Evaluating Point-Based POMDP Solvers on Multi-Core Machines

Guy Shani
Microsoft Research, Redmond, USA
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 mid-sized problems.**

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

In this paper we evaluate several ways in which point-based algorithms can be adapted to parallel computing. We overview the challenges and opportunities and present experimental results providing evidence to the usability of our suggestions.

I. Introduction

Partially Observable Markov Decision Processes (POMDPs) provide a rich modeling framework for agents acting in stochastic environments under partial observability. Such environments require the agent to consider the uncertainty of action outcomes, and to select actions to maximize the expected utility. Models such as Markov Decision Processes (MDPs) can optimize this decision process. However, in an MDP the agent must know all the relevant details of the environment, required to decide on the best action. In most real world domain, such an assumption is not practical. For example, a robot operating in any environment must rely on its sensors in order to compute the current environment state. These sensors can be noisy, and it is also unlikely that the sensors can observe all the needed information.

Due to this partial information over the environment, in many cases it is important to obtain additional information before deciding on the best action. Indeed, POMDPs allow the agent to optimally balance between information gathering and actions that obtain rewards. There are many examples of natural applications of POMDPs, such as decision support agents in medical domains (Hauskrecht & Fraser, 1998), spoken dialog management (Williams, 2008), and autonomous robots (Smith, 2007).

Typically, agents that employ POMDPs use a policy that maps either histories or beliefs to actions. An optimal policy is a policy that maximizes some aspect of the stream of rewards. A popular choice is the infinite sum of discounted rewards that the agents receives when acting in the environment. Due to the difficulty of 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 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 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. Indeed, the Machine Learning community has already begun to realize the potential of multi-core machines (Zanni et al., 2006; Chu et al., 2006).

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.

The point-based update step that lies at the core of all point-based algorithms offer some straight-forward parallelizing opportunities. We begin our paper by reviewing these options and evaluating the leverage that can be gained by such simple modifications.

Sequential algorithms vary in the restrictions they enforce over the ordering of atomic operations. When such restrictions are minimal, the opportunity to parallelize the algorithm increases. We overview a set of point-based algorithms explaining which methods gain the most from the lack of restrictions.

Multi-core machines can also be used to execute several instances of the same algorithm, sharing the same value function. In this case, algorithms that make deterministic decisions are unlikely to provide leverage, as all instances will execute the exact same sequence of operations. We show how to create stochastic versions of the deterministic point-based methods, allowing them to benefit from the parallel execution.

Finally, instead of running several instances of the same algorithms, it is possible to execute a number of different methods together in parallel. Since different algorithms possess different qualities, the synergy of a parallel execution can

speed up the computation of the value function. However, in this case it is possible that the value function growth would become a burden, rather than an advantage. We hence suggest a pruning mechanism that also benefits from the parallel execution.

We provide some partial evaluation of the various modifications suggested here. It is important to note that even if current scaling up due to multi-threaded 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. For example, we cannot currently execute a combination of all of our suggestions on our 16 core 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 some gain by parallelizing the low level point-based backups. Parallelizing algorithms such as PBVI and PVI has shown a better 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.

Our techniques are not directly applicable for distributed computing over networks of machines. In such cases, it is important to reduce the communication overhead over the slow network. It is currently unclear whether our suggested modifications can benefit from the additional computational power but the severe communication restrictions of such networks.

II. Background and Related Work

We begin with an overview of MDPs and POMDPs, the belief space MDP, and how a solution to a POMDP is computed. We then provide an short introduction to point-based methods for solving POMDPs.

A. MDPs, POMDPs and the belief-space MDP

Markov Decision Processes (MDPs) are designed to model autonomous agents, acting in a stochastic environment. Consider for example a robot traveling through a maze. The robot starts at some location and can either move forward, turn left, or turn right. As the robot moves its location may change, and thus, the environment, which includes the location of the robot, changes. The robot must reach some goal state, such as the exit door, or alternatively, collect rewards, such as items that are scattered through the maze.

Formally, an MDP is a tuple $\langle S, A, tr, R\rangle$ where:

- S is the set of all possible world states. In the above example, the environment state is the location and orientation of the robot.
- A is a set of actions the agent can execute. Our robot can only turn left, right, or move forward.
- $tr(s, a, s')$ defines the probability of transitioning from state s to state s' using action a. The transition function models the stochastic nature of the environment, such as the robot attempting to move forward but failing due to engine malfunction or because the wheels where slipping.
- $R(s, a)$ defines a reward the agent receives for executing action a in state s. Action costs can be modeled as negative rewards. In our example the robot receives a reward for getting out of the maze or for collecting an item. The robot may pay a cost each time it moves, modeling the energy loss incurred by the move.

