree Parallelization with locks****:* Each thread samples a random state from the belief of the root, initializes it's own private simulated history by copying the real history to it, and then runs a specific number of simulations to build the tree. Each node of the tree contains three values $(N(h), V(h), B(h))$ - the visitation count $N(h)$ of history h, the value $V(h)$, and the belief state $B(h)$. Each node is shared among multiple threads. Each thread maintains its own history, as the simulation proceeds, and separately updates the belief state of the node based on new history encountered while simulating. This means, the visitation count $N(h)$, the value $V(h)$ and the belief state $B(h)$ maintained at each node are shared values,although the history/route h each thread takes to arrive at these nodes might be different.So there are 2 possibilities for race conditions which need to be considered:

- During backpropagation: This is when the visitation count and value is updated. If multiple threads try to write to a node at the same time, it would lead to a race condition.
- Updating belief state: When a new history is encountered, the belief of the node has to be updated to include the simulation state. If multiple threads try to modify the belief of the same node, it would also lead to race conditions.

We explore locking at two different levels:

- Process level locking: In this we locked the backpropagation update process and the belief state update process. So, only one thread would be able to backpropagate at an instant, and same for belief state update. However, this lead to a lot of runtime overhead and performance was worse than serial.
- Node level locking: In this we instantiate a lock per node. So, any update to a node would be protected. And several nodes can be updated at once. This is essentially the most fine-grained locking possible. Node level locking is shown in Figure 5. This gaves us significant speedup over the serial version across several domains.

The algorithm for tree parallel POMCP with node-level locking is given below. It has been adapted from the POMCP algorithm presented in [3]:

2) ***Lockfree Tree Parallelization****:* In the previous section each node in the tree had it's own unique mutex to protect updates to the node. Every time a thread wanted to update the values of the node, it would have to acquire the lock to prevent race conditions. In order to improve speedup further, we remove locks completely from all nodes. In order to do so in a threadsafe manner, we had to make a few changes to parallel POMCP described above. First, the search and the belief update had to be separated. And second, the updates to the values and visitation counts of each node were converted into atomic operations in C++. Unfortunately, while updating values can be set to be atomic in nature, the belief update

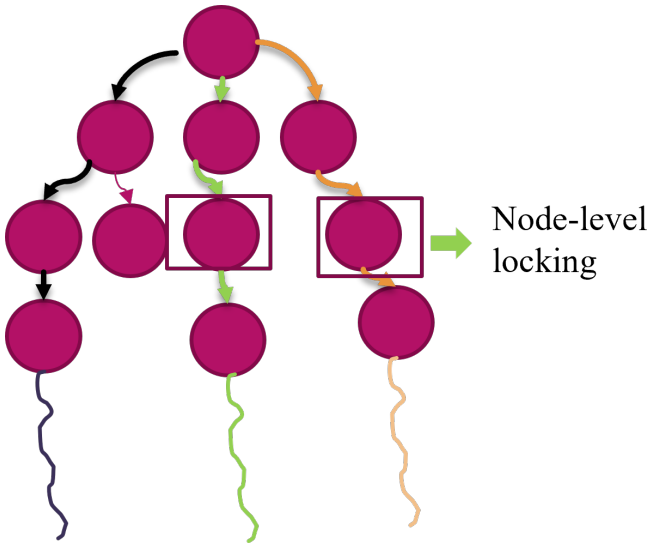

Fig. 5. Tree Parallel POMCP

cannot be done atomically. So the belief update is done separately.Once the search is complete, an optimal action is selected and thne agent receives an observation from the environment. A belief update is carried out by keeping all particles which match the actual observation received.

