GPU-based Motion Planning under Uncertainties using POMDP

Taekhee Lee and Young J. Kim

Abstract— **We present a novel GPU-based parallel algorithm to solve continuous-state POMDP problems. We choose the MCVI (Monte Carlo Value Iteration) method as our base algorithm [1], and parallelize this algorithm using multi-level parallel formulation of MCVI. For each parallel level, we propose efficient algorithms to effectively utilize the massive data parallelism of GPUs. To obtain the maximum parallel performance at highest level, we introduce two workload distribution techniques such as data/compute interleaving and workload balancing. To the best of our knowledge, our algorithm is the first parallel algorithm that executes POMDP efficiently on GPUs. Our GPU-based algorithm outperforms the existing CPU-based algorithm by a factor of 75∼90 on different benchmarks.**

I. Introduction

The primary task of motion planning is to determine a motion that brings a robot to a desired state while avoiding collision with obstacles and minimizing the robot's efforts or resources [2]. Motion planning is considered an important problem in robotics and intelligent system, computer animation, CAD, etc. Many existing algebraic formulation of motion planning, such as configuration space-based approach, is designed based on an assumption that exact information about robot and its surrounding environment is known a priori. Moreover, the control and sensing capability of a robot is assumed to be precise. However, this assumption is only valid in carefully engineered environment, for instance, such as well-sealed manufacturing assembly lines. In real world, there are lots of uncertainties arising due to a lack of accuracy of robot sensors, imprecisions of robot controls, and alterations and imperfections of environment information. These uncertainties can harm the reliability of robot motion planning seriously.

Over the past years, there has been a significant research effort to deal with uncertainties such as sampling-based planning with sensor uncertainty [3], evaluating an uncertain region in the configuration space [4], the probabilistic roadmap algorithm for robust motion plans [5], a priori probability distributions along the robot path [6], etc. However, a more general framework to handle uncertainties arising in different part of motion planning is a POMDP (Partially Observable Markov Decision Process) [7]. This method considers every possibilities by planning with a belief that is a probability distribution over a state space.

Even though a POMDP model can deal with a wide range of uncertainties, it is known to be notoriously challenging to precisely evaluate in particular for continuous-state POMDPs and computationally intractable (i.e. PSPACE-hard). [8], [9]. In recent years, more progress has been made on developing an efficient, approximate POMDP algorithm. Especially point-based POMDP algorithms [10], [11], [12], [13], [10], [14] have drastically reduced the amount of computation while keeping the intrinsic value of POMDP and thus the POMDP model becomes a more appealing choice to handle uncertainties in motion planning. However, these efficient algorithms take hours of computation time to deal with moderate-size POMDP problems.

Since the introduction of programmable GPUs (graphics processing units), applying the massive parallelism of GPU threads to a challenging computational problem has drawn a lot of attention from different research communities [15]. The main theme of this paper is also to use the massive parallelism readily available on GPUs to accelerate the POMDP computation. However, parallelizing POMDP algorithms on GPUs is very non-trivial. First of all, POMDP requires a large size of memory to deal with the curses of dimensionality and history, however, the typical memory size of GPU is much less than that of CPU. Moreover, the memory architecture on modern GPU is hierarchical and its caching capability is less efficient than the CPU counterpart. Finally, existing efficient POMDP algorithms such as [11], [16], [17] are iterative and sequential by nature, and thus identifying parallelism from these algorithms becomes challenging.

Main Contributions: In this paper, we present a GPU-based parallel algorithm to solve continuous-state POMDP problems. Our algorithm is based on the MCVI (Monte Carlo Value Iteration) method, originally developed for CPU [1] , and we parallelize this algorithm using a novel, multi-level parallel formulation of MCVI; our parallel algorithm can be executed at four different levels, belief-, action-, policy graph node- and particle-level. For each level of parallelism, we propose efficient algorithms running on GPUs. In particular, to get the maximum parallel performance at highest level (i.e. belief-level), we introduce two workload distribution techniques such as data/compute overlapping technique and workload balancing using the paradigm of gathering, scheduling and running. To the best of our knowledge, our algorithm is the first parallel algorithm that executes PODMP efficiently on GPUs. In our experiments, we also show that our GPU-based algorithm outperforms the existing CPU-based algorithm by a factor of 75∼90 on different benchmarks.