An MDP models an agent acting in an environment where it can directly observe the state it is at.

Realistically, a robot does not know where it is located within a maze. It has sensors that provide observations such as nearby walls. These sensors are imperfect, meaning that they sometimes detect a wall where none exist, and sometimes the sensors fail to detect an existing wall. Now, in order to find its way through the maze the robot must also gather information about the environment state — its own location within the maze.

A Partially Observable Markov Decision Process (POMDP) is designed to model such 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. In the example above the set of observations consists of all possible wall configurations.
- $O(a, s, o)$ is the probability of observing o after executing a and reaching state s, i.e. the sensor model, which incorporates 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 — the belief of the agent over the state space before it has executed or observed anything.

Given a POMDP we can 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. That is, we denote $b' = \tau(b, a, o)$ where:

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

$$pr(o|b, a) = \sum_s b(s)\sum_{s'} tr(s, a, s')O(a, s', o) \tag{2}$$

Therefore, τ is computed in $O(|S|^2)$.

B. Value Functions for POMDPs

Agents that employ POMDPs typically use a policy — a mapping from either histories or beliefs to actions. A POMDP solver computes a policy that maximizes some aspect of

the reward stream, such as the sum of discounted reward — $\sum_{t=0..\infty} \gamma^t r_t$ where r_t is the reward at time t, and $0 < \gamma \leq 1$ is a discount factor, capturing the importance of nearby rewards.

It is well known that the value function V for the belief-space MDP can be represented arbitrarily closely using 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 is defined by associating an action a to each vector α, so that $\alpha \cdot b = \sum_s \alpha(s)b(s)$ represents the value of doing action 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. The policy π_V is immediately derivable using:

$$\pi_V(b) = \text{argmax}_{a:\alpha_a \in V} \alpha_a \cdot b \quad (3)$$

We can compute the value function over the belief-space MDP iteratively:

$$V_{n+1}(b) = \max_a [b \cdot r_a + \gamma \sum_o pr(o|a,b) V_n(\tau(b,a,o))] \quad (4)$$

where $r_a(s) = R(s,a)$ is a vector representation of the reward function. The computation of the next value function $V_{n+1}(b)$ out of the current V_n (Equation 4) is known as a *backup* step. The backup step can be implemented efficiently (Pineau et al., 2003; Spaan & Vlassis, 2005) by:

$$backup(b) = \text{argmax}_{g_a^b : a \in A} \, b \cdot g_a^b \quad (5)$$

$$g_a^b = r_a + \gamma \sum_o \text{argmax}_{g_{a,o}^\alpha : \alpha \in V} \, b \cdot g_{a,o}^\alpha \quad (6)$$

$$g_{a,o}^\alpha(s) = \sum_{s'} O(a,s',o) tr(s,a,s') \alpha(s') \quad (7)$$

Note that the $g_{a,o}^\alpha$ computation (Equation 7) does not depend on the belief state b and can therefore be cached for future backups. All the algorithms we implemented use caching to speed up backup operations. Without caching the $g_{a,o}^\alpha$ results, the backup process takes $O(|S|^2|V||\Omega||A|)$.

While it is possible to execute full backups for V over the entire belief space, hence computing an optimal policy (Cassandra et al., 1997), the operation is computationally hard. Various approximation schemes attempt to decrease the complexity of computation, potentially at the cost of optimality.

A value function can be defined using other representation, such as a direct mapping between belief states and values. Given such a representation we use the H operator, known as the Bellman update, to compute a value function update:

$$Q_V(b,a) = b \cdot r_a + \gamma \sum_o pr(o|a,b) V_n(\tau(b,a,o)) \quad (8)$$

$$HV(b) = \max_a Q_V(b,a) \quad (9)$$

The computation time of the H operator is $O(T_v|S|^2|O||A|)$, where T_v is the time it takes to compute the value of a specific belief point using the value function V.

C. Point Based Value Iteration

Computing an optimal value function over the entire belief space does not seem to be a feasible approach. A possible approximation is to compute an optimal value function over a finite subset B of the belief space (Lovejoy, 1991). Unfortunately, an optimal solution over B does not guarantee optimality over belief points not in B. It is therefore possible that for some reachable belief states (which are not included in B) the resulting value function is sub-optimal. Such a schemes are based on the (empirically verified) assumption, that the computed value function will generalize well for other belief states not included in B.

Point-based algorithms (Pineau et al., 2003; Spaan & Vlassis, 2005; Smith & Simmons, 2005) 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 Point Based Value Iteration (PBVI) algorithm (Pineau et al., 2003), (Algorithm 1), begins with $B = b_0$, and at each iteration computes an optimal value function for the current belief points set. After the value function has converged the belief points set is expanded with all the most distant immediate successors of the previous set. Following Pineau et al. we used the L_2 distance metric in our reported experiments[1].

