Point-Based Value Iteration Algorithms for Multi-Core Machines

Guy Shani, Microsoft Research
guyshani@microsoft.com

Abstract

Recent scaling up of POMDP solvers towards realistic applications is largely due to point-based methods which quickly provide approximate solutions for medium-sized problems.

New multi-core machines offer an opportunity to scale up to much larger domains. These machines support parallel execution and can speed up existing algorithms considerably.

In this paper we suggest several ways in which point-based algorithms can be adapted to parallel computing. We overview the challenges and opportunities and present experimental evidence to the usability of our suggestions. Our results show that the opportunity lies mainly in parallelizing at the algorithmic level, not at the point-based backup level.

Introduction

Many interesting problems can be modeled as partially observable Markov decision problems (POMDPs). Due to the difficulty in computing an optimal policy, research has focused on approximate methods for solving POMDPs.

Policy generation via the computation of a (near) optimal value function is a standard approach for solving POMDPs. Point-based methods are currently provide the most effective approximations of the optimal value function. In point-based algorithms, a value function is computed over a finite set of reachable belief points, hoping that the value function would generalize well to the entire belief space. Such algorithms have shown the ability to scale up to medium-sized domains, supplying policies with good quality.

In the past few years it seems that the ability to speed up the clock rates of processors is nearing its boundary. The processor industry is now moving in a different direction in order to enhance the performance of processors — placing multiple processors (cores) on a single chip (Sutter & Larus 2005). Thus, new machines provide multi-threading programming abilities to enhance the performance of applications. It is now not uncommon to use machines with 8 or 16 cores, but in the near future systems are expected to have dozens, if not hundreds of cores. While multi-core algorithms rarely achieve even a linear speedup, it is still our hope that through using large amounts of cores we can scale up to POMDP domains which are currently beyond our reach.

In view of this promising technology we review point-based methods, suggesting adaptation of existing methods to multi-core architectures. We suggest modifications of existing algorithms as well as combinations of several algorithms. These modifications pose several challenges and difficulties that must be dealt with.

We provide some preliminary partially evaluation of the various modifications suggested here. It is important to note that even if current scaling up due to multi-thread implementations is limited because of the relatively small number of cores on a single machine, our suggested enhancements will be much more pronounced in the future, when machines will have an order of magnitude more cores. In fact, we cannot even execute a combination of all of our suggestions on our current machine. We are hence unable to demonstrate the full capability of the methods we propose here. We therefore consider the empirical evaluation provided here as no more than an evidence that our ideas have the potential to provide substantial speed up for point-based solvers.

Nevertheless, our empirical results show that there is little to gain by parallelizing the low level point-based backups. However, parallelizing algorithms such as PBVI and PVI has shown much potential for scaling up. We also demonstrate how the combination of various algorithms can help us to leverage the advantages of the different approaches, resulting in a rapid learning of good quality policies, superior to those learned by any single algorithm. Finally, we suggest a new way to prune dominated vectors by running simulations on a dedicated thread.

Background and Related Work

MDPs, POMDPs and the belief-space MDP

Markov Decision Processes (MDPs) model autonomous agents, acting in a stochastic environment. An MDP is a tuple $\langle S, A, tr, R \rangle$ where: S is the set of all possible world states, A is a set of actions the agent can execute, $tr(s, a, s')$ defines the probability of transitioning from state s to state s' using action a, and $R(s, a)$ defines a reward the agent receives for executing action a in state s. An MDP models an agent that can directly observe the environment state.

A Partially Observable Markov Decision Process (POMDP) is designed to model agents that do not have direct access to the current state, but rather observe it through noisy sensors. A POMDP is a tuple $\langle S, A, tr, R, \Omega, O, b_0 \rangle$

where S, A, tr, R compose an MDP, known as the underlying MDP. This MDP models the behavior of the environment. Ω is a set of available observations — the possible output of the sensors. $O(a, s, o)$ is the probability of observing o after executing a and reaching state s, modeling the sensor noise.

As the agent is unaware of its true world state, it must maintain a *belief* over its current state — a vector b of probabilities such that $b(s)$ is the probability that the agent is at state s. Such a vector is known as a belief state or *belief point*. b_0 defines the initial belief state.