The search algorithm now is a parallelized version of PO-UCT [3]. PO-UCT was also introduced by Silver in [3] and is similar to POMCP. The search occurs in history space. Each node of the tree now contains two values $(N(h), V(h))$, the visitation count and the value. Performing a parallel PO-UCT search followed by a particle belief update is equivalent to parallel POMCP where both the search and belief update occur simultaneously. However, since we only consider the search time, using PO-UCT might give us some advantage in speedup.It allows allows for a simpler implementation with less overhead. The algorithm is provided in Algorithm 2.The belief update done separately is given in Algorithm 3

3) ***Strategic deployment of thread****:* The threads should ideally explore different branches simultaneously. If they all go down the same route, it defeats the purpose of parellelizing the search and also leads to increased overhead due to frequent locking. POMCP uses the UCB1 formula (Eq.1) for node selection. UCB1 trades off exploration and exploitation and helps the threads strike a balance between the two.

To enhance the tradeoff, we explore virtual loss as introduced in [11]. A loss is assigned to the value of a node as soon as a thread enters it. This discourages other threads from going down the same route. Once the simulation is complete, the virtual loss value is added back to the nodes along the branch.

We also explore a modified version of the UCB1 formula to better deploy threads.In [6], an interesting statistic is introduced , for parallel MCTS, which keeps track of the number of ongoing simulations O at a particular node. Whenever a thread enters a node, this value is incremented.Upon completion of a simulation,as the thread leaves that node,

Algorithm 1 Parallel POMCP. Algorithm adapted from serial POMCP algorithm in Silver and Veness [3]

procedure SEARCH(h)
  repeat } executed in parallel
    if h == empty **then**
      $s \sim I$
    else
      $s \sim B(h)$
      $h_{priv} \leftarrow h$
    end if
    SIMULATE(s,h$_{priv}$, 0)
  until Timeout
  return $argmax_b V(hb)$
end procedure

procedure SIMULATE(s,h,depth)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    Memorypool $\leftarrow$ **locked()**
    for all $a \in A$ **do**
      $T(h,a) \leftarrow (N_{init}(h,a), V_{init}(h,a), \phi)$
    end for
    Memorypool $\leftarrow$ **unlocked()**
    return ROLLOUT$(s,h,depth)$
  end if
  $a \leftarrow argmax_b V(hb) + c\sqrt{\frac{\log(N(h)}{n(hb)}}$
  $(s',o,r) \sim G(s,a)$
  $R \leftarrow r + \gamma$SIMULATE(s',hao,depth+1)
  T(h) $\leftarrow$ **locked()**
  $B(h) \leftarrow B(h) \cup s$
  $N(h) \leftarrow N(h) + 1$
  T(h) $\leftarrow$ **unlocked()**
  T(ha) $\leftarrow$ **locked()**
  $N(ha) \leftarrow N(ha) + 1$
  $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
  T(ha) $\leftarrow$ **unlocked()**
  return R
end procedure

procedure ROLLOUT(s,h,depth)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  $a \sim \pi_{roollout}(h)$
  $(s',o,r) \sim G(s,a)$
  return $r + \gamma$ROLLOUT(s',hao,depth+1)
end procedure

Algorithm 2 Lockfree Parallel POMCP. Algorithm adapted from serial POMCP algorithm in Silver and Veness [3]

procedure SEARCH(h)
  repeat } executed in parallel
    if h == empty **then**
      $s \sim I$
    else
      $s \sim B(h)$
      $h_{priv} \leftarrow h$
    end if
    SIMULATE(s,h$_{priv}$, 0)
  until Timeout
  return $argmax_b V(hb)$
end procedure

procedure SIMULATE(s,h,depth)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  if $h \notin T$ **then**
    Memorypool $\leftarrow$ **locked()**
    for all $a \in A$ **do**
      $T(h,a) \leftarrow (N_{init}(h,a), V_{init}(h,a))$
    end for
    Memorypool $\leftarrow$ **unlocked()**
    return ROLLOUT$(s,h,depth)$
  end if
  $a \leftarrow argmax_b V(hb) + c\sqrt{\frac{\log(N(h)}{n(hb)}}$
  $(s',o,r) \sim G(s,a)$
  $R \leftarrow r + \gamma$SIMULATE(s',hao,depth+1)
  atomic update:
  $N(h) \leftarrow N(h) + 1$
  atomic update:
  $N(ha) \leftarrow N(ha) + 1$
  atomic update:
  $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
  return R
end procedure