Given the ever expanding belief space it is clear that at the limit, the belief set B_∞ will cover the entire reachable belief space. Thus, at the limit, PBVI will compute an optimal value function over all reachable beliefs. However, at the limit, the number of α-vectors can also be unbounded, making the point-based backup intractable.

PBVI has a number of shortcomings, not allowing it to scale up to larger domains. First, the belief expansion procedure (Algorithm 3) requires the time consuming computation of distances. Computing a distance between any two belief points requires $|S|$ operations. As we have $|B|$ belief points, and each belief point has $|A||O|$ successors, computing the expanded belief space requires $|B|^2|A||O||S|$ operations. To reduce this computation cost, Pineau et al. also suggest to randomly select a successor for each belief-action pair, reducing the computation to $|B|^2|A||S|$ at the cost of missing distant successors. The value function update phase of PBVI (Algorithm 2) requires a complete backup of all the belief points in the set B in an arbitrary order. Such a backup sequence is time consuming and as we argue later, not all backups are needed.

Algorithm 1 PBVI

1: $B \leftarrow \{b_0\}$
2: **while** true **do**
3: $\quad Improve(V, B)$
4: $\quad B \leftarrow Expand(B)$

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[2] (Algorithm 4) 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.

[1] We also experimented with L_1 and L_{inf} and did not notice any improvement over L_2.

[2] We present here a single value function version of Perseus.

Algorithm 2 Improve(V,B)

Input: V — a value function
Input: B — a set of belief points
repeat
  for each $b \in B$ **do**
    $\alpha \leftarrow backup(b)$
    $add(V, \alpha)$
until V has converged

Algorithm 3 Expand(B)

Input: B — a set of belief points
$B' \leftarrow B$
for each $b \in B$ **do**
  $Successors(b) \leftarrow \{b' | \exists a, \exists o\ b' = \tau(b, a, o)\}$
  $B' \leftarrow B' \cup argmax_{b' \in Successors(b)} dist(B, b')$
return B'

The belief points used by Perseus are very different from the ones used by PBVI and in many cases most of them are redundant. The random walk Perseus uses is however much faster than the belief expansion of PBVI. The value function update may only execute backups over a small subset of the beliefs in B and yet ensures that the value for all points in B improves after each iteration. However, the behavior of Perseus is very stochastic. The random selections cause high variation in performance and in more complicated problems may cause the algorithm to converge very slowly. Nevertheless, the ideas pointed out by Spaan and Vlassis — eliminating the need for complete backups, and computing B rapidly — are an important foundation to our work.

Algorithm 4 Perseus

Input: B — a set of belief points
repeat
  $\tilde{B} \leftarrow B$
  while $\tilde{B} \neq \phi$ **do**
    Choose $b \in \tilde{B}$
    $\alpha \leftarrow backup(b)$
    if $\alpha \cdot b \geq V(b)$ **then**
      $\tilde{B} \leftarrow \{b \in \tilde{B} : \alpha \cdot b < V(b)\}$
      $add(V, \alpha)$
until V has converged

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

$$e(b) = max_a[r_a \cdot b + \gamma \sum_o pr(o|b, a)V(\tau(b, a, o))] - V(b) \quad (10)$$

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 (Algorithm 5) 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.

Algorithm 5 Prioritized Value Iteration

Input: B — a set of belief points
while V has not converged **do**
  $b^* \leftarrow \text{argmax}_{b \in B}\, e(b)$
  $\alpha \leftarrow backup(b^*)$
  $add(V, \alpha)$

Algorithm 6 Choose

Input: B — a set of belief points, k — sample size
$B' \leftarrow B$
$b_{max} \leftarrow nil$
while B' not empty **do**
  for $i = 0$ to k **do**
    Select b with uniform distribution from B' and remove it
    if $e(b) > e(b_{max})$ **then**
      $b_{max} \leftarrow b$
  if $e(b_{max}) > 0$ **then**
    return b_{max}
return nil

Smith and Simmons (2004; 2005) present the Heuristic Search Value Iteration (HSVI - Algorithm 7) that maintains both an upper bound and lower bound over the value function. HSVI traverses the belief space following the upper bound heuristic, greedily selecting successor belief points where the gap between the bounds is the largest, until some stopping criteria has been reached. Afterwards HSVI executes backups and H operator updates over the observed belief points on the explored path in a reversed order.