Given a POMDP it is possible to define the belief-space MDP — an MDP over the belief states of the POMDP. The transition from belief state b to belief state b' using action a is deterministic given an observation o and defines the τ transition function. We denote $b' = \tau(b, a, o)$ where:

$$b'(s') = \frac{O(a, s', o) \sum_s b(s)tr(s, a, s')}{pr(o|b, a)} \quad (1)$$

Value Functions for POMDPs

An agent that uses an MDP or POMDP attempts to optimize some function of its reward stream, such as the discounted reward $\sum_i \gamma^i r_i$. The discount factor $0 < \gamma < 1$ models the higher value of present rewards compared to future rewards.

A solution to an MDP or POMDP is usually a stationary policy π — a mapping from states (MDP) or belief states (POMDP) to actions. Policies can be computed through a value function that assigns a value to each state.

The value function V for the belief-space MDP can be represented as a finite collection of $|S|$-dimensional vectors known as α vectors. Thus, V is both piecewise linear and convex (Smallwood & Sondik 1973). A policy over the belief space can be defined by associating an action a with each vector α, such that $\alpha \cdot b = \sum_s \alpha(s)b(s)$ represents the value of taking a in belief state b and following the policy afterwards. It is therefore standard practice to compute a value function — a set V of α vectors, inducing a policy π_V by $\pi_V(b) = \text{argmax}_{a:\alpha_a \in V} \alpha_a \cdot b$

We can compute the value function over the belief-space MDP iteratively by applying the Bellman equation:

$$V_{n+1}(b) = \max_a [b \cdot r_a + \gamma \sum_o pr(o|a, b)V_n(\tau(b, a, o))] \quad (2)$$

where $r_a(s) = R(s, a)$ is a vector representation of the reward function. Indeed, the above equation can be written in terms of vectors (Smallwood & Sondik 1973; Cassandra, Littman, & Zhang 1997), updating a value function defined as a set of vectors over the entire belief space. Unfortunately, such an update is not tractable for even very small domains.

Point Based Value Iteration

A possible approximation is to compute an optimal value function over a finite subset B of the belief space. We hope that the computed value function will generalize well for other belief states not included in B. Point-based algorithms choose a subset B of the belief points that is reachable from the initial belief state through different methods, and compute a value function only over the belief points in B.

The value function can be updated for a single belief point b only, generating an α-vector that maximizes the value of b. The computation of the next value of b — $V_{n+1}(b)$ out of the current V_n (Equation 2) is known as a *backup* step. The backup step can be implemented efficiently (Zhang & Zhang 2001; Pineau, Gordon, & Thrun 2003; Spaan & Vlassis 2005) by:

$$backup(b) = \text{argmax}_{g_a^b : a \in A} b \cdot g_a^b \quad (3)$$

$$g_a^b = r_a + \gamma \sum_o \text{argmax}_{g_{a,o}^\alpha : \alpha \in V} b \cdot g_{a,o}^\alpha \quad (4)$$

$$g_{a,o}^\alpha(s) = \sum_{s'} O(a, s', o)tr(s, a, s')\alpha^i(s') \quad (5)$$

We now briefly review the methods we attempt to improve in this paper. For a better introduction to these algorithms, we refer the reader to the original papers.

PBVI: The Point Based Value Iteration (PBVI) algorithm (Pineau, Gordon, & Thrun 2003), begins with $B = \{b_0\}$, and at each iteration repeatedly updates the value function using all the points in the current B:

$$V' = \{\alpha_b : b \in B, \alpha_b = backup(b)\} \quad (6)$$

After the value function has converged the belief points set is expanded with all the most distant immediate successors of the previous set:

$$Succ(b) \leftarrow \{b' | \exists a, \exists o \, b' = \tau(b, a, o)\} \quad (7)$$

$$B' \leftarrow B \cup \{b_i^* : b_i \in B, b_i^* = argmax_{b' \in Succ(b_i)} dist(B, b')\} \quad (8)$$

Perseus: Spaan and Vlassis (2005) suggest to explore the world using a random walk from the initial belief state b_0. The points that were observed during the random walk compose the set B of belief points. The Perseus algorithm then iterates over these points in a random order. During each iteration backups are executed over points whose value has not yet improved in the current iteration.