II. Previous Work

A. POMDP-based Motion Planning

a) Discrete-state POMDP: The POMDP provides a general framework for planning using the imperfect infor-

T. Lee and Y. J. Kim are with the Department of Computer Science and Engineering at Ewha Womans University in Seoul, Korea. {taekhee.lee|kimy}@ewha.ac.kr

mation of state, sensor and action of a robot [7], [18], [19], applicable to such areas as operation research, artificial intelligence, and robotics [20], [21], [22]. Unfortunately, however, solving a POMDP is computationally intractable due to the curse of dimensionality and history [8], [9]. Thus, several approximate methods have been proposed to convert POMDP to fully observable Markov Decision Process(MDP) by applying heuristic strategies [20], [21], [22]. Alternatively, point-based algorithms have been successfully shown to approximately solve a POMDP by working only on a limited subset of belief space instead of the complete belief space [13], [23], [10], [14], [24], [25]. Several methods such as [14], [10], [23], [24], [13] solve a POMDP by sampling beliefs only from reachable spaces. They maintain both upper and lower bounds of the optimal value function to test whether the sampled point is reachable or not [11]. Even these point-based methods can solve a POMDP with tens of thousands of states in reasonable time, they still require lots of computation time and memory.

b) Continuous-state POMDP: To deal with real-world problems, POMDPs need to model the continuous states, actions and observations. To plan under continuous environments, [26], [27] discretize such environment with a grid. However, it is rather hard to determine the proper dimension of the grid and the discretization may deteriorate the quality of a solution. To address this issue, [28], [29], [12], [30] used Gaussian mixtures or particle filters to represent beliefs. However, if the environment has discontinuities such as obstacles, the number of Gaussian components grows too fast and becomes difficult to control. [31], [17], [32], [33] approximate the belief dynamics using an extended Kalman filter and provide locally-optimal solutions to continuous POMDP problems in polynomial time. [34] introduce a method for fast and safe motion planning using the probability of collision.

Monte Carlo Value Iteration (MCVI)-based methods [35], [1] solve continuous POMDP problems with an acceptably good result and speed. The MCVI also uses particles for representing a belief, and employs a policy graph to implicitly represent α vectors [36], [7]. To build a policy graph, MCVI performs Monte Carlo (MC) simulations. Although MCVI involves more computation than approximate methods such as [26], [27], it requires less memory by avoiding the inefficient discretization. More importantly, it can be relatively easy to parallelize, thus it is a natural choice to leverage the power of modern multi-core CPUs or GPUs on these algorithms. [37] takes advantage of multicore CPUs to devise a scalable parallel algorithm for point-based POMDPs and apply it to a complex and large domain.

B. GPU-based Motion Planning

Motion planning utilizing the massively parallel power of GPUs is relatively a new field. Existing GPU-based planning algorithms mostly focus on accelerating collision queries that act as a bottleneck of sampling-based planners. GPU-based algorithms such as [38], [39] use a work queue to parallelize collision queries based on bounding volume hierarchy. ITOMP [40] computes a collision-free trajectory that is smooth and satisfies dynamics constraint on GPU using a stochastic method. To the best of our knowledge, accelerating point-based POMDP algorithms using GPU still remains an open problem [41].