HSVI is stopped when the gap between bounds over the initial belief state b_0 is reduced to less than ϵ thus providing a guarantee over the quality of the value function. Even though Simth and Simmons prove that the gap is closed in a polynomial number of iterations, in most cases, closing this gap is impractical, especially due to the slow improvement of the upper bound. In practice HSVI computes good policies when the gap is still quite large.

Executing backups in a reversed order is important because the Bellman update uses the values of the successors to update the value of the current belief. As such, the value of a successor must be improved before the value of the current belief can be improved. Indeed, when backups in HSVI are done in order of detection the performance of HSVI is reduced drastically.

HSVI differs considerably from other point-based algorithms. First it collects new belief points after each iteration, as opposed to Perseus that uses a fixed set of points and PBVI that collects more points only if the current set was insufficient to produce a good policy. Second, the points that HSVI collects

depend on the computed value function. As such, while it is possible to combine ideas from Perseus and PBVI, such as collect B following PBVI expansion and update the value function using the Perseus method, such combinations with HSVI are non trivial.

While producing very good trajectories in belief space, the computation of these trajectories is time consuming as it requires the complete expansion of all the successors of every belief state that is visited. Maintaining and updating the upper bound is also time consuming and provides an additional burden on HSVI.

Algorithm 7 HSVI

Initialize $\underline{V}$ and $\bar{V}$
while $\bar{V}(b_0) - \underline{V}(b_0) > \epsilon$ **do**
  $Explore(b_0, \underline{V}, \bar{V})$

Algorithm 8 Explore($b, \underline{V}, \bar{V}$)

Input: a belief state b, upper and lower bounds on the value function $\underline{V}, \bar{V}$.
if $\bar{V}(b) - \underline{V}(b) > \epsilon\gamma^{-t}$ **then**
  $a^* \leftarrow \text{argmax}_a Q_{\bar{V}}(b, a')$ (see Equation 8)
  $o^* \leftarrow \text{argmax}_o(\bar{V}(\tau(b, a, o)) - \underline{V}(\tau(b, a, o))$
  $Explore(\tau(b, a^*, o^*), \underline{V}, \bar{V})$
  $add(\underline{V}, backup(b, \underline{V}))$
  $\bar{V} \leftarrow HV(b)$

Recently, Shani et al. Shani et al. (2007) suggested the Forward Search Value Iteration (FSVI) algorithm. FSVI uses ideas from HSVI, such as traversing the belief space following a heuristic and executing backups in a reversed order. The FSVI heuristic for traversing the belief space relies on an optimal Q function for the underlying MDP. This is a reasonable assumption as solving the underlying MDP is always easier than solving the POMDP. The algorithm simulates a traversal in both the MDP state space and the POMDP belief space, following always the best action for the MDP. As such, the traversal is ensured to minimize the expected number of steps to the goal.

FSVI traversals are very fast to compute, requiring only $|A| + |S| + |O|$ operations for the heuristic computation at each step, compared to the $O(|S|^2)$ operations required just for the belief update. The downside of following an MDP-based heuristic is the inability to create traversals that visit states that may provide important observations, unless these states lie on some path from a start state to a goal, following the MDP policy.

As FSVI trajectories do not depend on the value function it computes, it is possible to break the process into first collecting a set of belief points B, following a number of trajectories and maintaining the successor-predecessor relationship between belief states and after that computing a value function going over the trajectories in B in reverse order.

While in practice such an implementation is not useful, requiring additional memory for remembering the belief points,

Algorithm 9 FSVI

while Policy quality is insufficient **do**
  $B \leftarrow \{b_0\}$
  $b \leftarrow b_0$
  Choose s from the b_0 distribution
  while s is not a goal state **do**
    $a* \leftarrow \text{argmax}_a Q(s, a)$
    Choose s' from the $tr(s, a*, \cdot)$ distribution
    Choose o from the $O(a, s', \cdot)$ distribution
    $b' \leftarrow \tau(b, a*, o)$
    Add b' to B
    $b \leftarrow b'$
    $s \leftarrow s'$
  Execute backups on B in reversed order

this view of FSVI allows us to better compare FSVI to Perseus, PBVI and the new algorithms suggested in this paper.

D. Pruning Dominated Vectors

As the complexity of the backup function relies on the number of α-vectors in the value function, reducing the size of V has an important effect on the computation time of the algorithm. When computing the value of a belief state, we are only concerned with the upper envelope of the hyper-planes. As such, α-vectors that do not take part in this upper envelope are redundant and can be removed from the value function without reducing its accuracy.

The simplest method for pruning α-vectors is when they are pointwise dominated. We say that α is pointwise dominated by α' if for each s, $\alpha(s) \leq \alpha'(s)$. Computing whether a new α-vector is pointwise dominated by any vector in the value function takes $O(|V||S|)$.