PVI: The Bellman error is the improvement that will be gained from an update to belief state b:

$$e(b) = max_a [r_a \cdot b + \gamma \sum_o pr(o|b, a)V(\tau(b, a, o))] - V(b) \quad (9)$$

In the context of MDPs, updating states by order of decreasing Bellman error can speed up the convergence of value iteration. The Prioritized Value Iteration (PVI - Shani et al. (2006)) is an adaptation of this technique to POMDPs.

Like Perseus, PVI receives as input a predefined set of belief points B and computes an optimal value function over these points. PVI always execute a backup over the belief point with the maximal Bellman error. As opposed to the MDP case, after each vector was added to the value function, the Bellman error must be updated for all belief points in B. While PVI computes a small number of backups compared to other point based algorithms, the full update of the Bellman error is time consuming and reduces the efficiency of the algorithm considerably.

HSVI: The Heuristic Search Value Iteration (HSVI) is a point based and trial based algorithm. HSVI (Smith & Simmons 2005) maintains both an upper bound ($\bar{V}$) and lower bound ($\underline{V}$) over the value function. It traverses the belief space following the upper bound heuristic, selecting successor belief points where the gap $\bar{V}(b) - \underline{V}(b)$ should be reduced. It then executes updates over the observed belief points on the explored path in reverse order. Smith and Simmons show that within a finite number of updates, all reachable belief points will be finished. A belief state is finished once the gap between the bounds has been sufficiently reduced.

FSVI: Forward Search Value Iteration (FSVI) simulates an interaction of the agent with the environment, maintaining both the POMDP belief state b and the underlying MDP state s — the true state of the environment within the simulation (Shani, Brafman, & Shimony 2007). While at policy execution time the agent is unaware of s, in simulation we may use s to guide exploration.

FSVI uses the MDP state to decide which action to apply next based on the optimal value function for the underlying MDP, thus providing a path in belief space from the initial belief state b_0 to the goal (or towards rewards). The trial is ended when the state of the underlying MDP is a goal state, or after the number of steps exceeds a predefined threshold.

Parallelizing Point-Based Algorithms

When suggesting parallel implementations there are a number of important issues that must be considered:

Algorithm semantics — a parallel execution may cause an algorithm to behave differently than if it was executed over a single thread. When the semantics change it is possible that algorithm features, such as convergence guarantees, no longer hold. It is therefore important to explicitly identify changes that do not maintain the algorithm semantics.

Synchronized vs. A-Synchronized — in a synchronized setting threads must synchronize their computation process advancements. As such, in many cases some threads must wait for the results of other threads. A-synchronized applications allow each thread to advance on its own, thus avoiding wasted wait time. However, this is usually achieved by somewhat changing the semantics of the algorithm.

Synchronized access — even though threads may not be synchronized, access to shared memory may still need to be synchronized, in order to avoid the corruption of the data sets. Synchronized access may be implemented by locks, yet this may cause considerable slowdown.

Multi-thread overhead — when using multiple threads there is an overhead in starting and stopping the threads. This overhead can be reduced by using design patterns such as a Thread Pool (e.g. (Gomaa & Menascé 2000)). Still, in many cases it is inefficient to split too short computations into multiple threads and a careful balancing is required between the number of threads and the length of the computation task assigned to each thread.

Synchronized Distributed Computations

Point-Based Backups: The best scenario in parallel computation is when computations can be executed in parallel without affecting the result of the process. The most simple example of this scenario are operations such as Σ or $\max$ where the various components can be computed independently, and only afterwards the aggregated result is processed. The aggregation operator requires that all values will be present and therefore requires that all threads terminate. Thus, a synchronization step is needed in all these operations in order not to change the semantics of the algorithm.

The point-based backup (Equations 3 to 5) offers such an opportunity. These equations contain both Σ and $\max$ operations that can be parallelized. In the most extreme case we could use $|A||O||V||S|$ threads to compute the vector entries of all the $g^{\alpha}_{a,o}$ components. However, as too short tasks are not desirable, our less extreme approach uses $|A|$ threads, where each thread is computing a single g^b_a component.

When HSVI computes the next belief state in the traversal, and when PBVI expands the belief space, the computation of all the successors of a belief state is required. In this case, it is possible to apply parallel computing to compute each successor belief state in a different thread.