III. Preliminaries

A. Discrete-state POMDPs

Formally, the POMDP model is a tuple of $\{S,A,O,T,O(s),re_a(s)\}$ with a set of state S, actions A, observations O, a transition mode ($T(a,s) = p(s' \mid a,s)$, an observation model $O(s) = p(o \mid s)$, and a 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:

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

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

$$V_n(b) = \arg\max_a \; Q_n(b,a) \tag{2}$$

with

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

where S and O are discrete, $\gamma \in [0,1)$ is a discount factor, and n is the time horizon. Now, an optimal policy π^* can be defined as:

$$\pi^*(b) = \arg\max_a \; Q^*(b,a) \tag{4}$$

where Q^* is the Q-function associated with the optimal value function V^*. The optimal policy is often represented as 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* [18], [42], [7]. Here, V_i, the value of the current belief, is expressed as:

$$\begin{aligned} V_n(b) &= \max_{\{\alpha_n^i\}_i} \sum_S \alpha_n^i(s) b(s), \\ \alpha_n^i(s) &= \arg\max_{\{\alpha_n^i\}_i} \; b(s) \alpha_n^i \end{aligned} \tag{5}$$

where $\{\alpha_n^i\}_i$ is a set of vectors, each of which provides a value of a belief state. The value iteration generates a set of alpha vectors. In functional form, a value iteration is often expressed using the *Bellman recursion* [43].

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

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

B. Continuous-State POMDPs using MCVI

For continuous-state POMDPs such as the MCVI algorithm, a policy graph is used to represent a policy [7], [36]. A policy graph G is a directed graph with the nodes of an action $a \in A$ and the edges of an observation $o \in O$. The 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. The value function of b on G is derived from Eq.8 as:

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

We can evaluate $V_G(b)$ by performing MC simulation; sample many random states, called particles, from S with a probability of $b(s)$ and simulate their policy $\pi_{G,v}$ for $\forall v \in G$.

The MCVI algorithm finds an optimal policy by iteratively updating a policy graph G, and mainly consists of 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. To expand T_R, beliefs are generated from b_0 and evaluate the value function $V_G(b_0)$ by performing MC simulation on the policy graph G. Then, find the best node among generated beliefs and add it to T_R. Adding a new belief node continues until the depth of T_R is sufficiently high or the difference of the bounds on $V^*(b_0)$ is sufficiently reduced.
2) **Backup**: T_R is traversed from b_0 toward the terminal by performing the value iteration for every belief node to evaluate $V^*(b_0)$, and the policy graph G is updated to G' for each value iteration; this process is called MC-Backup. More specifically, an α function of a node v, α_v, of G is defined as:

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

$$\max_{a \in A} \left\{ \int_{s \in S} R(s,a) b(s) ds \right\} \tag{9}$$

Note that the sum in Eq.5 is replaced by an integral to handle the continuous-state space. The integration in Eq. 7 is performed using MC simulation; from Eqs.6 and 7, the optimal value function V^* can be calculated using value iteration (the Bellman recursion) as shown in Eq.10.
3) The sampling/backup steps are iterated until the difference between the upper and the lower bounds on $V^*(b_0)$ becomes less than a predefined value.

$$V_{G'} = HV_G = \max_{a \in A} \left\{ \int_{s \in S} R(s,a) b(s) ds + \gamma \sum_{o \in O} p(o \mid b, a) \max_{v \in G} \int_{s \in S} \alpha_v(s) b'(s) ds \right\} \tag{10}$$

Particle-level; Policy graph node-level; Action-level; Belief-level

IV. Multi-level Parallel POMDP Algorithm

Among other POMDP algorithms, the MCVI algorithm[1] is suitable for massive parallelism because (1) the value iteration using many particles can be executed in parallel and (2) the required memory footprint to run 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. Moreover, since the MC-Backup step explained in Sec.III-B takes 99% of the total running time in MCVI, our objective is to parallelize MC-Backup using massively-many parallel GPU threads. The implementation choice of our GPU parallelization is CUDA[44], mainly because CUDA is optimized for both high- and low-end GPUs and low-level optimization is available such as handling the cache or avoiding memory access conflicts.