However, in many cases a vector may be dominated by a set of vectors, but not by any single one of them. In this case, the pointwise domination gives us no indication of the redundancy of the vector. It is possible to find whether an α-vector belongs to the upper envelope by finding a witness belief point for that vector (Cassandra et al., 1997). Such a witness can be computed using a linear program. However, pruning using linear programming is extremely time consuming and becomes the main bottleneck for computation (Smith & Simmons, 2005).

Some point-based methods offer a natural method for pruning. PBVI optimizes a single vector for each belief state in B. Therefore, PBVI implicitly limits the value function size to $|B|$. The Perseus algorithm attempts to create even more compact value functions, as it does not add new vectors for points that were already improved within the current iterations. In the single, incremental version of the algorithms we present, these benefits no longer hold. For PVI, HSVI and FSVI, that improve a single value function, it is likely that if the same belief state is updated twice, the backup function would output two different α-vectors. Therefore, for incrementally updated value functions, the number of α-vectors in V does no longer depend on the number of encountered belief states.

A possible approach to still prune vectors that are not optimal for any belief point, is to keep track of vectors that

were used within the update process. We can then stop the algorithm after sufficient time has passed and remove all the vectors that were not used (Smith, 2007). The pruning induced by this approach depends strongly on the heuristic for selecting the updated points. For example, it is likely that while the heuristic updates one part of the belief space, other parts remain untouched, and relevant vectors for these parts might be pruned.

E. Multi-Core Machines and Distributed Algorithms

Multi-core machines combine a number of independent processors (cores) in a single machine. Engineers may make a distinction between cores that are embedded on the same integrated circuit die, cores on different dies (sometimes called multi-chip modules), and physically separate CPUs (known as multi-CPU units). For our application, we are interested in multiple cores that share access to the same memory. Therefore, we ignore such distinctions. For example, the machine used in most of our experiments has 2 quad-core units. Each quad-core unit falls under the pure definition of multi-core architecture, while joining two such units falls under the multi-CPU case, but we will refer to such a system as an 8 core machine.

While the cores conduct independent computations, they typically access the same main memory, even though some memory (e.g. cache) may belong to a single processor. Most modern computers use at least 2 cores (dual-core) and in many cases 4 cores (quad-core), but 16 and 32 core machine are also available. Moreover, multi-core technology is predicted to reach hundreds and perhaps even thousands cores in a few years (Asanovic et al., 2006).

General distributed algorithms that are computed in parallel on different machines should limit the amount of costly communications. As data sent between machines must pass through slow networks, the communication cost is in many cases the bottleneck for scaling up such algorithms. Multi-core machines, however, share the same memory. Therefore, algorithms on multi-core machines can communicate without a considerable overhead. It is important, though, that these algorithms will reduce the amount of synchronization, where one instance is waiting for a computation by another instance. It is rarely the case that algorithms that were designed for multi-core parallelism will apply directly to networks of multiple machines.

A recent, yet popular, approach to distributed computing is the map-reduce framework (Dean & Ghemawat, 2004). Map-reduce was designed for executing algorithms where either the input data cannot be stored on a single machine, or where intermediate computations produce data that cannot fit into a single machine. In such cases, the algorithm must first partition the data such that operations can be conducted on single partition independently. Each partition is mapped onto a single machine, where the operation is executed. The results of the computation are then either mapped again into different partitions, or collected and reduced in order to fit into a single machine. In our case, we are not concerned with the size of the data and therefore the map-reduce framework is not applicable for us.

A recent, yet popular, approach to distributed computing is the map-reduce framework (Dean & Ghemawat, 2004). In map-reduce, the algorithm first partitions the data such that operations can be conducted on single partition independently. Each partition is mapped onto a single machine (or single core), where the operation is executed. The results of the computation are then either mapped again into different partitions, or collected and reduced, for example by computing the max or the sum of the results. This approach may be directly applicable to some of the techniques that we suggest below, such as the backup step. However, map-reduce introduces an additional cost in moving data between locations. This relocation of data may be very costly, and we must ensure that this cost does not outweighs the benefit of distributed computing. For example, map-reduce is especially suitable to problems that require the processing of huge datasets, where the data required for the complete operation cannot fit on a single machine. In these cases, we have no choice but to move the data relevant for a single operation to one machine for a computation to be executed.

Another promising technology is the multi-GPU (Graphics Processing Units) machines. GPUs can execute a limited set of operations very rapidly. While these computations are targeted at producing high level graphics, other algorithms can also leverage this computation resource. Indeed, many of the parallel modifications that we suggest, and specifically, the low-level computation of the backup components are suitable for GPU implementation.

