



Massively parallel motion planning algorithms under uncertainty using POMDP

Taekhee Lee and Young J. Kim

Abstract

We present new parallel algorithms that solve continuous-state partially observable Markov decision process (POMDP) problems using the GPU (gPOMDP) and a hybrid of the GPU and CPU (hPOMDP). We choose the Monte Carlo value iteration (MCVI) method as our base algorithm and parallelize this algorithm using the multi-level parallel formulation of MCVI. For each parallel level, we propose efficient algorithms to utilize the massive data parallelism available on modern GPUs. Our GPU-based method uses the two workload distribution techniques, compute/data interleaving and workload balancing, in order to obtain the maximum parallel performance at the highest level. Here we also present a CPU–GPU hybrid method that takes advantage of both CPU and GPU parallelism in order to solve highly complex POMDP planning problems. The CPU is responsible for data preparation, while the GPU performs Monte Carlo simulations; these operations are performed concurrently using the compute/data overlap technique between the CPU and GPU. To the best of the authors' knowledge, our algorithms are the first parallel algorithms that efficiently execute POMDP in a massively parallel fashion utilizing the GPU or a hybrid of the GPU and CPU. Our algorithms outperform the existing CPU-based algorithm by a factor of 75–99 based on the chosen benchmark.

Keywords

POMDP, Monte Carlo value iteration, GPU, CPU–GPU, gPOMDP, hPOMDP

1. Introduction

Motion planning is an important problem in robotics, intelligent systems, computer animation, game artificial intelligence (AI), robotic surgery, computer-aided drafting (CAD), and manufacturing systems. Motion planning focuses on determination of a safe path or series of actions that enable robots to reach a target while minimizing resources. Many approaches have been proposed to solve motion planning problems including the use of roadmaps, cell decompositions, potential fields, and mathematical programming (Latombe, 1991). These approaches are used to solve general motion planning problems under the assumption that the information about a robot and its surrounding environment are known. Also, they assume that the robot controllers and sensors are precise in terms of their functionalities. However, these assumptions are only valid in well-controlled environments, such as manufacturing assembly lines. In practice, a robot in the real world encounters many uncertainties caused by inaccurate sensors, imprecise controls, and unexpected changes in and imperfections of information about the real-world environment. These inaccuracies can lead the robot to make wrong

motions and seriously risk the reliability of robot motion planning.

Over recent years, a significant body of research has attempted to address such uncertainty; for instance, using sampling-based planning with sensor uncertainty (Brendan and Oliver, 2007), evaluating an uncertain region in the configuration space (Guibas et al., 2008), using the probabilistic roadmap algorithm for robust motion plans (Missiuro and Roy, 2006), employing a priori probability distributions along the robot path (Berg et al., 2010), etc. However, a more general framework to address uncertainties arising in different parts of motion planning is the partially observable Markov decision process (POMDP)-based planner (Kaelbling et al., 1998). In principle, this method

Department of Computer Science and Engineering, Ewha Womans University, Seoul, Korea

Corresponding author:

Young J. Kim, Department of Computer Science and Engineering, Ewha Womans University, 52, Ewhayeodae-gil, Seodaemun-gu, Seoul 120–750, Korea.
Email: kimy@ewha.ac.kr

addresses all uncertainties with its stochastic planning process; for instance, a robot's state is defined as a belief represented by a probability distribution over a state space.

Even though the POMDP model can deal with a wide range of uncertainties, it is notoriously challenging to precisely evaluate, in particular, continuous-state POMDPs are computationally intractable, i.e. PSPACE-hard (Madani et al., 1999; Papadimitriou and Tsitsiklis, 1987). In recent years, more progress has been made in developing an efficient, approximated POMDP-based algorithm. Notably, point-based algorithms using POMDP model have drastically reduced the amount of computation required while retaining the intrinsic value of POMDP, thus making the POMDP model a more appealing choice for handling uncertainties in motion planning. However, even these efficient algorithms may still require hours of computation time to deal with moderate-size POMDP problems.

Since the introduction of programmable GPUs (graphics processing units), many researchers have sought to solve challenging computational problem by using the massive parallelism of GPU threads (Luebke et al., 2004). The main theme of this paper is to use the readily available massive parallelism of GPUs to accelerate POMDP computation. However, parallelizing POMDP-based algorithms on GPUs is very non-trivial. First of all, even though POMDP requires a large amount of memory to deal with the challenges of dimensionality and history, the typical memory size of a GPU is much less than that of a CPU. Moreover, the memory architecture of modern GPUs is hierarchical, and its caching capability is less efficient than that of the CPU counterpart. Finally, existing efficient POMDP-based algorithms such as those in Kurniawati et al. (2008), Kurniawati et al. (2011) and Berg et al. (2012a) are iterative and sequential by nature; thus, identifying parallelism among these algorithms is challenging. For instance, a simple parallel approach introduced by Bai et al. (2010) may not fully utilize the computing power of a GPU since they rely on only fine-grained parallelism.

1.1. Main contributions

In this paper, we present a GPU-based method (gPOMDP) for solving POMDP planning problems with a moderate complexity as well as a CPU-GPU hybrid method (hPOMDP) for solving more complicated POMDP planning problems. Both methods are designed to solve continuous-state POMDP problems. A preliminary version of our GPU-based method (i.e. gPOMDP) appeared in Lee and Kim (2013).

The gPOMDP is based on the MCVI (Monte Carlo value iteration) method, originally developed for CPUs (Bai et al., 2010), and we parallelize this algorithm using a novel, multi-level parallel formulation of MCVI.

The parallel algorithm can be executed at three different levels: belief, action, and policy graph node levels. For each level of parallelism, we propose efficient algorithms for use in a GPU. In particular, to achieve maximum parallel performance at the highest level (i.e. the belief level),

we introduce the two workload distribution techniques of the compute/data interleaving technique and workload balancing using the paradigm of gathering, scheduling and running (GSR). Our gPOMDP algorithm is designed to handle a moderately complicated POMDP problem with less than five POMDP state variables (i.e. degrees of freedom). However, for a larger number of POMDP state variables, the GPU needs to spend a significant amount of time generating and simulating particles.

Meanwhile, our hPOMDP is a hybrid method that utilizes both CPU and GPU parallelism to address more complicated POMDP problems. In hPOMDP, the CPU is responsible for data preparation such as generating random particles, while the GPU performs concurrent MC (Monte Carlo) simulations using a compute/data overlap technique between the CPU and the GPU.

To the best of the authors' knowledge, our algorithm is the first massively parallel algorithm that efficiently executes POMDP. In our experiments, both gPOMDP and hPOMDP outperform the existing CPU-based algorithm by a factor of 75–99 based on the chosen benchmark.

2. Previous work

In this section, we briefly survey motion planning algorithms related to both POMDP and GPU parallelization.

2.1. POMDP-based motion planning

2.1.1. Discrete-state POMDP. The POMDP provides a general framework for planning using the imperfect information of robot state, sensors and actions (Kaelbling et al., 1998; Smallwood and Sondik, 1973; Sondik, 1971), applicable to such areas as operation research, artificial intelligence, and robotics (Cassandra et al., 1996; Simmons and Koenig, 1995; Theodorou and Magadevan, 2002). Unfortunately, however, solving a POMDP is computationally intractable due to the challenges of dimensionality and history (Madani et al., 1999; Papadimitriou and Tsitsiklis, 1987). Thus, several approximate methods have been proposed to convert POMDP to a fully observable Markov decision process (MDP) by applying heuristic strategies (Cassandra et al., 1996; Simmons and Koenig, 1995; Theodorou and Magadevan, 2002). Alternatively, point-based algorithms have been successfully shown to approximately solve a POMDP by operating in a limited subset of belief space (Hoey et al., 2007; Hsu et al., 2008; Pineau et al., 2003; Shani et al., 2007; Smith and Simmons, 2005; Spann and Vlassis, 2004).

Several methods such as those in Smith and Simmons (2005), Spann and Vlassis (2004), Shani et al. (2007), Pineau et al. (2003) and Hsu et al. (2008) solve a POMDP by sampling beliefs only from reachable spaces. They maintain both upper and lower bounds of the optimal value function in order to test whether the sampled point is reachable (Kurniawati et al., 2008). Even when these point-based

methods can solve a POMDP with tens of thousands of states in reasonable time, they still require large amounts of computation time and memory.

2.1.2. Continuous-state POMDP. To deal with real-world problems, POMDPs need to model continuous states, actions and observations. To plan in continuous environments, the surroundings should be discretized with a grid (Roy, 2003; Thrun, 2000). However, it is rather difficult to determine the proper dimensions of the grid, and the discretization may deteriorate the quality of a solution.

To address this issue, Brooks et al. (2006), Brunskill et al. (2008), Porta et al. (2006) and Prentice and Roy (2007) used Gaussian particle filters to represent beliefs. However, if the environment has discontinuities such as obstacles, the number of Gaussian components increases quickly and becomes difficult to control. Thus, Berg et al. (2011, 2012a,b) and Patil et al. (2011) approximated the belief dynamics using an extended Kalman filter and provided locally optimal solutions to continuous POMDP problems in polynomial time. The research of Patil et al. (2012) introduces a method for fast and safe motion planning using the probability of collision.