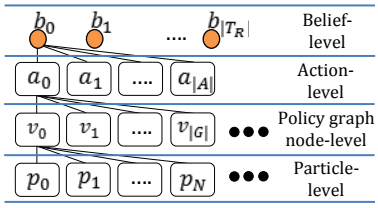

Fig. 1. **Backup Hierarchy** *The backup for a belief b_i is done in a hierarchical manner. $N \times |A| \times |G|$ simulations are needed for one belief node b_i to add a new node and edges to G.*

A. Policy graph node-level Parallelism

Eq. 10 is the main equation that constitutes the MC-backup process. We can further dissect this equation into four nested levels of computation (also illustrated in Fig.1): particle-, policy graph node-, action- and belief-level.

The easiest way to parallelizing Eq. 10 is to do it at policy graph node-level. More specifically, for each node v_i, we assigned a block of threads to it. Moreover, each particle p_i, mapped to a single thread, can share the high-performance CUDA shared memory assigned to the thread block that enables the rapid integration calculation. Algorithm 1 is the pseudo code for a CUDA kernel to perform the MC-simulation at policy graph node-level.

Algorithm 1 `MC-Simulation Kernel`
Input: $tID : ThreadID$, $bID : BlockID$, $b : Belief$, $G : PolicyGraph$
Output: $result[bID] : MC-simulation$

```
1:  p_tid = GenParticle(b);
2:  v_bid = PolicyGraphNode[bid];
3:  for i = 0 to simulationLength do
4:      if p_tid == goal then
5:          break;
6:      end if
7:      a = GetActionFromGraph(v_bid);
8:      reward[tid] += discount * GetReward(p_tid, a);
9:      o = GetObservation(p_tid, a);
10:     v_bid = GetNextNodeFromGraph(v_bid, o);
11:     p_tid = GetNextState(s, a);
12:     discount *= discountFactor;
13: end for
14: syncThread(); {sync all threads within a block}
15: if tid == 0 then {true only once per block}
16:     result[bid] = IntegrateRewardsOfAllThreads();
17: end if
```

Here, 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 blocks and N is the number of threads for each block; bID, tID are the block and thread indices. Particle on p_i in Fig. 1 can be mapped to p_{tid} and simulation is performed between the lines 3–13 using a particle p_{tid} sampled from a belief b on the belief tree T_R. During MC-simulation, 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 CPU to actually update the policy graph. After the kernel terminates, we obtain the integration value of v_i (i.e. MC-simulation result) from $result[i]$. The actual update on the policy graph occurs on the CPU side, since this operations requires random referencing on G that may not be well mapped to GPUs. Plus, this operation is not computationally heavy.

The performance result in Section. VI shows that Algorithm 1 outperforms CPU's result more than 10 times. However, it requires $|A|$ memory copies from GPU to CPU for MC-Backup on belief b and entire GPU threads are stalled whenever a memory copy is executed. This causes a serious performance loss. In the next section, we address this issue by increasing the parallel level to action-level.

B. Action-level Parallelism

To reduce the number of memory copies from GPU to 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 for $\forall a_i \in A$ and $\forall v_j \in G$, since each simulation result of a_i is independent of each other; thus, we call the same simulation kernel (Algorithm 1) for $\forall a_i \in A$ and $\forall v_j \in G$. 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 $PolicyGraphNode[bid]$ with $PolicyGraphNode[bid\%|A|]$ in Algorithm 1.
2) Execute Algorithm 1 in parallel for $|G| \times |A|$ blocks with N threads each.
3) The result of MC-simulation for v_j of a_i is $result[i \times |A| + j]$.