procedure ROLLOUT(s,h,depth)
  if $\gamma^{depth} < \epsilon$ **then**
    return 0
  end if
  $a \sim \pi_{roollout}(h)$
  $(s',o,r) \sim G(s,a)$
  return $r + \gamma$ROLLOUT(s',hao,depth+1)
end procedure

the value is decremented. At any given time, the value of O can directly tell us how many threads are running simulations through a given node.This value can be used for better selection of actions using the updated UCB1 formula presented in [6] :

$$V_{aug} = V(ha) + c\sqrt{\frac{\log(N(h) + O(h))}{N(ha) + O(ha)}} \tag{2}$$

We explored using virtual loss, the additional statistic of tracking ongoing simulations, as well a combination of both. The extra speedup we got from using these techniques was pretty minimal. Our conclusion is, for parallel POMCP, the exploration-exploitation tradeoff afforded by the UCB1

Algorithm 3 Belief update for Lockfree POMCP

```
procedure BELIEFUPDATE(a,obs)
    newBelief ← empty
    for all s ∈ belief do
        (s', o, r) ~ G(s, a)
        if o == obs then
            newBelief ← newBelief ∪ s
        end if
    end for
    return newBelief
end procedure
```

formula is sufficient to deploy threads evenly across the search tree.

IV. Experimental results in simulations

A. Comparison to Serial

We ran experiments on four different domains: Rocksample,Battleship,Pocman and Findit. Rocksample,Battleship and Pocman were introduced in the POMCP paper by Silver and Veness [3].FindIt is our own custom domain.We benchmark the performance of our parallel implementations against the serial version in two areas: 1) the average time to select an action at a given step, and 2) the overall reward. Our Tree parallel implementations were run on all four domains. We record the time needed to select an action (y-axis) for a certain number of simulations (on the x-axis) for the serial and parallel versions.We also plot the reward obtained on the domain for running different number of simulations, and show that the rewards from the parallel versions are consistent with those obtained from the serial version. A short description for each domain along with the results are given below.

1) Battleship: The Battleship domain is introduced in Silver and Veness [3]. There is a grid of size $n \times n$, containing m ships. Each ship has a different size and ships cannot be adjacent or diagonal to each other. The agent does not know the locations of the ships, but is tasked with sinking all ships. The agent is allowed to fire on each cell in the grid, but it cannot fire on the same cell more than once. Each action taken incurs a -1 reward (representing a time cost), so the agent must sink the ships in the least number of moves possible.

2) Rocksample: In the Rocksample problem [16], a rover is placed in an $n \times n$ grid. There are m rocks scattered on the grid and each of the rocks could be valuable or not. The agent is tasked with sampling the valuable rocks, and moving towards the exit area to the east.We use the version introduced by Silver and Veness [3]

3) FindIt: In this problem we consider a square grid world of size n. An object is placed at any coordinate on the grid. The agent doesn't know where the object is but can move to any grid cell and get a binary observation on whether the object is present or not.The program terminates once the agent finds the object. The agent also keeps track of the cells

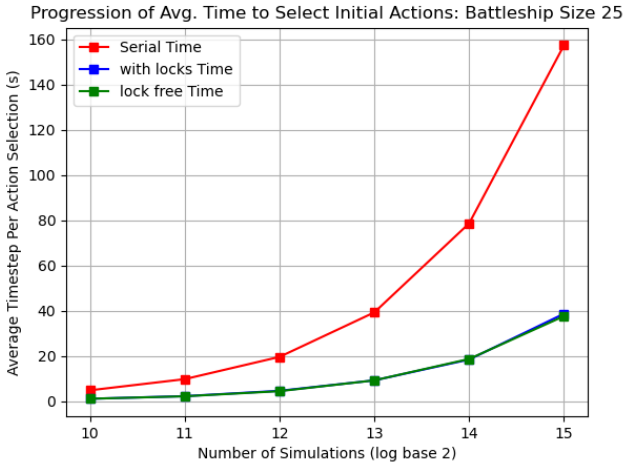

Fig. 6. Average time taken per action selection on Battleship (Locked/Lockfree tree parallel and serial)