MCVI-based methods (Bai et al., 2010; Lim et al., 2011; Thrun, 2000) solve continuous POMDP problems with an acceptable result and speed. The MCVI also uses particles to represent a belief and employs the notion of α vectors to implicitly represent a policy graph (Hansen, 1998; Kaelbling et al., 1998). To build a policy graph, MCVI performs MC simulations. Although MCVI involves more computation than approximation methods such as those in Thrun (2000) and Roy (2003), it requires less memory by avoiding inefficient discretization. The experiments of Shani (2010) take advantage of multi-core CPUs to devise a scalable parallel algorithm for point-based POMDPs and then apply it to a large, complex domain.

2.1.3. GPU-based motion planning. Motion planning using the massively parallel power of GPUs is a relatively new field. Existing GPU-based planning algorithms mostly focus on accelerating collision queries that act as a bottleneck for sampling-based planners. GPU-based algorithms such as those of Pan and Manocha (2010, 2012) use a work queue to parallelize collision queries based on the bounding volume hierarchy. ITOMP (Park et al., 2012) computes a collision-free trajectory that is smooth and satisfies dynamic constraints on GPUs using a stochastic method. To the best of the authors' knowledge, accelerating point-based algorithms based on POMDP model using GPUs still remains an open problem (Shani et al., 2012).

3. Preliminaries of POMDP

In this section, we provide a preliminary description of the POMDP model to help elucidate our GPU-based algorithm.

3.1. Discrete-state POMDPs

Formally, the POMDP model is a tuple of $\{S, A, O, T, O(s), re_a(s)\}$ with a set of states S , actions A , observations O , transition mode $T(a, s) = p(s' | a, s)$, observation model $O(s) = p(o | a, s)$, and reward function $re_a(s) \in \mathbb{R}$. At a given moment, the system is in a state s , an agent executes an action a and receives a reward $re_a(s)$, and the system state changes to s' . The system state is represented as a *belief*, a probability distribution over S . If S is discrete, the belief b after executing a and observing o is represented as follows:

$$b(s') = \frac{p(o | s')}{p(o | a, b)} \sum_{s \in S} p(s' | s, a) b(s) \quad (1)$$

where $b(s)$ calculates the probability of a state s from the current belief b . A function which chooses an action from the current belief is called a *policy*, and it is optimal when it chooses the optimal action. A value function calculates the reward of a given belief and can be recursively expressed as

$$V_n(b) = \max_a Q_n(b, a) \quad (2)$$

with

$$Q_n(b, a) = \sum_{s \in S} re_a(s) b(s) + \gamma \sum_o p(o | b, a) V_{n-1}(b^{a,o}) \quad (3)$$

where S and O are discrete, $\gamma \in [0,1)$ is a discount factor, $b^{a,o}$ is the belief state after taking an action a and observing o in belief state b , and n is the time horizon. Now, an optimal policy π^* can be defined as

$$\pi^*(b) = \arg \max_a Q^*(b, a) \quad (4)$$

where Q^* is the Q -function associated with the optimal value function V^* . The optimal policy is often represented in a tabular form using beliefs and actions. An optimal policy can be obtained from V^* which is often approximated by recursively evaluating a sequence of functions V_i , also known as *value iteration* (Hauskrecht, 2000; Kaelbling et al., 1998; Sondik, 1971). Here, V_i , the value of the current belief, is expressed as

$$V_n(b) = \max_{\{\alpha_n^i\}_i} \sum_s \alpha(s) b(s) \quad (5)$$

$$\alpha(s) = \arg \max_{\{\alpha_n^i\}_i} b(s) \alpha_n^i$$

where $\{\alpha_n^i\}_i$ is a set of vectors, each of which provides the value of a belief state. The value function V_i is approximated by a piecewise linear function, and α is the gradient of the linear function. Value iteration generates a set of alpha vectors. In functional form, a value iteration is often expressed using the *Bellman recursion* (Bellman, 1957):

$$V_n = HV_{n-1} = \max_{a \in A} \{r(b, a) + \gamma \sum_{o \in O} p(o | b, a) V_{n-1}(b')\} \quad (6)$$

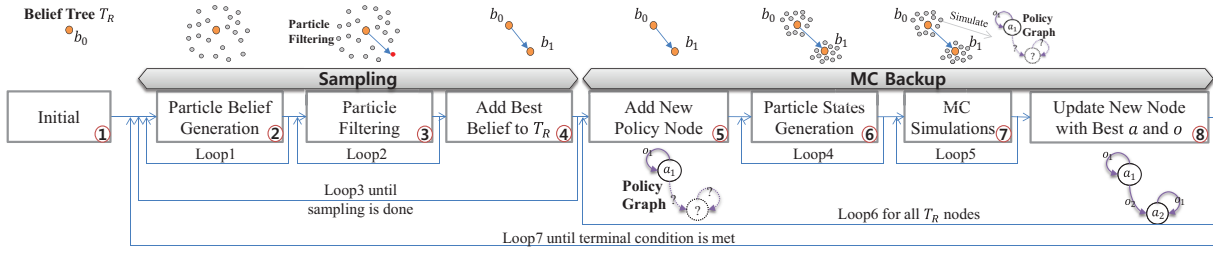


Fig. 1. The process of POMDP-based MCVI. The algorithm samples a new belief node that is reachable from the initial belief node using a particle-filtering method. Then, it updates the policy graph using the MCVI method. Note that many loops are required to identify the best node, best policy graph action and best observation.

where $r(b, a)$ is the agent's immediate reward for the given belief b and action a , and γ is the discount factor.

3.2. Continuous-state POMDPs using MCVI

For continuous-state POMDP-based algorithms such as the MCVI algorithm, a policy graph is used to represent a policy (Hansen, 1998; Kaelbling et al., 1998). A policy graph G is a directed graph with the nodes of an action $a \in A$ and the edges of an observation $o \in O$. An agent starts from a suitable node v of G , executes its corresponding policy, and moves to another node based on its observation as a result of executing the policy. For example, the starting node v_0 of the given policy graph in Figure 2 assigns the action a_2 to the agent. Then, the agent performs the action a_2 and receives the observation from the world. If the agent receives the observation o_2 from the world, the agent moves to the policy graph node v_2 and receives the action a_3 for the next action. The value function of b on a policy graph G is derived from (5) as

$$V_G(b) = \max_{v \in G} \int_{s \in S} \alpha_v(s) b(s) ds \quad (7)$$

The expected total reward of a node v of G , α_v , is defined as

$$\alpha_v(s) = E\left(\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right) = r(s, a_v) + E\left(\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t)\right) \quad (8)$$

Note that the sum in (5) is replaced by an integral to handle the continuous-state space. We can evaluate the integration of (7) by performing MC simulations; namely, we sample many random states, called particles, from S with a probability of $b(s)$ and simulate their policy $\pi_{G,v}$ for all $v \in G$.

From (6) and (7), the optimal value function V^* can be calculated using value iteration (the Bellman recursion) as

$$V_G = HV_G = \max_{a \in A} \left\{ \int_{s \in S} R(s, a) b(s) ds + \gamma \sum_{o \in O} p(o|b, a) \max_{v \in G} \int_{s \in S} \alpha_v(s) b'(s) ds \right\} \quad (9)$$

Using the value function in (9), the MCVI algorithm finds an optimal policy by iteratively updating a policy graph G with the optimal actions and the resulting observations according to the following two steps.

1. **Sampling.** A belief tree T_R , consisting of belief nodes reachable from b_0 , is constructed. Initially T_R has only one root node b_0 as illustrated in the first step of Figure 1. To expand T_R , belief particles are generated from b_0 in the second step of Figure 1 and the value function $V_G(b_0)$ is evaluated using (9). Next we perform particle filtering on the generated particles in the third step of Figure 1. Then, the best node, identified through particle filtering, is added to T_R in the fourth step of Figure 1. Adding a new belief node continues until the depth of T_R is sufficient or the difference of bounds on $V^*(b_0)$ (the optimal value of (7)) is sufficiently reduced.
2. **Backup.** Here T_R is traversed from b_0 toward the terminal node to evaluate $V^*(b_0)$ by performing value iteration for every belief node. When each node is traversed, particles are generated randomly biased toward the current belief node in the sixth step of Figure 1 and the generated particles are simulated for evaluating the second integral of (9) in the seventh step of Figure 1. To update the policy graph, a new node is added to the existing policy graph (Figure 2, left), and the best action and observations are obtained by simulating all possible combinations of actions and observations from the newly added node. The policy graph G is updated to G' for each value iteration based on the action and observations found in the eighth step of Figure 1 and also illustrated in Figure 2; this process is called MC backup.

