A Parallel Point-Based POMDP Algorithm Leveraging GPUs

Kyle Hollins Wray and **Shlomo Zilberstein**
College of Information and Computer Sciences
University of Massachusetts, Amherst, MA 01003
{wray, shlomo}@cs.umass.edu

Abstract

We parallelize the Point-Based Value Iteration (PBVI) algorithm, which approximates the solution to Partially Observable Markov Decision Processes (POMDPs), using a Graphics Processing Unit (GPU). We detail additional optimizations, such as leveraging the bounded size of non-zero values over all belief point vectors, usable by serial and parallel algorithms. We compare serial (CPU) and parallel (GPU) implementations on 10 distinct problem domains, and demonstrate that our approach provides an order of magnitude improvement.

Introduction

Partially Observable Markov Decision Processes (POMDPs) (Kaelbling *et al.* 1998) are a powerful mathematical model which describe the interactions of an agent in a stochastic environment, with numerous real-world applications such as semi-autonomous driving (Wray and Zilberstein 2015). Since POMDPs are PSPACE-hard, approximation algorithms have been developed, such as Point-Based Value Iteration (PBVI) (Pineau *et al.* 2003) and Perseus (Spaan and Vlassis 2005). This algorithm performs a single update equation (single instruction) over multiple parts of the POMDP (multiple data), which is amenable to parallelization. We present a General-Purpose Graphics Processing Unit (GPGPU) approach to PBVI, as well as discuss key optimizations which vastly improve performance all point-based algorithms. To the best of our knowledge, our approach and non-zero belief point optimization technique are novel. Parallel PBVI for multi-core CPUs has been explored (Shani 2010); however, GPUs enable additional parallelism since they may execute tens of thousands of threads. GPUs have been used with Monte Carlo value iteration, but not PBVI (Lee and Kim 2013). Our main contributions include: (1) GPGPU-based parallel PBVI algorithm, (2) an optimization for PBVI, (3) experiments that demonstrate an order of magnitude improvement, and (4) an easy-to-use library entitled *nova* which contains both our CPU and GPU implementations.

Optimizing Parallel PBVI

A POMDP is a tuple $\langle S, A, \Omega, T, O, R \rangle$. S is a set of n states. A is a set of m actions. Ω is a set of z observations. $T : S \times A \times S \rightarrow [0, 1]$ is a state transition such that $T(s, a, s') = Pr(s'|s, a)$. $O : A \times S \times \Omega \rightarrow [0, 1]$ is an observation transition such that $O(a, s', \omega) = Pr(\omega|a, s')$. $R : S \times A \rightarrow \mathbb{R}$ is a reward (denoted $R(s, a)$). POMDPs have a *horizon* $h \in \mathbb{N}$ and a *discount factor* $\gamma \in (0, 1)$. The agent maintains a set of r *beliefs* $B \subseteq \triangle^n$ over the true state (standard n-simplex $\triangle^n$). A *policy* $\pi : B \rightarrow A$ determines which action to take based on a belief. The goal is to find a policy that maximizes the expected reward over time, or *value function* $V : B \rightarrow \mathbb{R}$. Since it is piecewise linear and convex, we use a set of α-*vectors* $\Gamma = \{\alpha_1, \ldots, \alpha_r\}$. At time t, the value at b is:

$$V^t(b) = \max_{a \in A} \sum_{s \in S} b(s)R(s, a) + \sum_{\omega \in \Omega} \max_{\alpha \in \Gamma^{t-1}} \sum_{s \in S} b(s)V^t_{sa\omega\alpha}$$

$$V^t_{sa\omega\alpha} = \gamma \sum_{s' \in S} O(a, s', \omega)T(s, a, s')\alpha(s')$$

PBVI updates the α-vectors Γ^t only at a fixed set of beliefs:

$$\Gamma^t_{a\omega} = \{[V^t_{s_1 a\omega\alpha}, \ldots, V^t_{s_n a\omega\alpha}]^T, \forall \alpha \in \Gamma^{t-1}\}$$

$$\Gamma^t_b = \{[R(s_1, a), \ldots, R(s_n, a)]^T + \sum_{\omega \in \Omega} \operatorname*{argmax}_{\alpha \in \Gamma^t_{a\omega}} \alpha \cdot b, \forall a \in A\}$$

$$\Gamma^t = \{\operatorname*{argmax}_{\alpha \in \Gamma^t_b} \alpha \cdot b, \forall b \in B\}$$

There are two main optimizations required for GPU algorithms: (1) *coalesced* memory access (each thread accesses contiguous lines in memory; i.e., the same *cache line*) and (2) avoid *branching* (e.g., avoid if-statements and keep threads synchronized to the same instructions). Thus, all variables (T, B, etc.) are structured arrays which ensure that thread "execution groups" called *warps* are aligned memory accesses to the same region (coalesced access), and we use a *reduction* for argmax (avoid branching).

Our approach has two steps which fully parallelize computing Γ^t_b and Γ^t. First, we launch threads that compute the optimal $\alpha \in \Gamma^t_{a\omega}$ by evaluating each $\alpha \cdot b$ in parallel (rmz *blocks* of r threads). Importantly, we only store the value and index of $\alpha \cdot b$, use a reduction to compute the maximum in $\log(r)$, and recompute it compute $\alpha_{ba} \in \Gamma^t_b$. Second, we perform a similar technique for Γ^t for each B.