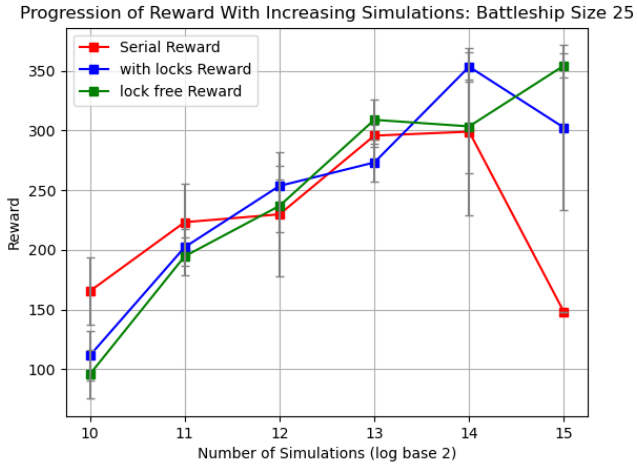

Fig. 7. Cumulative reward for Battleship (Locked/Lockfree tree parallel and serial)

visited and is penalized if it moves to a visited cell again. It is heavily penalized if it receives a positive observation but does not find the object at that location. There is also a per step penalty to reduce the number of steps.

4) Pocman: This is another domain introduced by Silver and Veness [3].It is a partially observable adaptation of the video game Pacman.

B. Effect of Action Space

The performance of parallel POMCP improves with the size of the action space.The Tree Parallel POMCP was tested on the Battleship domain of different sizes.On the battleship domain of size 10, which has an action space of 100, a 2x speedup was achieved (Fig. 14). On the battleship domain of size 25 , which has an action space of 625, a 4x speedup was achieved (Fig. 15). On the battleship domain of size 30, which has an action space of 900, a 5x speedup was achieved (Fig. 16).

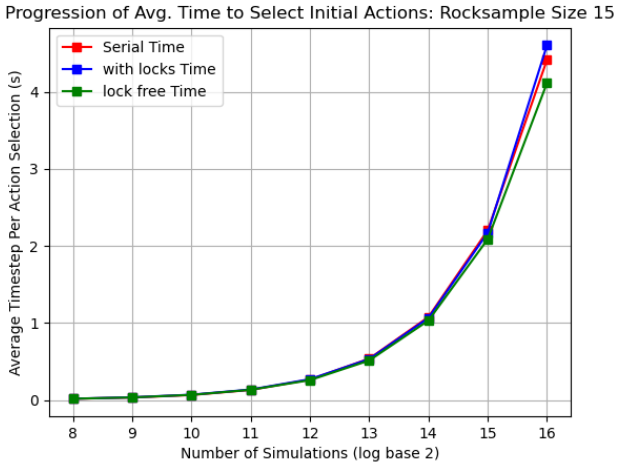

Fig. 8. Average time taken per action selection on Rocksample (Locked/Lockfree tree parallel and serial)

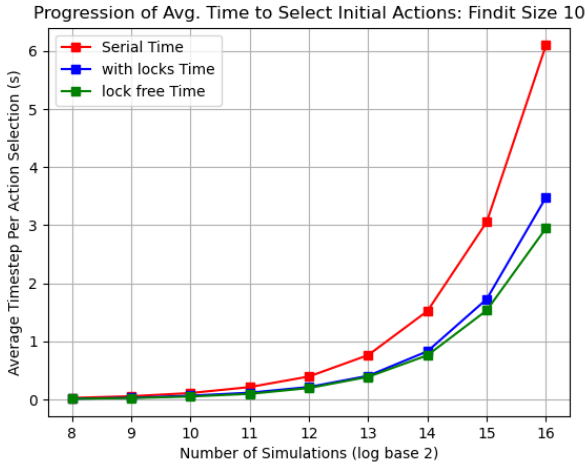

Fig. 10. Average time taken per action selection on FindIt (Locked/Lockfree tree parallel and serial)