The sampling/backup steps are iterated until the difference between the upper and lower bounds on $V^*(b_0)$ drops below a predefined threshold. Note that seven loops are involved in the entire MCVI algorithm (Figure 1). The main objective of our parallel algorithm is to identify all possible loops and execute them in parallel by taking advantage of the massive number of cores available on GPUs or hybrid machines.

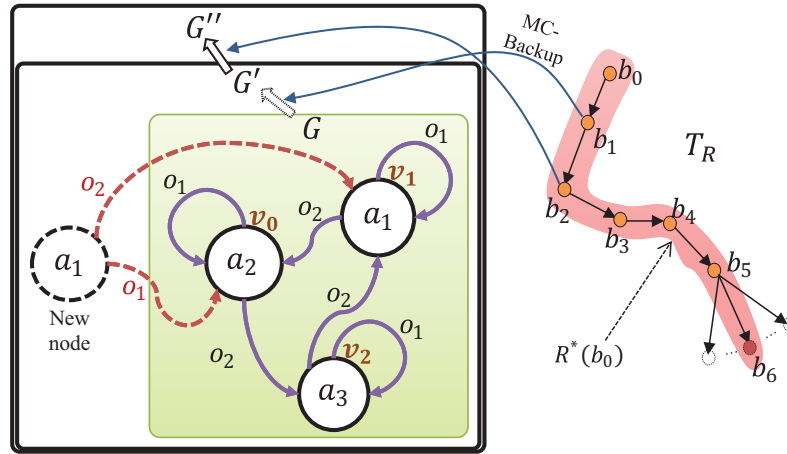


Figure 2. Policy graph update. These figures illustrate sampling-backup iteration. The right figure is the sampled belief tree T_R and the left figure is the policy graph. To sample a new belief from b_5 , we first generate several beliefs as shown in the second step of Figure 1. Then, we perform particle filtering for all beliefs and identify the best belief in the third step of Figure 1. In this case, b_6 is selected and added as a child node of b_5 in the fourth step of Figure 1. To update the policy graph, we add a new node to the existing policy graph in the sixth step of Figure 1 and perform MC simulations on b_0 with all possible combinations of actions and observations from the new node in the seventh step of Figure 1. After completing the MC simulations on b_0 , we obtain the best action a_1 and observations o_1 and o_2 . Then the new node is updated to a_1 and linked to other nodes of the policy graph with o_1 and o_2 in the eighth step of Figure 1. After all MC simulations for all nodes of T_R are complete, return to the second step of Figure 1 and iteratively repeat the sampling and backup steps.

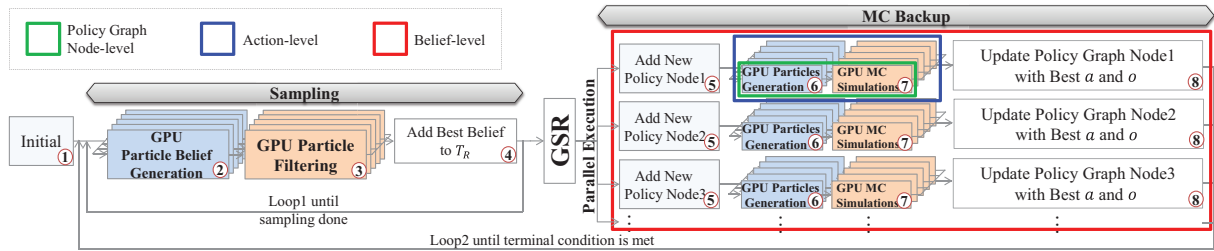


Fig. 3. The process of gPOMDP. The step numbers are matched with those in Figure 1, and the multiply layered boxes of each step indicate parallel execution. The green, blue, and red boxes indicate policy graph node-, action-, and belief-level parallelisms, respectively. Note that, in gPOMDP, five loops are eliminated from the process shown in Figure 1.

4. gPOMDP: GPU multi-level POMDP-based parallel algorithm

Among the current POMDP algorithms, the MCVI algorithm (Bai et al., 2010) is suitable for massive parallelism because (1) value iteration using many particles can be executed in parallel and (2) the required memory footprint to operate the algorithm is relatively small since it does not maintain the history of belief states. Thus, the goal of our algorithm is to parallelize the MCVI algorithm using GPUs. In particular, since the MC backup step explained in Section 3.2 consumes 99% of the total running time in MCVI, our objective is to parallelize MC backup using massively many parallel GPU threads.

The sampling step and the MC backup step shown in Figure 1 are successfully parallelized by executing multiple samplings and MC backups (from the second to third steps and from the fifth to seventh steps of Figure 1) in order to achieve the maximum performance enhancement.

Moreover, before starting the GPU-based MC simulation step (the fifth to eighth steps of Figure 3), our GSR method effectively schedules CUDA blocks for minimizing workload variances; the GPU idle time is reduced by half compared with that not using the GSR method.

Figure 3 illustrates the process of the gPOMDP algorithm utilizing the GSR workload distribution method (the GSR method will be explained in Section 5). The step numbers in Figure 3 are matched with those in Figure 1 and the multi-layered boxes in Figure 3 indicate parallel execution. To parallelize the MC backup step, we employ three different levels of parallelism: policy graph node (green box), action (blue box), and belief levels (red box).

4.1. CUDA architecture

The implementation choice of our GPU parallelization is CUDA (NVIDIA, 2012), mainly because CUDA is optimized for both high- and low-end GPUs, and low-level

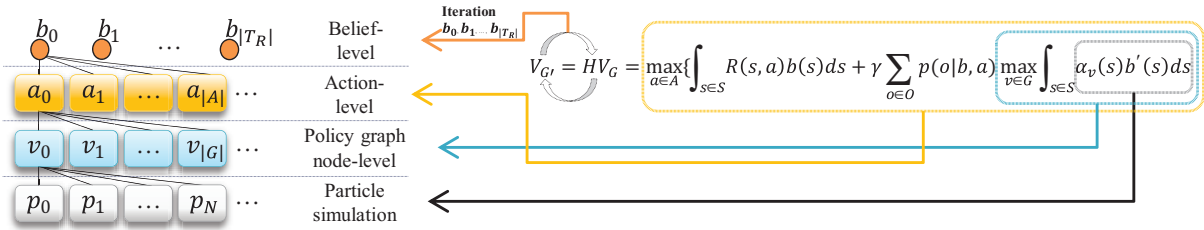


Fig. 4. Backup hierarchy. The backup for a belief b_i is performed in a hierarchical manner. Levels indicate the depth of hierarchy of the MC backup process, and we can execute more concurrent simulations by increasing the level. Here $N \times |G| \times |A|$ simulations are needed for one belief node b_i to add a new node and its edges to G .

optimization is available using steps such as handling the cache or avoiding memory access conflicts. The CUDA is a parallel GPU computing platform and programming model developed by NVIDIA. The parallel tasks that can be executed by CUDA are called *kernels*. Thousands of threads in the form of a grid of blocks can run the same kernel in parallel. Moreover, a block is a batch of threads that can cooperate with each other via shared memory and execution synchronization. The following aspects should be considered to maximize the performance of CUDA-based parallel computation (NVIDIA, 2012).

- One should carefully choose the task unit for a kernel so that many threads can be executed in a single kernel call in order to produce the best performance.
- One should interleave the CPU and GPU computation as much as possible in order to reduce the idle time of the GPU.
- The shared memory in CUDA should be maximally used for the threads within the same block in order to maximize the memory throughput.

4.2. Policy graph node-level parallelism

Equation (9) is the main equation of the MC backup process. We can further dissect this equation into three nested levels of computations, as illustrated in Figure 4: the policy graph node level, the action level, and the belief level.

The easiest way to parallelize (9) is at the policy graph node-level. More specifically, for each node v_i , a block of threads is assigned. Moreover, each particle p_i , mapped to a single thread, can share the high-performance CUDA shared memory that enables the rapid integration calculation. Figure 5 shows the assignment of blocks and threads to achieve policy graph node-level parallelism. In the figure, cb_i is a CUDA block and ct_i is a CUDA thread. N particles (N CUDA threads) are simulated for each policy graph node v_i so that $|G|$ policy graph nodes (i.e. $|G|$ CUDA blocks) are executed.