C. Belief-level Parallelism

In general, it is difficult to achieve the belief-level parallelism (i.e. running Algorithm 1 concurrently for $\forall b_i$), because the policy graph needs to be updated sequentially from b_i to b_{i+1}. However, the value iteration can be still performed on several beliefs concurrently while sacrificing the improvement rates to get the optimal value. The following theorem says that the backup on belief b_{i+1} without the backup on 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 on b_k for an arbitrary $k, 1 \leq k < n$ can improve the policy graph [45], [35], [1].*

However, if we run MC-simulation concurrently in arbitrary sequence (say, $b_0, b_k, ...$), 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 MCVI algorithm. However, Section. VI shows that the performance gain by applying the belief-level parallelism compensates for the loss of the improvement rate.

V. GPU Workload Distribution

As the parallel level in our algorithm goes up from particle- to belief-level, more and more blocks and threads are needed in one kernel execution; for instance, the total number of threads required at the belief-level parallelism is $|T_R| \times N \times |A| \times |G|$. As the number of kernel executions increases, so do the timings for retrieving the policy graph from CPU to GPU and updating it. 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 data/compute overlap. For example, a data transfer for stream 1 and kernel execution for stream 2 can be executed concurrently unless 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 to synchronize all threads to finish. For example, as illustrated in Fig.2, the maximum simulation length of v_0 is longer than v_1 and thus v_1 should wait until v_0 is finished and the threads assigned to v_1 becomes idle; similarly for v_2, v_3. Instead, if we run v_0, v_2 concurrently with a similar simulation time, the GPU idle time of core can be significantly reduced. The number of working sets is predefined and can be empirically obtained by experiments.

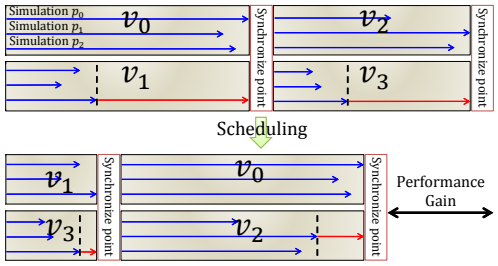

Fig. 2. **Performance gain of the workload balancing.** *The arrow means the simulation time for each thread. The red arrow is the idle time for each block to be synchronized. After workload scheduling, the idle time has been reduced from top to bottom.*

We perform three steps to achieve the above optimizations: gathering, scheduling, and rerunning:

Gathering G: In this step, we gather all blocks and store them at a three dimensional array with an index to the node v, action a and belief b. The array will be used for tossing the MC-simulation result of each block back 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 GPU. If this happens, we stop gathering the blocks and do the next step.

Scheduling S: The scheduling step determines the workload of each block, obtained from the gathering step, and group them into a few working sets. To determine the workload for a block v, we rely on simulation coherence. More specifically, after MC simulation is finished for v, we record the average simulation length of a thread at a table and use it for the workload of v of 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 (8 in our implementation). The actual scheduling is done by redistributing all the blocks while keeping the workload variance for each working set minimum.

Running R: The running step operates on the working sets with the compute/memory interleaving technique as follows. First, we create a number of CUDA streams as many as the number of working sets, and assign CUDA streams to each working set. Next, we perform an execution/retrieval sequence for all CUDA streams. Since execution and retrieval can be done asynchronously, we retrieve the results from a completed working set while executing the next working set.

VI. Results and Discussions

A. Performance Benchmark

We have implemented our algorithm on a Windows 7 PC, equipped with an Intel i7 2.67GHz CPU 3GB and NVIDIA GeForce 680 2GB. We have tested the performance of our algorithm on three robot motion planning tasks such as underwater navigation, corridor, and collaborative search and capture (i.e. herding), also used in [11], [12], [35].

Throughout the experiment, we set the number of working sets to 8 that was empirically obtained in our experiments. Due to the random nature of MCVI, we run the trial many times and measure its average performance. We also compare the performance of our algorithm against that of the CPU-based public MCVI library, APPL[1] [1].