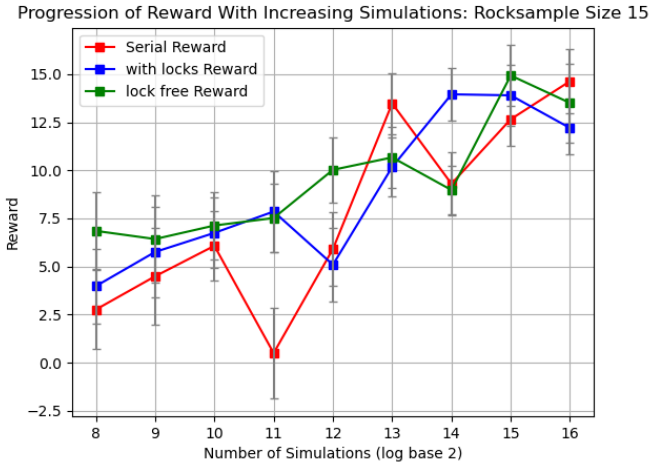

Fig. 9. Cumulative reward for Rocksample (Locked/Lockfree tree parallel and serial)

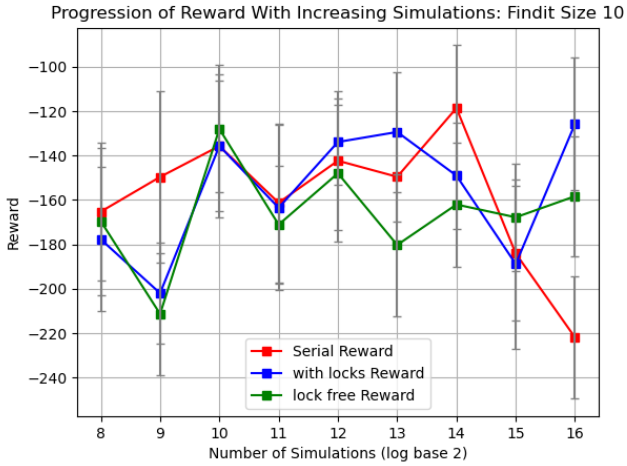

Fig. 11. Cumulative reward for FindIt (Locked/Lockfree tree parallel and serial)

Simulations for parallel POMCP start at the root node. Actions connect the root node to child nodes.If the number of actions in a domain is less, then it leads to a narrow search tree and increased locking in the parallel version.

C. *Effect of number of threads*

Our experiments were conducted on a 6-core machine with hyperthreading that supports 2 threads per core. We ran parallel POMCP using different number of threads and report our findings in Fig 17 and Fig 18.

D. *Effect of different heuristics in Search*

As described before, we tried several approaches to effectively deploy threads across the search tree. We tried virtual loss, the Wu-Uct statistic of tracking ongoing simulations and a combination of both. The effect on the speedup and reward was minimal as can be seen in the plots given in Fig 19 and Fig. 20.:

V. DISCUSSION

We introduce different ways to parallelize POMCP and achieve upto 5x speedup without any change in the reward.We introduce two different ways to use Tree parallelization in the context of POMCP, one with fine grained locking and one in a lockfree manner. The speed achieved however, heavily depends on the domain being explored. Tree parallel POMCP scales well with an increase in the number of actions that the agent can take. However, it only shows marginal improvement when the action space is smaller.

In the future, we want to explore how to scale Parallel POMCP further. Since we have removed as much redundant locking as possible, the next step to improving performance would be alter the core POMCP algorithm in a way that lends well to parallelization. We also want to perform extensive experiments on actual robots in different domains, and analyze the speedup and accuracy in different applications.