Algorithm 1 is the pseudo-code for a CUDA kernel to perform MC simulations at the policy graph node-level. We launch $|G| \times N$ threads where $|G|$ is the size of the current policy graph, and N is the size of particles for MC simulation. Moreover, $|G|$ is equal to the number of

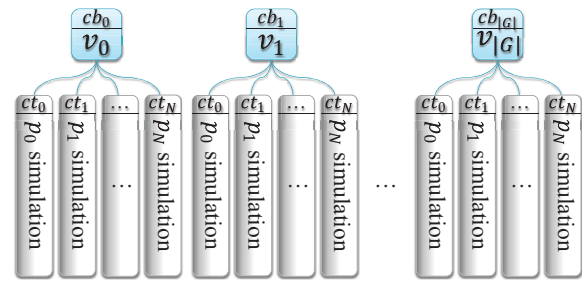


Fig. 5. Policy graph node-level parallelism. All MC simulations of all policy graph nodes for one action are issued concurrently at the policy graph node level.

blocks, and N is the number of threads for each block; bID and tID are the block and thread indices, respectively. A particle for the p_i simulation shown in Figure 5 is obtained from $GetParticle(b, tID)$ in line 1, and the simulation is performed in lines 3–13 using a particle p generated from a belief b on the belief tree T_R . Here $GetParticle(b, tID)$ returns a randomly generated particle based on b and tID . During MC simulations, the reward is accumulated to a shared memory location $reward[tID]$. In line 16, we integrate all $reward[i]$ and store it at the global memory $result[bID]$, accessible from the CPU to update the policy graph. Note that $reward[i]$ is located in the high-speed CUDA shared memory; the integration of all simulation values in $IntegrateRewardsOfAllThreads()$ can be performed extremely quickly using the parallel reduction method (NVIDIA, 2012). After the kernel terminates, we obtain the integration value of v_i (i.e. MC simulation results) from $result[i]$. The actual update on the policy graph occurs on the CPU side since this operation requires random referencing to the policy graph nodes that may not be well mapped to GPUs. In addition, this operation is not computationally heavy.

The result in Section 7 shows that Algorithm 1 produces a ten-fold greater performance than the CPU. However, GPU thread utilization can be very poor when we copy the MC simulation results from the GPU memory to the CPU memory. We need $|A|$ memory copies from the GPU to the CPU for MC backup on belief b since the number of required simulations is $N \times |G| \times |A|$ and Algorithm 1 executes $N \times |G|$ simulations for each CUDA kernel

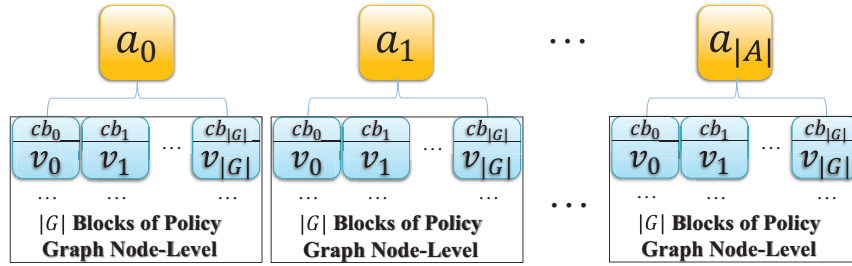


Fig. 6. Action-level parallelism. All MC simulations of all policy graph nodes for all actions are issued concurrently at the action level. Each rectangle belonging to node a_i contains blocks for performing policy graph node-level parallelism, as shown in Figure 5.

Algorithm 1. Policy Graph Node-level Kernel.

Input: tID: Thread ID(N), bID: Block ID($\text{---}G\text{---}$), b: Belief,
G: Policy Graph, reward[N]: CUDA shared memory
Output: result[bID]: MC simulation results

```

1:  $p = \text{GetParticle}(b, \text{tID});$ 
2:  $v = \text{PolicyGraphNode}[\text{bID}];$ 
3: for  $i = 0$  to  $\text{simulationLength}$  do
4:   if  $p == \text{goal}$  then
5:      $\text{break};$ 
6:   end if
7:    $a = \text{GetActionFromGraph}(v);$ 
8:    $\text{reward}[\text{tID}] += \text{discount} \times \text{GetReward}(p, a);$ 
9:    $o = \text{GetObservation}(p, a);$ 
10:   $v = \text{GetNextNodeFromGraph}(v, o);$ 
11:   $p = \text{GetNextState}(p, a);$ 
12:   $\text{discount} = \text{discount} \times \text{discountFactor};$ 
13: end for
14:  $\text{syncThread}();$  {sync all threads within a block}
15: if  $\text{tID} == 0$  then {true only once per block}
16:    $\text{result}[\text{bID}] = \text{IntegrateRewardsOfAllThreads}();$ 
17: end if

```

execution call. In the next section, we mitigate this issue of a high number of memory copies by increasing the parallelism level to the action-level.

4.3. Action-level parallelism

To reduce the number of memory copies from the GPU to the CPU, we should spawn as many blocks and threads as possible in a single kernel execution. We can achieve this objective by running MC simulation in parallel for all $a_i \in A$ and all $v_j \in G$ since each simulation result of a_i is independent; thus, we call the same simulation kernel (Algorithm 1) for all $a_i \in A$ and all $v_j \in G$. One tricky aspect that we should consider while performing action-level MC simulation is that we need to ensure that the indexing mechanism for each particle p_i does not overflow to access the policy graph G as follows.

1. Replace $\text{PolicyGraphNode}[\text{bid}]$ with $\text{PolicyGraphNode}[\text{bid} \% |A|]$ in Algorithm 1.
2. Execute Algorithm 1 in parallel for $|G| \times |A|$ blocks each with N threads.

3. The result of MC simulation for v_j of a_i is $\text{result}[i \times |A| + j]$.

Figure 6 illustrates action-level parallelism. Note that each action a_i includes all corresponding policy graph node-level simulations. As a result, we can execute $N \times |G| \times |A|$ simulations (CUDA threads) for each kernel execution call. The performance result in Section 7 indicates that the action-level parallelism doubles the performance level of policy graph node-level parallelism.

4.4. Belief-level parallelism

In general, it is difficult to achieve the belief-level parallelism (i.e. running Algorithm 1 concurrently for all b_i) because the policy graph needs to be updated sequentially from b_i to b_{i+1} . However, value iteration can still increase the quality of the policy graph when concurrently performing MC backup on several beliefs even though it sacrifices the improvement rates to achieve the optimal value. The following theorem says that the backup of belief b_{i+1} without the backup of b_i can still increase the quality of the policy graph.

Theorem 1. Given a set of beliefs b_i , $0 \leq i < n$ and $b \in R^*(b_0)$, the backup of b_k for an arbitrary k , $1 \leq k < n$ can improve the policy graph (Bai et al., 2010; Hansen and Zhou, 2003; Lim et al., 2011).

However, if we run concurrent MC simulations in an arbitrary sequence (say, b_0, b_k, \dots), the improvement rate of the policy graph update can be poor if b_k is not reachable from b_0 via some action $a \in A$. In this case, we may need more sampling/backup iterations to achieve the same improvement rate of the original policy graph update in the MCVI algorithm. However, Section 7 shows that the performance gain by applying belief-level parallelism compensates for the loss in improvement rate. With the belief-level parallelism, each belief b_i includes all corresponding simulations including all those of the action and policy graph node levels, as shown in Figure 7.

5. GPU workload distribution

As the parallel level in our algorithm increases from the policy graph node to the belief level, more blocks and

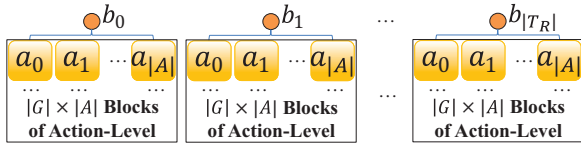


Fig. 7. Belief-level parallelism. All MC simulations of all policy graph nodes for all actions that belong to all belief tree nodes are issued concurrently at the belief level. Each rectangle belonging to node b_i contains blocks for the action-level parallelism in Figure 6.

threads are needed in a single kernel execution; for instance, the total number of threads required for belief-level parallelism is $N \times |G| \times |A| \times |T_R|$. As the number of kernel executions increases, so does the time required to retrieve and update the policy graph from the CPU to the GPU. We address this problem by using CPU/GPU interleaving and workload scheduling as follows.

A CUDA stream is a sequence of GPU operations that execute in issue-order on the GPU. A stream enables a compute/data overlap. For example, a data transfer for stream 1 and kernel execution for stream 2 can be executed concurrently as long as there is no memory conflict between the streams.

We group thread blocks with the same workload into a working set; here, the workload is defined as the time needed to synchronize all threads. For example, as illustrated in Figure 8, the maximum simulation length of v_0 is longer than that of v_1 ; thus, v_1 waits until v_0 is finished, and the threads assigned to v_1 become idle. Conversely, if we run v_0 and v_2 concurrently with similar simulation times, the GPU idle time can be significantly reduced, as shown in Figure 9. The number of working sets is predefined and can be empirically obtained by experimentations.

We perform three steps to achieve the above optimizations: gathering, scheduling, and running, as illustrated in Figure 10.

Gathering G . In this step, we gather all blocks and store them in a three-dimensional array with an index to the node v , action a and belief b . The array will be used for returning the MC simulation result of each block to the corresponding action and belief. The gathering step continues until all blocks in belief tree T_R are gathered. It is possible that the gathered number of blocks could exceed the memory capacity of the GPU. If this happens, we stop gathering the blocks and proceed to the next step.