PBVI: Moving to a higher level of concurrency, we will now look at how PBVI can be parallelized. The two main procedures of PBVI — expanding the belief subspace B (Equation 8) and improving the value function (Equation 6) can both be executed in parallel. It is possible to run the backup process over each belief state independently of the other belief states. Here, the arbitrary order of backup computations becomes highly useful. We cannot guarantee the order of computations of the various threads, but PBVI does not impose any order.

The belief expansion step of PBVI is extremely time consuming, due to the need to compute distances between many pairs of points. It is possible to compute for each belief state in B its most distant successor regardless of the successors of other points. Also, belief expansion does not depend on the value function. This gives us another opportunity for parallelizing; We can compute the expansion of the belief space at the same time as the value function update.

These changes require some synchronization steps in order not to modify the semantics of PBVI. After each belief space expansion we must run the value function update until convergence. After splitting the belief point backups to different threads, we need to wait for all threads to finish before starting another value function update.

PVI: A second algorithm that offers an opportunity for straight forward concurrency is PVI. In PVI much of the computation difficulty comes from the need to update all the Bellman errors after each new α-vector was added. Shani et al. (2006) suggest to resolve this by updating the Bellman error only over a sample of the points. However, by splitting the error updates into several threads, an exact computation can be done rapidly. Again, there is a need to wait until all threads have terminated, find the point with the maximal error, and execute a backup. Then, a new Bellman error update can be started. Therefore, this version is once again synchronized.

Combining Point-Based Algorithms

Up until now we have discussed how specific point-based algorithms can be enhanced by using a multi-core architecture. We can also run several different algorithms over the same domain in parallel. Assuming that different algorithms have different strengths, the combined result of all algorithms might be better than a single algorithm.

For example, Shani et al. (2007) explain how their FSVI algorithm cannot tackle information gathering tasks. However, in many interesting domains, FSVI rapidly computes high quality policies. It is likely that by combining FSVI with other algorithms, such as HSVI that are guaranteed to find an optimal policy, will result in an algorithm that is both fast and provably optimal.

We suggest that the algorithms will share the same value function. This approach is superior to executing all algorithms using different value functions and then joining the value functions together, as good α vectors discovered by one algorithm can be used by all others.

However, such an approach will modify the semantics of the algorithms. For example, it is possible that while PVI updates the Bellman error, new vectors will be inserted making the Bellman error inaccurate again. As a result, PVI is no longer guaranteed to always execute the best local backup.

Pruning Dominated Vectors

A problem that may rise when having multiple algorithms adding α vectors to the value function is that the size of the value function may grow too much. Pruning vectors is not easy. While only the vectors that are part of the upper envelope are important, discovering which vectors are dominated is difficult. In the Incremental Pruning algorithm (Cassandra, Littman, & Zhang 1997) linear programs are used to find a witness belief state for which an α vector is optimal, thus proving that the vector is a part of the upper envelope. However, linear programs degrade performance considerably.

We offer a simpler approach to finding witnesses for α vectors. It is possible to run simulations of executions of the current value function over the environment. Through the simulation we record for each α vector the number of times it was used. After the simulation is over, we can prune out vectors that were never used, since no belief state within the simulation proved to be a witness for this α vector.

The simulations must execute a reasonable number of trials before pruning out vectors, so that different possible traversals through the belief space following the current policy will be selected. Given a sufficient number of iterations the probability that an important vector will be removed is low, but even if such an event has occurred, the vector can be recomputed again. Simulations also offer us another opportunity for parallelizing, since trials are independent and can be carried out by different threads. Thus, a large number of trials can be rapidly done, enhancing the probability that all important vectors were observed.

This method also ensures that we consider only reachable witnesses, and might therefore prune out vectors for which the linear program might have found an unreachable witness. While normally running these simulations will cause a point-based algorithm to slow down considerably, running this procedure in a different thread can be beneficial.