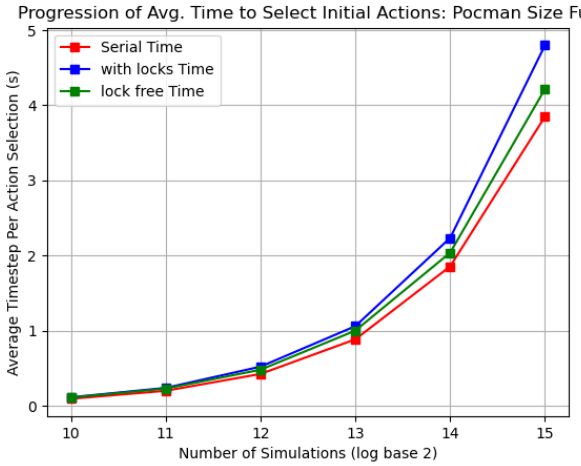

Fig. 12. Average time taken per action selection on Pocman (Locked/Lockfree tree parallel and serial)

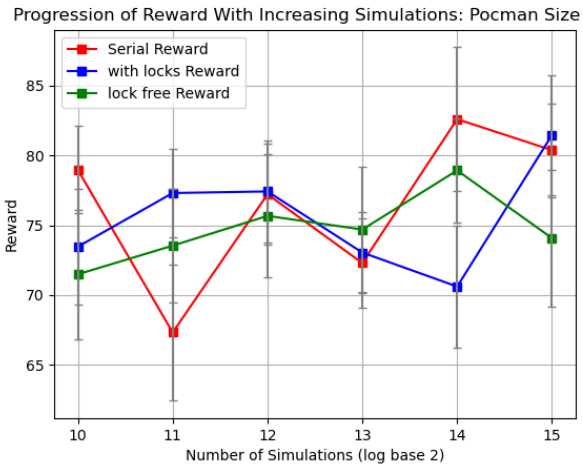

Fig. 13. Cumulative reward for Pocman (Locked/Lockfree tree parallel and serial)

REFERENCES

[1] 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.

[2] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1, pp. 99–134, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437029800023X

[3] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," in *Advances in Neural Information Processing Systems*, J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, Eds., vol. 23. Curran Associates, Inc., 2010. [Online]. Available: https://proceedings.neurips.cc/paper/2010/file/edfbe1afcf9246bb0d40eb4d8Paper.pdf

[4] J.-A. Delamer, Y. Watanabe, and C. P. Chanel, "Safe path planning for uav urban operation under gnss signal occlusion risk," *Robotics and Autonomous Systems*, vol. 142, p. 103800, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889021000853

[5] Y. Wang, F. Giuliari, R. Berra, A. Castellini, A. D. Bue, A. Farinelli, M. Cristani, and F. Setti, "Pomp: Pomcp-based online motion planning for active visual search in indoor environments," *ArXiv*, vol. abs/2009.08140, 2020.