III. 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.

Reduction operators — a parallel execution of an operator such as $\sum$ or $\max$ requires the computation of the components of the operator and then combining (reducing) the components into a single answer. There are several different techniques for computing the reduction operator, such as using a binary tree where each two components are reduced over a different thread. In our evaluation, due to the relatively low cost of the reductions, we used a single thread.

A. Synchronized Distributed Computations

1) 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 5 to 7) 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.

2) 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 and improving the value function 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.

3) 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.

B. 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.

1) Multiple Instances of an Algorithm: Another possibility to consider is the execution of multiple instances of the same algorithm at the same time. In this case, algorithms with a stochastic component, such as Perseus (random backup order), PVI (stochastic belief set selection) and FSVI (stochastic trials) can benefit from executing multiple instances at the same time. On the other hand, algorithms such as PBVI and HSVI that make only deterministic selections will gain little or no benefit.

It is possible, in the case of HSVI for example, that one instance has updated the value functions, while a different instance is computing the forward traversal. In this case, the forward traversal might diverge due to the update. However, such changes are not expected to be too beneficial.

In this case, making stochastic variations of deterministic algorithms can have appealing properties. In the case of PBVI, Pineau et al. already suggested to expand the current belief set B by selecting only a single random action for each belief $b \in B$, attempting to reduce the computation time of the belief set expansion. When using multiple instances, selecting random actions will cause different instances of PBVI to operate using different belief sets, thus providing better coverage of the belief space.

A similar adaptation for HSVI is less obvious. HSVI relies on the current value function bounds to create the belief space traversals. As such, choosing a different heuristic, such as randomizing the action selection does not seem justified. However, HSVI converges once all beliefs are finished. Usually, a single update to a node does not make it finished, meaning that the gap between bounds at that point should still be reduced. HSVI chooses to leave such nodes unfinished, until a later trial would reach them again. We suggest to randomly (e.g. when a uniform random number falls below some exploration threshold) decide whether an unfinished node should be left unfinished, or whether we should attempt again to reduce the gap using another traversal. As such, our algorithm stochastically branches in unfinished nodes, creating different traversals in different instances.

The reason this approach is not attractive in non-parallel settings, is that it is likely that the propagation of values to the initial belief state would take longer. Thus, the anytime performance of the algorithm is reduced. In a multiple algorithm setting, we can run one regular HSVI instance that always propagates values to the initial belief state, and allow exploration only in the other instances, thus retaining the anytime properties of the original algorithm.

C. 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.

We offer a simple 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 approach is similar to the the usage-based pruning suggested by Smith (2007). However, our pruning does not depend on the value function computation heuristic, but on the execution of the current policy. As different trials will reach different parts of the environment, given the stochastic properties of the domain, it is unlikely that an important vector would never be used.

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.

IV. Empirical Evaluations

We ran several sets of tests in order to provide evidences to the various multi-threaded enhancements we suggest. 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. Most of our experiments (except for the experiments in Section V-B2) were done on a dual quad-core machine (8 processors) with 2.66GHz processors. We use Java and the JVM is limited to 1.5GB of RAM.

A. Results

1) 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 6), requiring $|A|$ threads only at a time. We also tried further splitting the tasks so that each $g_{a,o}^{\alpha}$ (Equation 7) is computed on a different thread. As expected, this has reduced the performance.

We experimented over four domains — Hallway, Hallway2 (Littman et al., 1995), TagAvoid (Pineau et al., 2003) and RockSample (Smith & Simmons, 2005). Over these four domains we compare PBVI on a single thread and on multiple threads. PBVI was executed for a fixed number of iterations and we measure the average time (milliseconds) of backup executions. We executed each experiment 5 times and report the average result. Standard deviations were extremely small — less than 0.01. Table I (left column) compares single (S) and multi (M) thread backup execution time (milliseconds) for the various domains. The number of backups that were used changes between domains but is always more than 5000 during each execution.

As the semantics of the algorithm do not change, the exact same value function is computed, and there is no need to

compare the quality of the value functions. Also, PBVI was used here because it executes a relatively large number of backups, but the speedup of the backup process using threads is applicable to any point-based algorithm.

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

TABLE I
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 $\times$ 7	672	162	2069	307

TABLE II
COMPARING SINGLE (S) THREAD AND 8 THREADED (M) EXECUTION TIME (SECONDS) OF PBVI AND PVI.

The results for TagAvoid are rather surprising — splitting the backup process into tasks has helped very little. However a closer look at the Tag Avoid domain can explain this. In this domain 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 further splitting of the operation useless.

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.