Scheduling S . The scheduling step determines the workload of each block, obtained from the gathering step, and groups them into a small number of working sets. To determine the workload for a block v , we rely on simulation coherence. More specifically, after MC simulations are finished for v , we record the average simulation length of a thread in a table and use it for the workload of v for the next simulation. The workload table is updated at every sampling-backup iteration. The size of a working set is the number of gathered blocks divided by the number of

working sets (eight in our implementation). The actual scheduling is performed by redistributing all the blocks while maintaining the workload variance for each working set at a minimum.

Running R . The running step operates on working sets with the compute/data interleaving technique as follows. First, based on the number of working sets, we create a number of CUDA streams and assign one CUDA stream to each working set. Next, we perform an execution/retrieval sequence for each CUDA stream. Since execution and retrieval can be performed asynchronously, we retrieve the results from a completed working set while executing the next working set. The GSR is performed before MC backup, as shown in Figure 3.

6. hPOMDP: CPU–GPU POMDP-based hybrid algorithm

In Section 5, we showed that workload balancing based on simulation length can enhance the performance of the POMDP-based algorithm. However, two additional issues must be addressed to further enhance the POMDP performance.

- The simulation time increases as the complexity of planning increases due to an increasing number of POMDP state variables and increasing dimensions of the problem space.
- Increasing the number of particles also increases the simulation time. Moreover, the particle generation time of the GPU is considerable in this case.

gPOMDP can quickly generate random particles on the GPU and store them in GPU memory when the number of particles is small, e.g. fewer than 7000 particles in less than 16 ms. However, there is a cost associated with particle generation in the GPU, i.e. particle simulation cannot begin before particle generation is complete. Thus, we have observed that the performance enhancement ratio starts to degrade (e.g. to less than 50) when gPOMDP is used for complicated benchmarks, for instance, aircraft benchmark discussed in Section 7. The next section describes a new CPU–GPU hybrid method that can efficiently solve complicated POMDP problems.

6.1. CPU–GPU parallel execution

Our CPU–GPU hybrid method (i.e. hPOMDP) generates particles in the CPU and simulates them in the GPU using the GSR method. Since particle simulation is performed in the GPU, hPOMDP needs to upload particles from the CPU to the GPU. The pseudo code for hPOMDP is given in Algorithm 2.

Unlike gPOMDP with the GSR method, we do not generate particles in the GPU (line 4) but in the CPU, uploading the results back to the GPU (lines 7–8), while the GPU

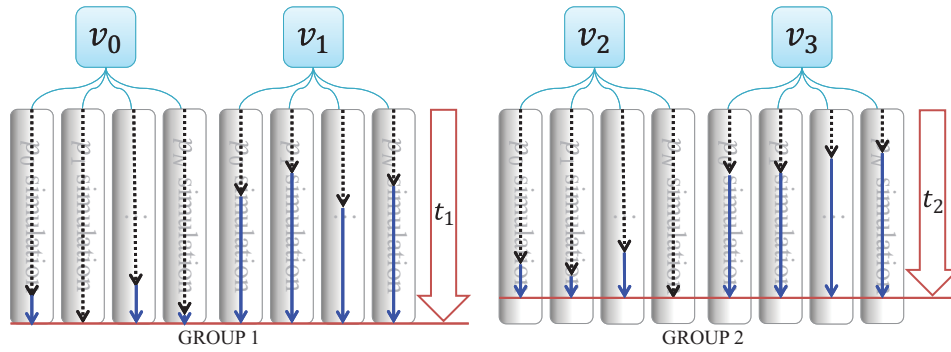


Fig. 8. Synchronization problem in parallel algorithms. The black arrow indicates the simulation length for each thread, and the blue arrow indicates the waiting length for synchronization. In this case, the threads are grouped into GROUP1 and GROUP2, and the simulations of each group are executed in parallel. The red arrow is the total simulation time of each group, and the total time for all groups is $t_1 + t_2$. Many CUDA cores assigned to each simulation become idle while waiting for synchronization.

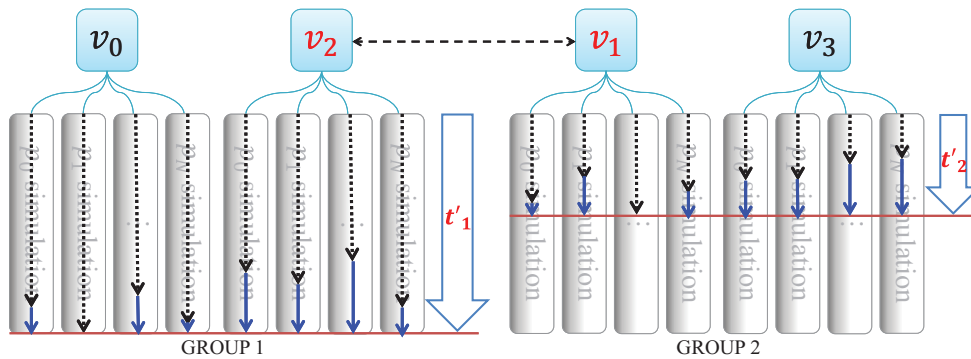


Fig. 9. Performance gain by workload balancing. After workload balancing, the waiting time for synchronization (the sum of blue arrow lengths) is reduced by a factor of two. The total time for all groups is $t'_1 + t'_2$ and is reduced by $\frac{2}{3}$ compared with that in Figure 8.

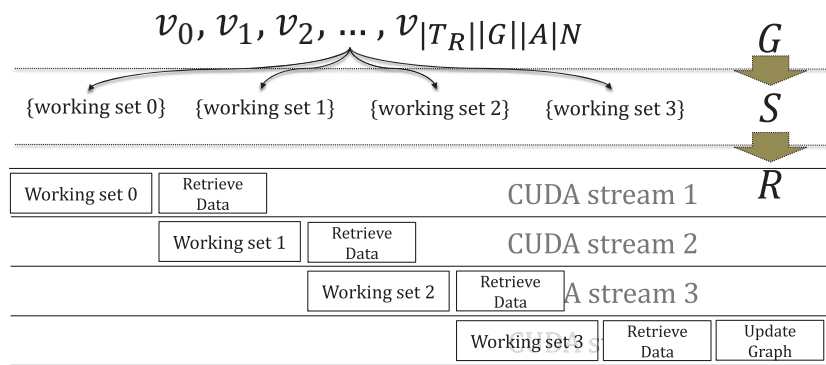


Fig. 10. The sequence of gathering, scheduling and running. Step G gathers all blocks that can be performed concurrently. Step S distributes the gathered blocks into several working sets. Step R performs kernel execution by maximizing the computation/memory overlap.

is performing simulations based on the previously uploaded particles (line 4). The pipeline of hPOMDP is shown in Figure 11. The total computation time does not increase even though the CPU generates particles because we avoid

such latency by using CPU/GPU overlap techniques combined with compute/data overlap techniques of the CUDA framework; in other words, the second and third steps overlap with each other, as do the sixth and seventh steps.

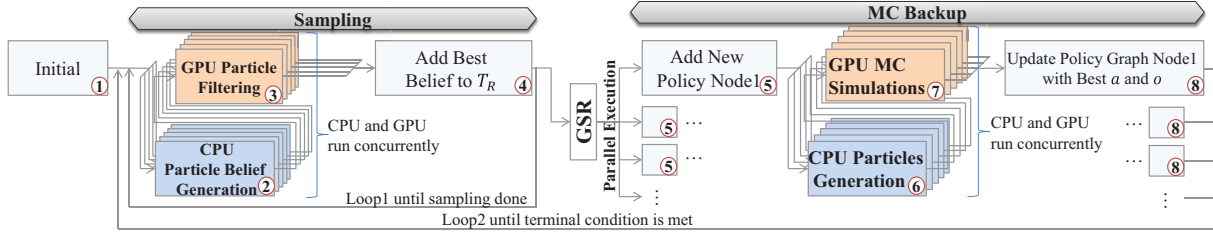


Fig. 11. The process of hPOMDP. The step numbers are matched with those in Figure 3. hPOMDP generates particles on the CPU side in order to maximize the utilization rate of the GPU. When the number of particles is less than 7000 and the complexity of the benchmark is low, GPU particle generation can compensate for CPU–GPU data transfer; however, it is more efficient to generate particles in the CPU when the number of particles is greater than 7000 and the complexity of the benchmark is high because the particle simulation time exceeds the particle generation and upload time.

Algorithm 2. hPOMDP stream execution.

Input: numberOfWorkingSets, workingSet
Output: result[i]; MC simulation results of all blocks

```

1: for i=0; to numberOfWorkingSets do
2:   if i < numberOfWorkingSets then
3:     {This function uses particles generated in line 7.}
4:     StreamExecutionWithoutParticleGen(workingSet[i]);
5:   end if
6:   if i + 1 < numberOfWorkingSets then
7:     CPUParticleGeneration();
8:     UploadParticle(workingSet[i + 1]);
9:   end if
10:  if i > 0 then
11:    RetrieveData(result[i - 1], workingSet[i - 1]);
12:  end if
13: end for

```

To demonstrate the efficiency of the hPOMDP method, we profile the times of hPOMDP in Table 1 for the aircraft benchmark (Figure 12(d)). Table 1 shows the times of gPOMDP and hPOMDP for a single policy graph node with 9000 particles. Here, we can see that the simulation time of the aircraft benchmark exceeds the sum of particle generation and upload times of hPOMDP. As a result, if we can parallelize particle generation and upload in the CPU with particle simulation in the GPU, we can avoid latency on the CPU side, and the total time of hPOMDP can be reduced to 439 ms compared with the 533 ms of gPOMDP.