After the simulations are done we need to filter out vectors that were dominated from the value function. As the value function is shared between different threads, we should make this filtering carefully. We suggest the following method — our value function has a *read* pointer and a *write* pointer. Usually both point to the same vector set. New vectors are always written using the write pointers and algorithms use the read pointer to find optimal vectors. Once the simulations are done, the pruning thread initializes the write pointer to a new empty set. Now, the thread writes to this new set all vectors that were observed during the simulations. If other algorithms attempt to add new vectors, they also do so into this new set, through the write pointer. Once all vectors have been added, the read pointer is also changed to point to the new set. This way we do not have to block algorithms that are trying to read or write vectors during the filter process, but it is possible that some requests will not get the best possible vectors.

Empirical Evaluations

We ran several sets of tests in order to provide evidences to the various multi-threaded enhancements we suggest. Our tests were run on relatively small domains than the ones currently used in literature, so that we can run multiple executions rapidly. The limited scope of this paper also does not allow us to evaluate all the suggested modifications, and we leave additional evaluations to future work.

To implement our approach we used the Thread Pool design pattern (see, e.g. (Gomaa & Menascé 2000)), to reduce the cost of creating threads. Each thread executes tasks taken from a task queue. Threads can write more tasks to the queue and wait for the tasks to finish. All our experiments were done on an dual quad-core machine (8 processors) with 2.66GHz processors. We use Java and the JVM is limited to 1.5GB of RAM.

Results

Point-Based Backups: We begin by evaluating the advantages of parallelizing the backup process itself. We assign to each thread the computation of a single g_a^b operation (Equation 4), requiring $|A|$ threads only at a time. Farther splitting the tasks so that each $g_{a,o}^{\alpha}$ (Equation 5) is computed on a different thread reduced the performance.

We experimented over four domains — Hallway, Hallway2 (Littman, Cassandra, & Kaelbling 1995), TagAvoid (Pineau, Gordon, & Thrun 2003) and RockSample (Smith & Simmons 2005). We compare PBVI using different numbers of threads. PBVI was executed for 10 of iterations and we measure the average time (milliseconds) of backup executions. We executed each experiment 5 times and report in Table 1 the average result. Standard deviations were extremely small — less than 0.01. The number of backups that were used changes between domains but is always more than 5000 during each execution.

The results for TagAvoid are rather surprising — splitting the backup process into tasks has helped very little. However this because the agent location is completely observable. Thus, the number of successors of a belief state is very small. This greatly reduces the computation time of the backup process and therefore makes any farther splitting of the operation useless.

Threads	1	2	4	8
Hallway	34	31	27	14
Hallway2	190	154	93	74
TagAvoid	33	38	32	20
RockSample5,7	108	59	33	17

Table 1: Comparing multi thread execution time (milliseconds) of the backup operation.

	PBVI		PVI	
Domain	S	M	S	M
Hallway	195	76	416	66
Hallway2	592	324	693	110
TagAvoid	447	83	135	26
RockSample5,7	672	162	2069	307

Table 2: Comparing single (S) and multi (M) thread execution time (seconds) of PBVI and PVI.

The results are most pronounced for the RockSample domain. This is both because this is the largest domain, and hence, backups take relatively long to complete. Also, this domain has the largest amount of actions, and therefore all threads can be used simultaneously. In fact, in this domain it appears that adding more threads beyond the 8 we used would have resulted in increased performance.

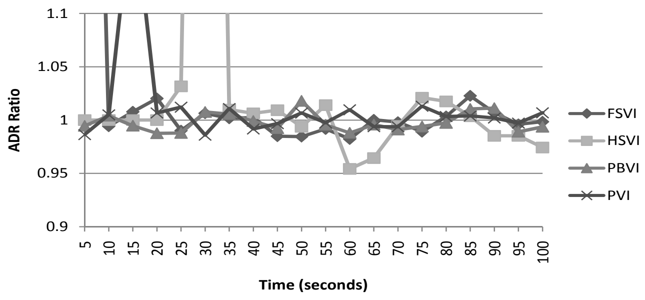
PBVI: The PBVI algorithm can be parallelized by splitting the belief states into threads both when computing point-based backups and when expanding the belief space. We ran PBVI for 10 of iterations over the 4 domains, and report average time over 5 executions. Table 2 (middle column) compares single (S) and multi (M) thread execution time (seconds) for the various domains.

As expected, computing the value function update and the belief set expansion over different points in parallel is useful. In the first few iterations the advantage of multiple threads is less noticeable, due to the small number of operations in each set. As the algorithm advances the effect of having multiple threads becomes more pronounced, since much more work can be done in each thread independently.