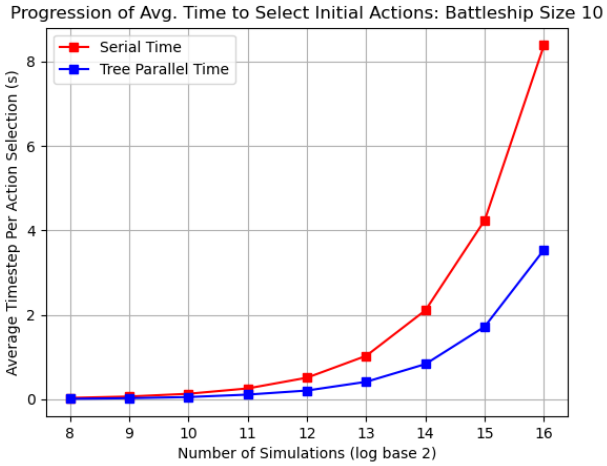

Fig. 14. 2x speedup in Battleship size 10x10

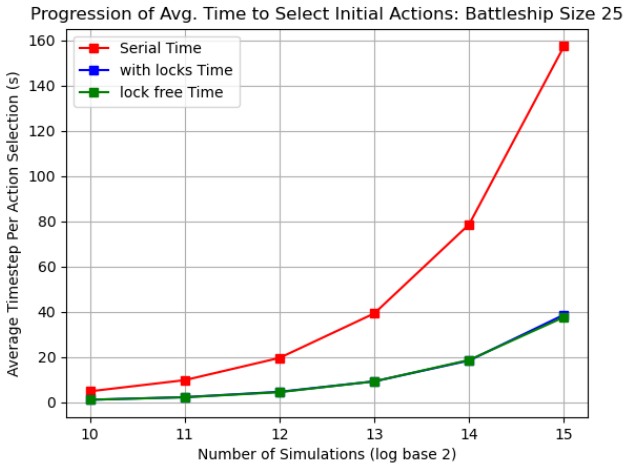

Fig. 15. 4x speedup in Battleship size 25x25

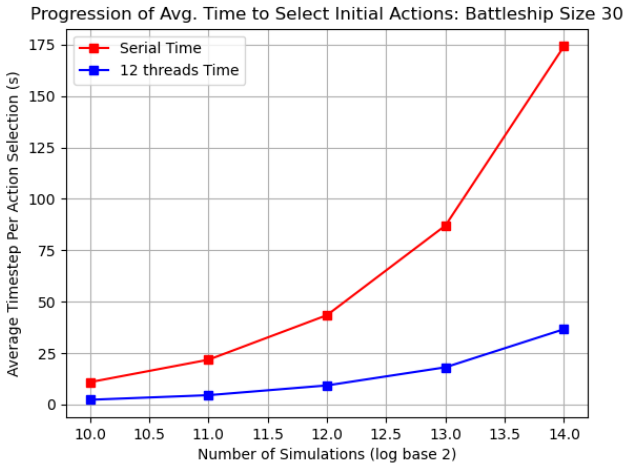

Fig. 16. 5x in Battleship size 30x30

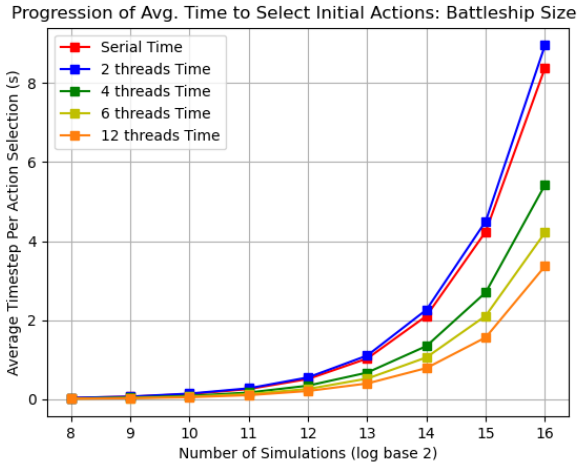

Fig. 17. Average time take per action selection in Battleship for different number of threads