However, since hPOMDP does not always produce the best result for a POMDP problem, in the next section, we introduce a gPOMDP/hPOMDP transition formula.

6.2. gPOMDP/hPOMDP transition formula

Now we introduce a transition formula that can determine which of gPOMDP or hPOMDP works best for a given benchmark by comparing the generation and upload times of the CPU with the simulation time of the GPU.

We first calculate how much time in milliseconds the CPU spends generating one random number (C_{Gen}) by

Table 1. Particle generation, upload, simulation, and total times for MC simulations with 9000 particles in milliseconds. This table shows the profiling results for completing MC simulations of one node with 9000 particles using gPOMDP and hPOMDP.

Task	Generation	Upload	Simulation	Total
Aircraft (gPOMDP)	94	N/A	439	533
Aircraft (hPOMDP)	108	232	439	439

generating many particles and averaging the generation times. Since the number of particles (N) and POMDP state variables (NS) and the size of particle in bytes ($PSize$) are determined a priori, we can calculate the sum of particle generation and upload times of the CPU as follows:

$$T_{prep} = N \times NS \times C_{Gen} + \frac{N \times PSize}{Band} \quad (10)$$

where $N \times PSize$ is the memory size of N particles, and $Band$ is the CPU-GPU memory bandwidth. In other words, the first term of (10) is the particle generation time of the CPU, and the second term is the upload time from the CPU to the GPU.

To estimate the simulation time of N particles in the GPU, we suppose the worst case scenario. In other words, we measure the time, G_{Simul} , to complete N particle simulations with the maximum simulation length in the GPU. To obtain G_{Simul} , we first create a single node policy graph with a random action and generate N random particles for this node. We also add loop edges for all observations to the node. Then, we execute N CUDA threads (i.e. N simulations) for the policy graph with the maximum simulation length and measure the total time.

Using (10) and G_{Simul} , we choose hPOMDP to run the given POMDP problem if $T_{prep} < G_{Simul}$; otherwise, we use gPOMDP. Table 2 shows T_{prep} and G_{Simul} on different benchmarks. In case of the aircraft benchmark, T_{prep} is 325 ms (< 452 ms) and thus we use hPOMDP.

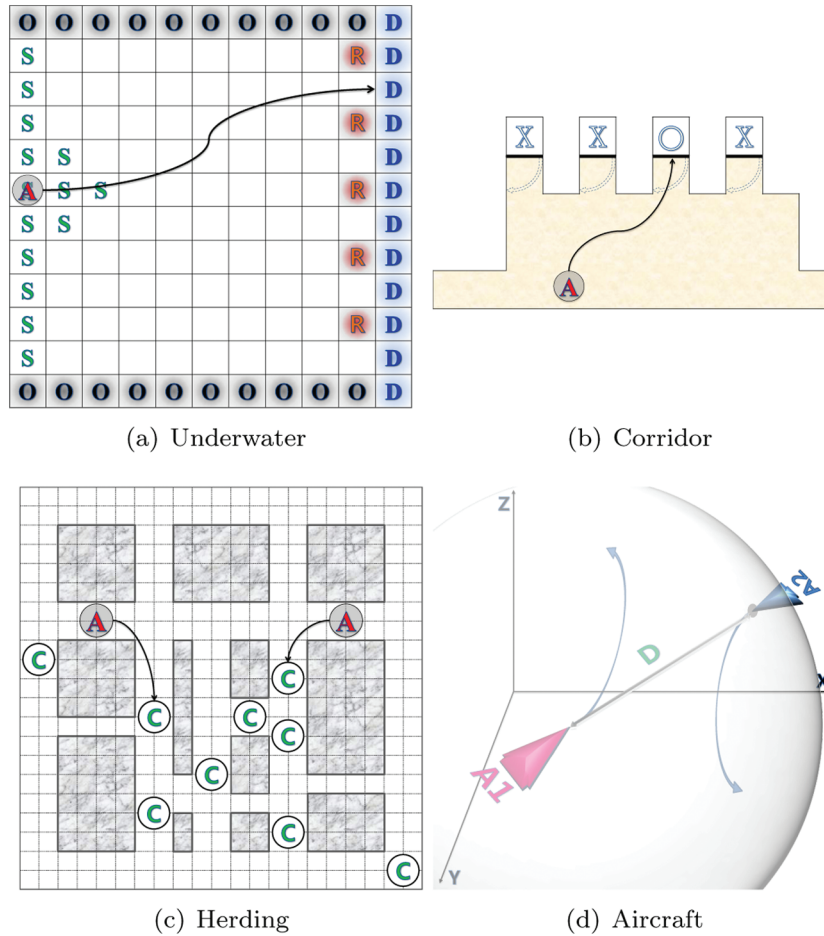


Fig. 12. Various POMDP benchmarks.

7. Results and discussion

In this section, we show our implementation results and discuss them.

7.1. Performance benchmark

We tested our algorithm on a PC running Windows 7, equipped with an Intel i7 2.67 GHz CPU with 3 GB memory and with NVIDIA Geforce 680 GPU 2 GB 1024 CUDA cores. We applied a CUDA as a parallel framework and tested four robot motion planning tasks, namely underwater navigation, corridor, collaborative search and capture (i.e. herding), and unmanned aircraft collision avoidance (i.e., aircraft). Even though these benchmarks are relatively simplistic, given the high complexity of solving continuous POMDP problems, they serve as standard benchmarks for POMDP-based motion planners (Bai et al., 2011; Kurniawati et al., 2008; Lim et al., 2011; Porta et al., 2006). In our current implementations, we used only one CPU core. The aircraft benchmark operates in three-dimensional space while the others operate in two-dimensional space.

Underwater navigation (Figure 12(a)). The agent receives a positive reward when it reaches the destination “D” while

it receives the negative reward when it encounters the rock “R” or takes an action for moving. A robot starts from an initial point among “S” marks and tries to reach the destination “D” by taking actions for moving. The robot can localize itself only at the bottom or top of the map at “O” marks.

Corridor (Figure 12(b)). The robot navigates a corridor with four doors. The goal is to enter the second door from the right “O” mark. The robot receives a positive reward when it enters the target door while receiving a negative reward when entering wrong door “X” mark or moving beyond both ends of the corridor.

Herding (Figure 12(c)). Two robots track 12 escaped crocodiles in a 21×21 grid map with obstacles. The robots have to collaborate with each other to catch the crocodiles and are centrally controlled. The robots receive a positive reward when they successfully capture a crocodile.

Aircraft (Figure 12(d)). Two unmanned aircraft try to avoid collision by maintaining a minimum safety distance from each other. Each aircraft receives a negative reward when the distance between them falls below the safety distance. This benchmark is defined in 3D continuous space, and the movement of each aircraft is calculated using 12 parameters including the 3D position (x, y, z) , heading angle

Table 2. Values of T_{prep} and G_{Simul} on different benchmarks. The first column denotes different benchmarks and columns 2–5 are the terms in (10). In all cases, $Band$ is 0.7 GB/s. These experiments are obtained by simulation for worst-case scenarios.

Benchmark	N	NS	$PSize$	C_{Gen}	T_{prep}	G_{Simul}
Underwater	1000	2	16	0.007	14	8
Corridor	1000	2	16	0.007	14	7
Herding	1000	6	48	0.006	36	22
Aircraft	9000	12	96	0.002	325	452

Table 3. Performance comparisons. The first column indicates the benchmarks. gPOMDP is applied to all benchmarks except the aircraft benchmark, which uses hPOMDP. Columns 2–6 are the total times in seconds to reach the target using the policy graph node-level, action-level, belief-level without GSR, belief-level with GSR, and the APPL library. The last column shows speed-ups of gPOMDP or hPOMDP compared with APPL. For hPOMDP, we adopted the optimal parallelism in the GPU, the GSR method.

Benchmark	Policy	Action	Belief	GSR	APPL	Speed-ups
Underwater	24	12	8	4	300	75
Corridor	34	18	15	9	726	81
Herding	804	297	204	139	12,000	86
Aircraft	N/A	N/A	N/A	1028	101,806	99

θ , and horizontal and vertical speed (u , v) of two aircrafts. This benchmark has a higher complexity than the other 3 benchmarks because it works in 3D space rather than 2D space and uses 12 parameters, while the others use no more than 4 parameters.

For all experiments, we set the number of working sets to eight, which showed the best performance in our pre-experiments. We ran the trial 100 times for each benchmark and averaged the results due to the random nature of the MCVI algorithm. For comparison, we used the CPU-based public MCVI library; APPL¹ (Bai et al., 2010).