Task	Reward	Policy	Action	Belief	GSR	APPL
Underwater	740	24	12	8	4	300
Corridor	1.5	34	18	15	9	726
Herding	15.5	804	297	204	139	12000

TABLE I

Performance Comparisons. *The second column is the target reward. Columns 3-7 are the total timings in seconds to reach the target using various levels of our algorithm and APPL library.*

Our experimental result of APPL is a bit different from [1] since some optimization parameters for caching and OpenMP are unknown in the public library. To compare the performances, we set the target total reward same for both APPL and our GPU algorithm, and measure the total timing until each of the algorithms reaches the target value. We use only single core CPU for the MCVI algorithm. As shown in Table I, our GPU-based algorithm outperforms the CPU-based MCVI algorithm by 75∼90 times.

[1] http://bigbird.comp.nus.edu.sg/pmwiki/farm/appl/

B. Performance Analysis

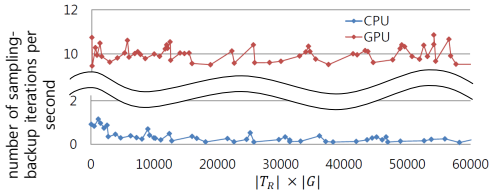

Fig. 3. **Number of Iterations for Underwater Navigation.** *Our GPU-based method (red) performs 10 sampling/backup iterations per second on average irrespective of the problem size while CPU-based APPL does only 0.1∼0.2 iterations.*

Fig.3 shows that our GPU-based method can perform more sampling/backup iterations per second than APPL for the underwater navigation benchmark, and the number of iterations remain constant regardless of the problem size (i.e. the size of belief tree $|T_R|$ and 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 sampling/backup operations. 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, since the number of iterations is not much affected by the problem size.

In Fig.4, 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 the underwater navigation. The total computation time using action-level is reduced to a half of policy graph node-level, mainly because the frequency of retrieving the integration value of v has been reduced. The arithmetic density further increases, as the belief-level parallelism is employed and the performance was enhanced by 150%. Finally, the belief-level parallelism 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.

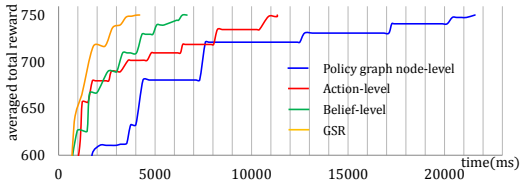

Fig. 4. **Performance Comparison using Different Levels of Parallelism.**

VII. Conclusion

In this paper, we present a GPU-based parallel algorithm to solve continuous-state POMDP motion planning problems under uncertainties. Our algorithm use the multi-level parallelism to perform MC-backup which is a bottleneck in existing CPU-based MCVI algorithm. We observe that our GPU-based algorithm outperforms the existing CPU-based algorithm by a factor of 75∼90 on different benchmarks. However, our algorithm is still limited by the capacity of GPU, for instance, the memory size of GPU. For future work, we would like apply our method to online planning algorithms such as [46], [47] or other planning problems involving uncertainties, and extend our method to more challenging 3D planning problems.

ACKNOWLEDGEMENT

This research was supported in part by IT R&D program of MKE/MCST/KOCCA (KI001818) and NRF in Korea (No.2012R1A2A2A01046246, No. 2012R1A2A2A06047007).

REFERENCES

[1] H. Bai, D. Hsu, W. Lee, and V. Ngo, "Monte Carlo value iteration for continuous-state POMDPs," in *Proc. The Int. Workshop on the Algorithmic Foundations of Robotics* 2010.

[2] S. M. LaValle, *Planning Algorithms*. Cambridge, U.K.: Cambridge University Press, 2006, available at http://planning.cs.uiuc.edu/.

[3] B. Brendan and B. Oliver, "Synthesis of hierarchical finite-state controllers for pomdps," in *2007 IEEE Int. Conf. on Robotics and Automation*, 2007.