PVI: In PVI the computation of the Bellman error can be divided into different threads for different belief points. PVI was executed over a belief set of 1000 points for 150 iterations. Table 2 (right column) compares single (S) and multi (M) thread execution time (seconds) for the various domains.

The results here are encouraging. PVI speedup is more noticeable than PBVI. This is mainly because in PVI we are computing a value function over 1000 points, while PBVI during most of the iterations had much less points. Therefore, the number of points assigned to a thread is much larger for PVI. This demonstrates again the need to properly balance the amount of work a thread is executing, and not to split the computation too much.

Vector Pruning: We ran the PBVI, PVI, HSVI and FSVI with and without pruning, stopping the execution every 5 seconds, computing ADR (average discounted reward) over $10,000$ trials and outputting the number of vector in the current value function ($|V|$). The pruning thread executed 500 trials, pruned unobserved vectors and started over. Figures 1 and 2 present our results. We show here the ADR and the $|V|$ ratio with and without pruning. The ratio is computed by taking the value without pruning and dividing by the equivalent value with pruning. In the ADR case a ratio below 1 means that pruning improves the quality of the value function. In the V case a higher ratio means that more vectors were pruned. Pruning might improve the value function since pruning vectors results in faster backup operations. Therefore, an algorithm manages to execute more iterations in the given time frame and therefore creates a better policy within the given time frame.



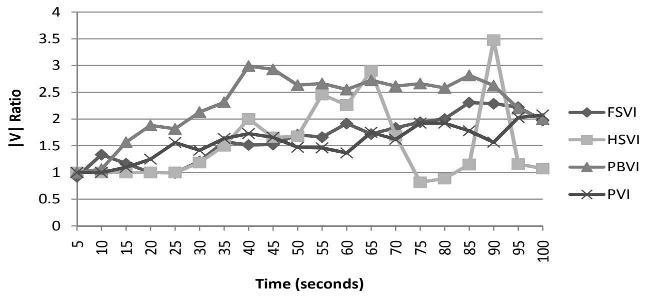
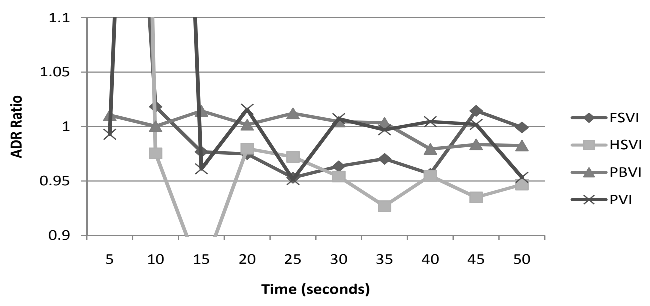


Figure 1: Vector pruning over the Hallway domain.

Over the Hallway domain, vector pruning reduces the value function in most cases to half its original size. This means that for about half of the vectors the pruning method could not find a reachable witness belief point. When looking at the ADR ratio in the Hallway domain we see that the resulting policy quality is very similar. The method that displays the largest deviation is HSVI which was slow to converge and therefore its value function still fluctuated within the checked time frame.

The results are even more encouraging over the Tag Avoid domain. In this domain pruning not only considerably reduces the number of vectors, but also provides substantial improvement in ADR. This is because when pruning was used, backup operations become faster and therefore the algorithm has executed more iterations within the time frame.

Combining Algorithms Table 3 shows the results of combining various algorithms together on several benchmarks. We stopped the algorithms every 2 seconds and computed the ADR using 1000 trials using the current policy.



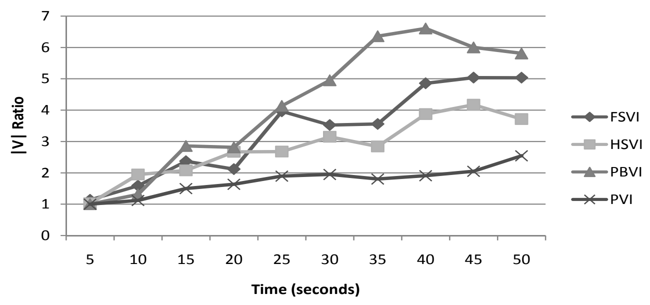


Figure 2: Vector pruning over the Tag Avoid domain.