2) PBVI: As we explained above, 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 this process over the same 4 domains, and report average time over 5 executions. In order to separate the contributions of the various modifications we suggest, we evaluated the changes to the algorithms using the original, single threaded point-based backup. Table II (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. However, even though we have used 8 threads, the speedup is only about 50%. It is well known that multi-threaded implementation rarely achieves a linear improvement. In our case, the needed synchronization steps for collecting the results from the threads after each value function update or belief set expansion reduces the advantage of multi-threading.

Also, 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.

3) PVI: In PVI the computation of the Bellman error can be divided into different threads for different belief points. Shani et al. (2006) suggest to reduce this computation by sampling a subset of the belief set B, but we report results here only over a full update of the Bellman errors. Once we understand the advantage of computing the error in different threads, we can re-introduce sampling and allow PVI to use much larger belief sets. PVI was executed over a belief set of 1000 points for 150 iterations. Table II (right column) compares single (S) and multi (M) thread execution time (seconds) for the various domains.

The results here are more encouraging. PVI speedup is much 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.

4) Vector Pruning: Next, we look at our pruning approach. 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.

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.

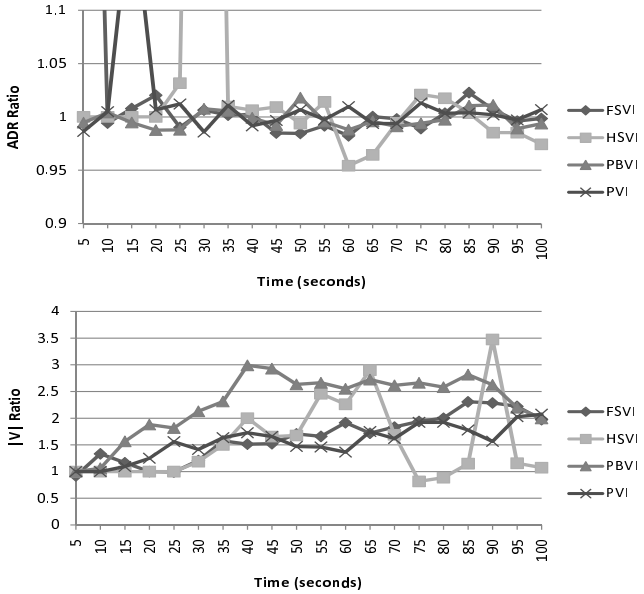


Fig. 1. Vector pruning over the Hallway domain.

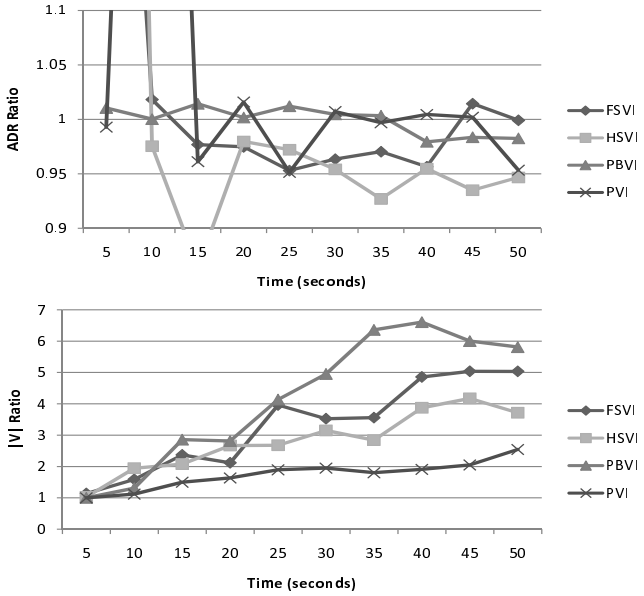


Fig. 2. Vector pruning over the Tag Avoid domain.

B. Multiple Instances of an Algorithm

We continue to evaluate the execution of a number of instances of the same algorithm on a number of domains. Table III shows our results over a set of benchmarks. On each benchmark we ran 1 to 6 instances of the algorithms FSVI, PVI, PBVI, PBVIR (PBVI with random action expansion), HSVI and HSVIR (HSVI with random exploration) on all benchmarks. All executions also dedicated one thread to the vector pruning technique we suggested. The algorithms were stopped after each second and we compute the current ADR over 1500 trials.

For each set of instances of the same algorithm we report the average ADR and standard deviation during the execution. The average ADR is appropriate here since it encapsulates both the convergence speed and the quality and stability of the ADR.

As expected, Table III clearly shows how HSVI and PBVI gain nothing from multiple instances. A single HSVIR instance usually performs worse than a single HSVI instance, since it takes more time to propagate the values back to the initial belief state. However, all stochastic algorithms (FSVI, HSVIR, PVI, PBVIR) gain additional leverage from running multiple instances. This gain is rapidly reduced when too many instances are used. This is mainly due to the need for some synchronization through the shared value function. Future research should focus on reducing this needed synchronization, thus allowing these algorithms to fully present their powers.