[4] H. K. L. Guibas, D. Hsu and E. Rehman, "Bounded uncertainty roadmaps for path planning," in *Proc. The Int. Workshop on the Algorithmic Foundations of Robotics*, 2008.

[5] P. Missiuro and N. R. Adapting, "Adapting probabilistic roadmaps to handle uncertain maps," in *IEEE Int. Conf. on Robotics and Automation*, 2006.

[6] J. Berg, P. Abbeel, and K. Goldberg, "Lqg-mp: Optimized path planning for robots with motion uncertainty and imperfect state information," in *Proc. of Robotics: Science and Systems*, 2010.

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

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

[9] O. Madani, S. Hanks, and A. Condon, "On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems," in *Proc. 16th National Conf. on American Association for Artificial Intelligence*, 1999.

[10] M. T. J. Spaan and N. Vlassis, "A point-based pomdp algorithm for robot planning," in *IEEE Int. Conf. on Robotics and Automation*, 2004.

[11] H. Kurniawati, D. Hsu, and W. Lee, "SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces," in *Proc. Robotics: Science and Systems*, 2008.

[12] J. M. Porta, N. Vlassis, M. T. J. Spaan, and P. Poupart, "Point-based value iteration for continuous pomdps," *J. Mach. Learn. Res.*, vol. 7, pp. 2329–2367, Dec. 2006.

[13] D. Hsu, W. Lee, and N. Rong, "A point-based POMDP planner for target tracking."in *IEEE Int. Conf. on Robotics and Automation*, 2008.

[14] T. Smith and R. Simmons, "Point-based POMDP Algorithms: Improved Analysis and Implementation," in *Proc. of the Conf. on Uncertainty in Artificial Intelligence*, July 2005.

[15] D. Luebke, M. Harris, J. Krger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "Gpgpu: general purpose computation on graphics hardware," in *ACM SIGGRAPH 2004 Course Notes*, 2004.

[16] H. Kurniawati, Y. Du, D. Hsu, and W. Lee, "Motion planning under uncertainty for robotic tasks with long time horizons," *Int. J. Robotics Research*, vol. 30, no. 3, pp. 308–323, 2011.

[17] J. Berg, S. Patil, and R. Alterovitz, "Efficient approximate value iteration for continuous gaussian pomdps," in *Proc. AAAI Conf. on Artificial Intelligence*, 2012.

[18] E. J. Sondik, "The optimal control of partially observable markov processes," in *Stanford University PhD thesis*, 1971.

[19] R. D. Smallwood and E. J. Sondik, "The optimal control of partially observable markov processes over a finite horizon," *ARTIFICIAL INTELLIGENCE*, vol. 21, no. 5, pp. 1071–1088, 1973.

[20] R. Simmons and S. Koenig, "Probabilistic robot navigation in partially observable environments," in *Proc. of the Int. Jnt. Conf. on Artificial Intelligence*, 1995, pp. 1080–1087.

[21] A.R.Cassandra, L. Kaelbling, and J. Kurien, "Acting under uncertainty: Discrete bayesian models for mobile robot navigation," in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, vol. 2, 1996, pp. 963–972.

[22] G. Theocharous and S. Magadevan, "Approximate planning with hierarhical partially observable markov decision processes for robot navigation," in *Proc. of the IEEE/RSJ Int. Conf. on Robotics and Automation, Washington D.C.*, 2002.

[23] G. Shani, R. I. Brafman, and S. E. Shimony, "Forward search value iteration for pomdps," in *in Proc. Int. Jnt. Conf. on Artificial Intelligence*, 2007.

[24] J. Pineau, G. Gordon, and S. Thrun, "Point-based value iteration: An anytime algorithm for pomdps," in *Proc. of the Conf. on Uncertainty in Artificial Intelligence*, 2003, pp. 477–484.

[25] J. Hoey, A. von Bertoldi, P. Poupart, and A. Mihailidis, "Assisting persons with dementia during handwashing using a partially observable markov decision process," in *in Proc. Int. Conf. on Vision Systems*, 2007.