The performance result of APPL is different from that of Bai et al. (2010) since we do not use further optimization parameters such as OpenMP and caching. To compare the performance, we use the same target total reward or failure ratio for both APPL and our algorithm and measured the total times needed to reach the target value. Table 3 shows our experimental results. Our gPOMDP algorithm outperforms the APPL, CPU-based MCVI algorithm, by 75–99 times.

7.2. Performance analysis

First, we evaluated the simulation length variances of each policy graph node v_i before and after the workload balancing in Figures 13(a) and (b). The figure shows that the variance is much reduced after the workload balancing. The rate of idle GPU cores depends on the variance.

Figure 14 shows that our GPU-based method can perform more simulations per second than APPL for the underwater navigation benchmark, and that the number of simulations per second remains high regardless of the problem size (i.e. the sizes of the belief tree $|T_R|$ and the policy graph $|G|$). In the graph, the X -axis denotes the problem size ($|T_R| \times |G|$) and the Y -axis denotes the number of

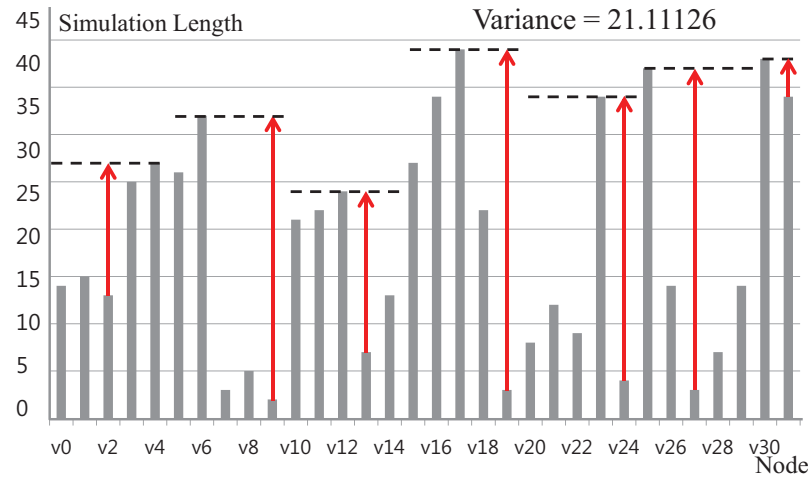
simulations per second. This graph also explains why our parallel POMDP solver significantly outperforms APPL. Also, the constant graph implies that our method is applicable to more challenging benchmarks such as the aircraft benchmark, since the number of simulations per second is not much affected by the problem size.

In Figure 15, we also show how much time should be spent for different levels of parallelism in our algorithm to achieve the same target reward value. The benchmarking scenario is also underwater navigation. The total computation time using the action level is reduced to half that of the policy graph node level, mainly because the frequency of retrieving the integration value of v has been reduced. The arithmetic intensity further increases when using belief-level parallelism and the performance was enhanced by 150% compared with action-level parallelism. Finally, the belief level with GSR method achieves the best performance using the workload balancing and compute/data parallelism. The total timing was reduced to 50% of that without GSR.

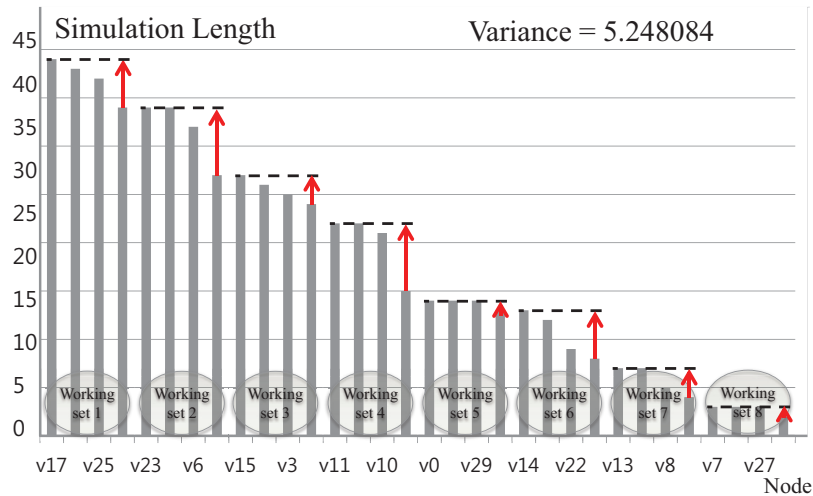
Figure 16 shows the performance comparison of gPOMDP and hPOMDP on different numbers of particles. The aircraft benchmark is used for this comparison. We can see that the performance of gPOMDP is better than that of hPOMDP when the number of particles is less than 7000. The maximum performance of gPOMDP is 45,000 simulations per second. Note that the capacity of hPOMDP is 76,000 simulations per second in the case of complex benchmarks.

8. Conclusion

In this paper, we have presented novel algorithms using the GPU (gPOMDP) and a hybrid of the GPU and the CPU



(a) Before



(b) After

Figure 13. Comparison of simulation length (a) before and (b) after workload balancing. (a) The Y-axis of the graph is the average simulation length (i.e. average simulation steps) of each policy graph node v_i . Without GSR, the CUDA framework groups these nodes according to input order (in this case, the group size is five) and executes the grouped nodes in parallel. Note that many nodes wait a long time to synchronize with others (the red arrow is waiting time). In this case, the variance of the simulation lengths is 21.11126. (b) After workload balancing, the execution sequence of nodes is determined according to the simulation length. Since the total number of nodes is 32, and we set the number of working sets to 8, each working set executes four nodes in parallel. After workload balancing, the variance of simulation lengths is reduced to 5.248084.

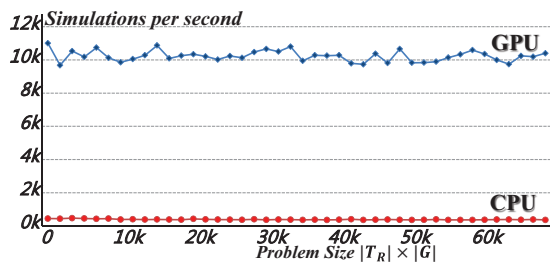


Figure 14. Number of simulations per second for underwater navigation. Our GPU-based method (red) performs 10,000 simulations per second on average irrespective of the problem size, while CPU-based APPL does only 100–200 simulations in the same amount of time.

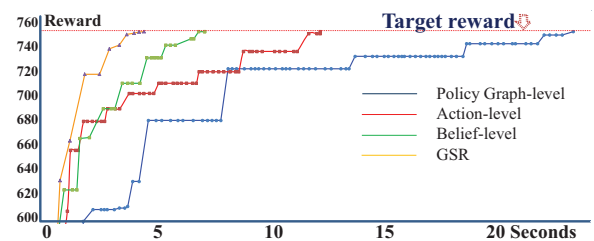


Fig. 15. Performance comparison using different levels of parallelism.

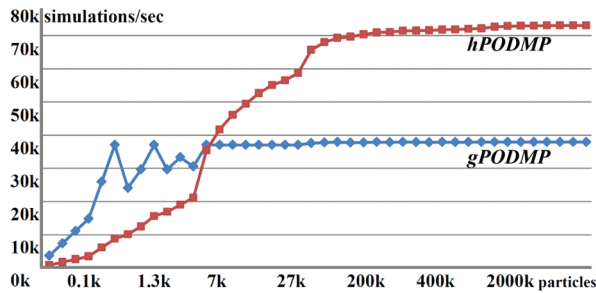


Fig. 16. The performance comparison of gPOMDP and hPOMDP. We measured the number of simulations per second while increasing the number of particles. The blue line denotes gPOMDP and the red line denotes hPOMDP. A total of 1024 GPU cores were used for both gPOMDP and hPOMDP. hPOMDP performs better when more than 7000 particles are present.

(hPOMDP) to solve continuous-state POMDP problems. Our algorithm uses multi-level parallelism to perform MC backup, which is a bottleneck in the existing CPU-based MCVI algorithm. Our algorithms outperform the existing CPU-based algorithm by a factor of 75–99 depending on the chosen benchmark. There are a few limitations associated with our algorithm. When many CUDA cores try to concurrently access the policy graph, the utilization rate may decrease because we maintain a single policy graph in GPU memory, which can cause memory conflicts. As a remedy to this problem, we may divide the policy graph into small independent graphs and perform the simulations independently of each graph. Moreover, to achieve the highest level of parallelism, we sacrifice the improvement rates in the original MCVI algorithm. Dense sampling of particle beliefs may address this issue. Finally, our algorithm is limited by the capacity of the GPU, for instance, the GPU memory size. One possible avenue to solve this limitation is the use of a GPU cluster, e.g. NVIDIA GPU GRID (NVIDIA, 2014), since our algorithm is scalable with respect to the number of GPU cores and memory size. In the future, we plan to consider multiple CPU cores and GPU cluster to solve more complicated POMDP benchmarks such as those for articulated robots and to apply our method to online planning algorithms such as those of Ross et al. (2008) and Chang et al. (2004).