Since our machine has only 8 cores, we could not exploit the advantages that were presented in the previous experiments, such as using using multiple threads to compute a backup. Therefore, given an unlimited amount of cores, the results would have been much more impressive. We also run a single thread dedicated to pruning dominated algorithms.

When multiple algorithms are executed together we expect two benefits: the convergence should be faster and the policy should be superior. To evaluate the convergence speed we choose a fixed ADR (reported in previous papers) and report the time it took to reach it. To evaluate the best policy, we are also presenting the best ADR that was achieved and the time needed to achieve it.

The results here are interesting. It is clear that in all domains an algorithm can be improved by combining it with another algorithm. However, not every combination of algorithms is superior. This is partially because the more algorithms you execute in parallel the more interaction and synchronization effort is needed.

	TagAvoid			RockSample5,7		
Methods	Time to -6.5	Max ADR	Time Time	Time to 22	Max ADR	Time Time
PBVI (P)	-	-6.7	36	-	18.0	40
PVI (V)	22	-6.1	38	-	18.6	10
HSVI (H)	18	-6.3	40	16	24.2	32
FSVI (F)	22	-6.1	38	12	24.1	36
P;V	14	-5.9	40	18	23.3	40
P;H	16	-6.1	40	14	23.3	26
P;F	16	-6.1	38	8	22.5	32
V;H	12	-6.1	40	6	23.4	14
V;F	16	-6.0	40	4	22.7	8
H;F	20	-6.1	40	4	23.7	30
P;V;H	14	-6.1	40	20	23.8	28
P;V;F	18	-5.8	40	12	24.2	40
P;H;F	18	-6.0	40	20	23.4	26
V;H;F	8	-5.7	30	8	23.8	14
P;V;H;F	26	-6.0	40	10	22.9	22

Table 3: Evaluating combinations of algorithms. For each domain and algorithm we report the time (seconds) it took to reach a predefined ADR, the best ADR achieved by the method, and the time it took to reach the best ADR.

Conclusions

This paper discusses various opportunities for point-based value iteration algorithms for solving POMDPs, in view of the expected increase in the number of processors on a single computer. We have explained how the basic operations of a point-based algorithm, such as the point-based backup can be implemented using threads. We also explained how algorithms such as PBVI and PVI can be enhanced in a multi thread system without changing the algorithm semantics.

We show how to combine different algorithms together producing a hybrid algorithm that converges faster. The concurrent execution also allows us to use a new vector pruning technique, that prunes vectors that do not have a reachable belief witness, and are therefore never used in practice.

We provide experiments over several benchmarks as a proof of concept to our various modifications, and discuss some potential extensions of our research.

References

Cassandra, A. R.; Littman, M. L.; and Zhang, N. 1997. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *UAI'97*, 54–61.

Gomaa, H., and Menascé, D. A. 2000. Design and performance modeling of component interconnection patterns for distributed software architectures. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*.

Littman, M. L.; Cassandra, A. R.; and Kaelbling, L. P. 1995. Learning policies for partially observable environments: Scaling up. In *ICML'95*.

Pineau, J.; Gordon, G.; and Thrun, S. 2003. Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI*.

Shani, G.; Brafman, R.; and Shimony, S. 2006. Prioritizing point-based pomdp solvers. In *ECML*.

Shani, G.; Brafman, R.; and Shimony, S. 2007. Forward search value iteration for pomdps. In *IJCAI-07*.

Smallwood, R., and Sondik, E. 1973. The optimal control of partially observable processes over a finite horizon. *OR* 21.

Smith, T., and Simmons, R. 2005. Point-based pomdp algorithms: Improved analysis and implementation. In *UAI 2005*.

Spaan, M. T. J., and Vlassis, N. 2005. Perseus: Randomized point-based value iteration for POMDPs. *JAIR* 24:195–220.

Sutter, H., and Larus, J. 2005. Software and the concurrency revolution. *Queue* 3(7).

Zhang, N., and Zhang, W. 2001. Speeding up the convergence of value iteration in partially observable markov decision processes. *JAIR* 14:29–51.