[26] S. Thrun, "Monte carlo POMDPs," in *Advances in Neural Information Processing Systems 12*, S. Solla, T. Leen, and K.-R. Müller, Eds. MIT Press, 2000, pp. 1064–1070.

[27] N. Roy, "Finding approximate pomdp solutions through belief compression," Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, September 2003.

[28] A. Brooks, A. Makarendo, S. Williams, and H. Durrant-Whyte, "Parametric pomdps for planning in continuous state spaces," *Robotics & Autonomous Systems*, vol. 54, no. 11, pp. 887–897, 2006.

[29] E. Brunskill, L. Kaelbling, T. Lozano-Perez, and N. Roy, "Continuous-state pomdps with hybrid dynamics," in *Int. Symp. on Artificial Intelligence & Mathematics*, 2008.

[30] S. Prentice and N. Roy, "The belief roadmap: Efficient planning in linear pomdps by factoring the covariance," in *Int. Symp. on Artificial Intelligence & Mathematics*, 2007.

[31] J. Berg, S. Patil, and R. Alterovitz, "Motion planning under uncertainty using iterative local optimization in belief space," *Int. Journal Robotics Research - IJRR*, 2012.

[32] J. Berg and S. Patil and R. Alterovitz, "Motion planning under uncertainty using differential dynamic programming in belief space," in *Proc. Int. Symposium of Robotics Research - ISRR*, 2011.

[33] S. Patil, J. Berg, and R. Alterovitz, "Motion planning under uncertainty in highly deformable environments," in *Proc. Robotics: Science and Systems - RSS*, 2011.

[34] S. Patil, J. Berg, R. Alterovitz, "Estimating probability of collision for safe planning under gaussian motion and sensing uncertainty," in *Proc. IEEE Int. Conf. on Robotics and Automation - ICRA*, 2012.

[35] Z. Lim, D. Hsu, and W. Lee, "Monte carlo value iteration with macro-actions," in *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[36] E. Hansen, "Solving pomdps by searching in policy space," in *Proc. AAAI Conf. on Artificial Intelligence*, 1998, pp. 211–219.

[37] G. Shani, "Evaluating point-based pomdp solvers on multicore machines," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 40, no. 4, pp. 1062 –1074, aug. 2010.

[38] J. Pan and D. Manocha, "Gpu-based parallel collision detection for real-time motion planning," in *Workshop on Algorithmic Foundation of Robotics (WAFR)*, 2010.

[39] J. Pan, D. Manocha, "Gpu-based parallel collision detection for fast motion planning," in *Int. Journal of Robotics Research (IJRR)*, 2012.

[40] C. Park, J. Pan, and D. Manocha, "Gpu-based parallel collision detection for real-time motion planning," in *Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 2012.

[41] G. Shani, J. Pineau, and R. Kaplow, "A survey of point-based pomdp solvers," *Autonomous Agents and Multi-Agent Systems*, pp. 1–51, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10458-012-9200-2

[42] M. Hauskrecht, "Value function approximations for partially observable markov decision processes," *Journal of Artificial Intelligence Research*, vol. 13, pp. 33–95, 2000.

[43] R. E. Bellman, "Dynamic programming," in *Princenton University Press*, 1957.

[44] NVIDIA, "Cuda programming guide 4.2," *http://developer.nvidia.com/cuda*, 2012.

[45] E. Hansen and R. Zhou, "Synthesis of hierarchical finite-state controllers for pomdps," in *Int. Conf. on Automated Planning and Scheduling*, 2003.

[46] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, "Online planning algorithms for pomdps," *Journal of Articial Intelligence Research*, 2008.

[47] H. S. Chang, R. Givan, and E. K. P. Chong, "Parallel rollout for online solution of partially observable markov decision processes," *Discrete Event Dynamic Systems*, 2004.