Acknowledgements

We thank David Hsu and Haoyu Bai for providing POMDP benchmarks and useful discussions.

Funding

This work was supported in part by NRF in Korea (grant number 2014K1A3A1A17073365) and MCST/KOCCA in the CT R&D program 2014 (grant number R2014060011).

Note

1. See <http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/>

References

- Bai HY, Hsu D, Lee WS and Ngo VA (2010) Monte Carlo value iteration for continuous-state POMDPs. In: *Algorithmic foundations of robotics IX—proceedings of international workshop on the algorithmic foundations of robotics (WAFR)*. New York: Springer.
- Bai HY, Hsu D, Kochenderfer MJ and Lee WS (2011) Unmanned aircraft collision avoidance using continuous-state POMDPs. In: *Proceedings of robotics: science and systems*.
- Bellman RE (1957) *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Berg JVD, Abbeel P and Goldberg K (2010) LQG-MP: optimized path planning for robots with motion uncertainty and imperfect state information. In: *Proceedings of robotics: science and systems*, Zaragoza, Spain.
- Berg JVD, Patil S and Alterovitz R (2011) Motion planning under uncertainty using differential dynamic programming in belief space. In: *Proceedings of international symposium of robotics research (ISRR)*.
- Berg JVD, Patil S and Alterovitz R (2012a) Efficient approximate value iteration for continuous Gaussian POMDPs. In: *Proceedings of AAAI conference on artificial intelligence*.
- Berg JVD, Patil S and Alterovitz R (2012b) Motion planning under uncertainty using iterative local optimization in belief space. *The International Journal of Robotics Research* 31(11): 1263–1278.
- Brendan B and Oliver B (2007) Synthesis of hierarchical finite-state controllers for POMDPs. In: *2007 IEEE international conference on robotics and automation*.
- Brooks A, Makarendo A, Williams S and Durrant-Whyte H (2006) Parametric POMDPs for planning in continuous state spaces. *Robotics and Autonomous Systems* 54(11): 887–897.
- Brunskill E, Kaelbling L, Lozano-Perez T and Roy N (2008) Continuous-state POMDPs with hybrid dynamics. In: *International symposium on artificial intelligence and mathematics*.
- Cassandra AR, Kaelbling LP and Kurien JA (1996) Acting under uncertainty: discrete Bayesian models for mobile robot navigation. In: *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, vol. 2, pp. 963–972.
- Chang HS, Givan R and Chong Edwin KP (2004) Parallel rollout for online solution of partially observable Markov decision processes. *Discrete Event Dynamic Systems* 14(3): 309–341.
- Guibas LJ, Hsu D, Kurniawati H and Rehman E (2008) Bounded uncertainty roadmaps for path planning. In: *Proceedings of the international workshop on the algorithmic foundations of robotics*.
- Hansen E and Zhou R (2003) Synthesis of hierarchical finite-state controllers for POMDPs. In: *International conference on automated planning and scheduling*.
- Hansen EA (1998) Solving POMDPs by searching in policy space. In: *Proceedings of AAAI conference on artificial intelligence*, pp. 211–219.
- Hauskrecht M (2000) Value function approximations for partially observable Markov decision processes. *Journal of Artificial Intelligence Research* 13: 33–95.
- Hoey J, Bertoldi AV, Poupart P and Mihailidis A (2007) Assisting persons with dementia during handwashing using a partially observable Markov decision process. In: *Proceedings of international conference on vision systems*.

- Hsu D, Lee WS and Rong N (2008) A point-based POMDP planner for target tracking. In: *Proceedings of IEEE international conference on robotics and automation*, pp. 2644–2650.
- Kaelbling LP, Littman ML and Cassandra AR (1998) Planning and acting in partially observable stochastic domains. *Artificial Intelligence* 101(1–2): 99–134.
- Kurniawati H, Du Y, Hsu D and Lee WS (2011) Motion planning under uncertainty for robotic tasks with long time horizons. *The International Journal of Robotics Research* 30(3): 308–323.
- Kurniawati H, Hsu D and Lee WS (2008) SARSOP: efficient point-based POMDP planning by approximating optimally reachable belief spaces. In: *Proceedings of robotics: science and systems*.
- Latombe JC (1991) *Robot Motion Planning*. Norwell, MA: Kluwer Academic Publishers.
- Lee TH and Kim YJ (2013) GPU-based motion planning under uncertainties using POMDP. In: *2013 IEEE international conference on robotics and automation (ICRA)*, pp. 4576–4581.
- Lim ZW, Hsu D and Lee WS (2011). Monte Carlo value iteration with macro-actions. In: *Advances in neural information processing systems (NIPS)*.
- Luebke D, Harris M, Krüger J, Purcell T, Govindaraju N, Buck I, et al. (2004) ‘GPGPU: general purpose computation on graphics hardware’. In: *SIGGRAPH’04: ACM SIGGRAPH 2004 Course Notes*.
- Madani O, Hanks S and Condon A (1999) On the undecidability of probabilistic planning and infinite-horizon partially observable Markov decision problems. In: *Proceedings of 16th national conference of the American Association for Artificial Intelligence*.
- Missiuro P and Roy N (2006) Adapting probabilistic roadmaps to handle uncertain maps. In: *IEEE international conference on robotics and automation*.
- NVIDIA (2012) Cuda programming guide 4.2. <http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation>.
- NVIDIA (2014) NVIDIA GRID. <http://www.nvidia.com/object/nvidia-grid.html>.
- Pan J and Manocha D (2010) GPU-based parallel collision detection for real-time motion planning. In: *Workshop on algorithmic foundation of robotics (WAFR)*.
- Pan J and Manocha D (2012) GPU-based parallel collision detection for fast motion planning. *The International Journal of Robotics Research* 31(2): 187–200.
- Papadimitriou CH and Tsitsiklis JN (1987) The complexity of Markov decision processes. *Mathematics of Operations Research* 12(3): 441–450.
- Park C, Pan J and Manocha D (2012) ITOMP: incremental trajectory optimization for real-time replanning in dynamic environments. In: *International conference on automated planning and scheduling (ICAPS)*.
- Patil S, Berg JVD and Alterovitz R (2011) Motion planning under uncertainty in highly deformable environments. In: *Proceedings of robotics: science and systems (RSS)*.
- Patil S, Berg JVD and Alterovitz R (2012) Estimating probability of collision for safe planning under Gaussian motion and sensing uncertainty. In: *Proceedings of the IEEE international conference on robotics and automation (ICRA)*.
- Pineau J, Gordon G and Thrun S (2003) Point-based value iteration: An anytime algorithm for POMDPs. In: *Proceedings of the conference on uncertainty in artificial intelligence*, pp. 477–484.
- Porta JM, Vlassis N, Spaan MTJ and Poupart P (2006) Point-based value iteration for continuous POMDPs. *Journal of Machine Learning Research* 7: 2329–2367.
- Prentice S and Roy N (2007) The belief roadmap: Efficient planning in linear pomdps by factoring the covariance. In: *International symposium on artificial intelligence and mathematics*.
- Ross S, Pineau J, Paquet S and Chaib-draa B (2008) Online planning algorithms for POMDPs. *Journal of Artificial Intelligence Research* 32(1): 663–704.
- Roy N (2003) *Finding Approximate POMDP solutions Through Belief Compression*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.
- Shani G (2010) Evaluating point-based POMDP solvers on multi-core machines. *IEEE Trans Syst Man Cybernet B* 40(4): 1062–1074.
- Shani G, Brafman RI and Shimony SE (2007) Forward search value iteration for POMDPs. In: *Proceedings of the international joint conference on artificial intelligence*.
- Shani G, Pineau J and Kaplow R (2012) A survey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems* 27(1): 1–51.
- Simmons R and Koenig S (1995) Probabilistic robot navigation in partially observable environments. In: *Proceedings of the international joint conference on artificial intelligence*, pp. 1080–1087.
- Smallwood RD and Sondik EJ (1973) The optimal control of partially observable Markov processes over a finite horizon. *Artificial Intelligence* 21(5): 1071–1088.
- Smith T and Simmons R (2005) Point-based POMDP algorithms: improved analysis and implementation. In: *Proceedings of the conference on uncertainty in artificial intelligence*.
- Sondik EJ (1971) *The optimal control of partially observable Markov processes*. PhD thesis, Stanford University.
- Spann MTJ and Vlassis N (2004) A point-based POMDP algorithm for robot planning. In: *2004 IEEE international conference on robotics and automation (ICRA’04)*, vol. 3, pp. 2399–2404.
- Theocharous G and Magadevan S (2002) Approximate planning with hierarchical partially observable Markov decision processes for robot navigation. In: *Proceedings of the IEEE/RSJ international conference on robotics and automation*, Washington, DC.
- Thrun S (2000) Monte Carlo POMDPs. In: *Advances in Neural Information Processing Systems 12*. Cambridge, MA: MIT Press, pp. 1064–1070.