We limit array sizes to improve performance and reduce RAM usage so that we can solve much larger POMDPs. Specifically, we only use state-action pairs for R and limit the size of T and B (optionally, O and A). We let n_s denote the maximum *number of successors* such that $n_s =$

Domain	n	m	z	r_1	n_s	r_z	$\|B_1\| = r_1$		$\|B_2\| = 2r_1$		$\|B_3\| = 3r_1$	
							GPU	CPU	GPU	CPU	GPU	CPU
Tiger	2	3	2	16	2	2	0.002	0.001	0.003	0.002	0.004	0.004
Grid (4x3)	11	4	6	32	9	4	0.014	0.013	0.027	0.046	0.040	0.097
Tiger Grid	36	5	17	128	5	32	1.475	5.507	3.154	21.733	5.619	48.963
Hallway2	92	5	17	256	88	88	7.896	55.856	23.271	236.361	51.184	491.354
Tag	870	5	30	256	5	29	6.607	41.081	23.952	169.765	71.665	358.989
Rock Sample (7x8)	12545	13	2	256	1	256	8.140	45.897	17.754	170.649	40.136	344.447
Drive: San Franc.	172	8	2	256	2	2	0.073	0.216	0.144	0.803	0.259	1.741
Drive: Seattle	432	10	2	512	2	2	0.233	1.180	0.597	4.255	1.125	9.552
Drive: N.Y.C.	1064	12	2	1024	2	2	1.083	7.256	2.997	24.827	6.468	56.010
Drive: Boston	2228	12	2	2048	2	2	5.288	27.700	19.155	96.822	41.544	202.478

Table 1: Update step computation time (in seconds) for 6 well-known domains and 4 Drive cities averaged over 10 trials.

$\max_{s,a} |\{s' \in S | T(s,a,s') > 0\}|$, define $\mathcal{S} : S \times A \rightarrow \mathcal{P}(S)$ as the mapping to available successor states, and redefine $T : S \times A \times \{1, \ldots, n_s\} \rightarrow (0, 1]$. State sizes tend to have a large impact on performance and memory. This common optimization enables larger problems to be quickly solved.

To the best of our knowledge, no one has discussed a similar optimization with respect to belief points: remove zero values from all $b \in B$. All uses of b are dot-products in our PBVI equations, for which computation can be greatly reduced. Similarly, we let r_z denote the maximum *number of non-zero belief values* such that $r_z = \max_b |\{s \in S | b(s) > 0\}|$, define $\mathcal{Z} : B \rightarrow \mathcal{P}(S)$ as the mapping to non-zero belief points, and redefine all $b \in B$ to be parallel arrays to the state-arrays in $\mathcal{Z}(b)$ of only the non-zero values over beliefs.

Experimentation

We implemented our parallel approach using CUDA[1] on an Intel(R) Core(TM) i7-4702HQ CPU at 2.20GHz, 8GB of RAM, and an Nvidia(R) GeForce GTX 870M graphics card using C++, CUDA(C) 7.0, and Python 3.4.3. We compared the computation time of a single update step of GPU and CPU implementations. These two algorithms were run on 10 problem domains, each with a varying number of states, actions, observations, and structure regarding the state and observation transitions, linearly varying the number of belief points within each domain. We include 4 cities from the semi-autonomous driving domain (Wray and Zilberstein 2015), denoted *Drive*, using the road distance cost function. Table 1 shows the domains and final run time results. They represent averages over 10 trials in order to reduce any noise resulting from our belief point expansion.

The complexity of our CPU implementation's update step is $O(mzrn_s(n+rr_z))$. In comparison, our GPU implementation's complexity is $O(r_z(m+n_s)+\log(r)+n_s+n)$. This assumes an ideal scenario in which enough threads are created to perform instructions in parallel. The exponential increase of hardware performance and capability will likely make this increasingly more realizable with time.

Conclusion

Our results demonstrate approximately an order of magnitude improvement is realizable with the current generation of GPUs. The vast asymptotic difference between our implementations, however, suggests that this can be greatly improved by increasing the number of threads. Additionally, the comparison between Drive ($r_z \ll n$) and domains such as Tiger Grid ($r_z \approx n$) provides strong evidence that the maximum number of non-zero belief points r_z significantly affects performance. In general, proactively incorporating how many non-zero values are in *all* probability vectors (e.g., our B optimization "trick") can yield large performance gains.

Our work presents a new perspective on designing parallel algorithms for POMDPs by leveraging the power of GPUs. Experimentation shows that our GPU-based implementation successfully outperforms our CPU-based implementation, utilizing the aforementioned optimizations. We compiled these implementations into a single open-source library entitled *nova*, which also includes parallel solvers for other MDP models, so that others may build upon this work to produce high-performance parallel algorithms.

Acknowledgments

This work was supported in part by NSF grant IIS-1405550.

References

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Journal of Artificial Intelligence Research*, 101(1):99–134, 1998.

Taekhee Lee and Young J. Kim. GPU-based motion planning under uncertainties using POMDP. In *Proc. of the IEEE International Conference on Robotics and Automation*, pages 4576–4581, 2013.

Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. *Proc. of the 18th International Joint Conference on Artificial Intelligence*, pages 1025–1032, 2003.

Guy Shani. Evaluating point-based POMDP solvers on multicore machines. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 40(4):1062–1074, 2010.

Matthijs T. J. Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for POMDPs. *Journal of Artificial Intelligence Research*, 24(1):195–220, 2005.

Kyle Hollins Wray and Shlomo Zilberstein. Multi-objective POMDPs with lexicographic reward preferences. In *Proc. of the 24th International Joint Conference of Artificial Intelligence*, pages 1719–1725, 2015.

[1]https://developer.nvidia.com/about-cuda