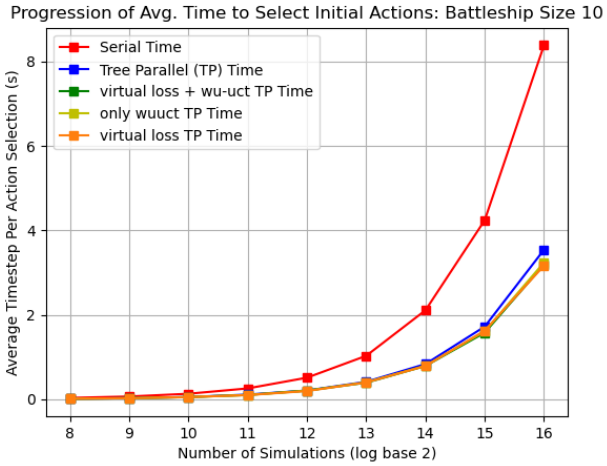

Fig. 19. Average time take per action selection using different heuristics

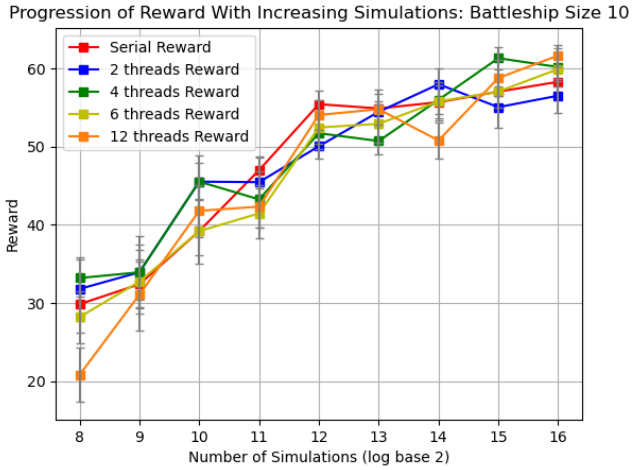

Fig. 18. Cumulative reward for Battleship on using different number of threads

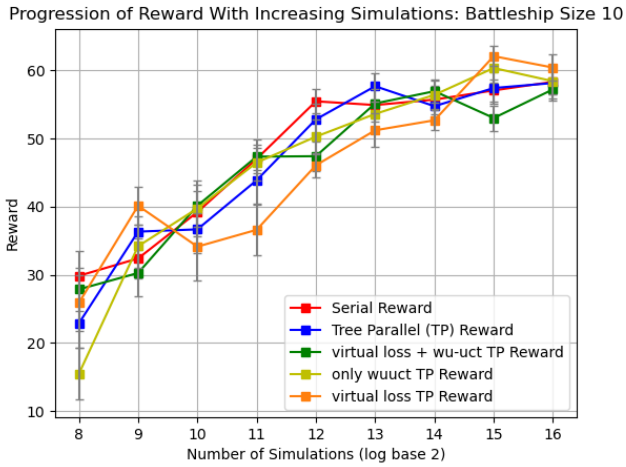

Fig. 20. Cumulative reward on using different heuristics

[6] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu, "Watch the unobserved: A simple approach to parallelizing monte carlo tree search," in *International Conference on Learning Representations*, Apr. 2020. [Online]. Available: https://openreview.net/forum?id=BJlQtJSKDB

[7] N. Roy and S. Thrun, "Coastal navigation with mobile robots," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper/1999/file/df9028fcb6b065e000ffe8a4f03eeb38-Paper.pdf

[8] K. Hsiao, L. P. Kaelbling, and T. Lozano-Perez, "Grasping pomdps," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 2007, pp. 4685–4692.

[9] F. Doshi and N. Roy, "Efficient model learning for dialog management," in *2007 2nd ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, 2007, pp. 65–72.

[10] A. Somani, N. Ye, D. Hsu, and W. S. Lee, "Despot: Online pomdp planning with regularization," in *Advances in Neural Information Processing Systems*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper/2013/file/c2aee86157b4a40b78132f1e71a9e6f1-Paper.pdf

[11] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 60–71.

[12] R. B. Segal, "On the scalability of parallel uct," in *Computers and Games*, H. J. van den Herik, H. Iida, and A. Plaat, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 36–47.

[13] N. A. Barriga, M. Stanescu, and M. Buro, "Parallel uct search on gpus," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014, pp. 1–7.

[14] S. A. Mirsoleimani, H. J. van den Herik, A. Plaat, and J. A. M. Vermaseren, "A lock-free algorithm for parallel mcts," in *ICAART*, 2018.

[15] P. Cai, Y. Luo, D. Hsu, and W. S. Lee, "Hyp-despot: A hybrid parallel algorithm for online planning under uncertainty," in *Robotics: Science and Systems XIV, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, June 26-30, 2018*, H. Kress-Gazit, S. S. Srinivasa, T. Howard, and N. Atanasov, Eds., 2018. [Online]. Available: http://www.roboticsproceedings.org/rss14/p04.html

[16] T. Smith and R. Simmons, "Heuristic search value iteration for pomdps," in *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, ser. UAI '04. Arlington, Virginia, USA: AUAI Press, 2004, p. 520–527.