	Hallway	Hallway2	Tag	RS5 $\times$ 7
PVI				
$\times 1$	0.486(0.02)	0.310(0.012)	-7.0(0.6)	17.1(2.1)
$\times 2$	0.491(0.03)	0.317(0.013)	-6.8(0.5)	18.2(1.2)
$\times 3$	0.505(0.006)	0.322(0.010)	-6.7(0.4)	19.0(0.9)
FSVI				
$\times 1$	0.504(0.01)	0.319(0.007)	-8.5(1.4)	20.2(2.1)
$\times 2$	0.505(0.008)	0.320(0.006)	-7.7(1.2)	20.5(2.8)
$\times 3$	0.511(0.008)	0.324(0.008)	-7.7(0.6)	20.8(1.1)
HSVI				
$\times 1$	0.458(0.04)	0.275(0.06)	-8.8(2.6)	20.5(3.7)
$\times 2$	0.426(0.08)	0.251(0.08)	-10.3(3.4)	19.3(3.6)
HSVIR				
$\times 1$	0.467(0.06)	0.287(0.01)	-8.5(2.9)	18.9(5.4)
$\times 2$	0.489(0.02)	0.291(0.01)	-8.0(3.3)	19.8(5.2)
$\times 3$	0.498(0.02)	0.308(0.01)	-7.8(1.7)	20.9(3.2)
PBVI				
$\times 1$	0.499(0.02)	0.319(0.009)	-7.6(0.7)	16.3(2.2)
$\times 2$	0.495(0.02)	0.313(0.01)	-7.9(1.3)	15.6(1.5)
PBVIR				
$\times 1$	0.495(0.02)	0.308(0.02)	-7.3(1.2)	15.9(1.9)
$\times 2$	0.508(0.01)	0.313(0.01)	-7.1(1.6)	16.4(0.4)
$\times 3$	0.512(0.006)	0.323(0.01)	-7.0(0.8)	17.0(0.1)

TABLE III
EVALUATING MULTIPLE INSTANCES OF THE SAME ALGORITHMS. WE REPORT AVERAGE ADR AND STANDARD DEVIATION OVER THE FIRST 25 SECONDS OF EXECUTION. WE REPORT RESULTS ONLY FOR 2-3 CONCURRENT INSTANCES ENTRIES, SINCE MORE INSTANCES DID NOT ACHIEVE SIGNIFICANT IMPROVEMENT.

1) Combining Algorithms: Next, we tested the ability to combine together several algorithms. Table IV shows the results of combining various algorithms together. We ran combinations of FSVI (F), PBVI (P), HSVI (H) and PVI (V) on several benchmarks. We stopped the algorithms every 2 seconds and computed the ADR using 1000 trials using the current policy. The executions was stopped after 40 seconds for the first two domains and after 100 seconds for the Rock Sample 7×8 domain.

These experiments were executed on an 8 core machine. Therefore, we did not apply other modifications, such as using multiple threads to compute the Bellman error for PVI, or using multiple threads to compute a backup. We used a single thread version of each algorithm in the experiments below. Therefore, given an unlimited amount of cores, we can expect significant benefits.

Aside from the algorithms we also run a single thread dedicated to pruning dominated algorithms. We ran some experiments without the pruning thread but the combined algorithms run out of memory after a short while.

	TagAvoid			RockSample5 × 7			RockSample7 × 8		
Algorithms	Time to -6.5	Max ADR	Time	Time to 22	Max ADR	Time	Time to 19.0	Max ADR	Time
PBVI (P)	-	-6.7	36	-	18.0	40	-	15.4	80
PVI (V)	22	-6.1	38	-	18.6	10	-	17.4	50
HSVI (H)	18	-6.3	40	16	24.2	32	20	19.8	100
FSVI (F)	22	-6.1	38	12	24.1	36	40	19.1	40
P;V	14	-5.9	40	18	23.3	40	90	19.5	90
P;H	16	-6.1	40	14	23.3	26	30	19.8	30
P;F	16	-6.1	38	8	22.5	32	100	19.1	100
V;H	12	-6.1	40	6	23.4	14	20	20.4	90
V;F	16	-6.0	40	4	22.7	8	60	20.5	60
H;F	20	-6.1	40	4	23.7	30	30	20.4	30
P;V;H	14	-6.1	40	20	23.8	28	70	20.5	100
P;V;F	18	-5.8	40	12	24.2	40	-	18.4	40
P;H;F	18	-6.0	40	20	23.4	26	30	19.8	100
V;H;F	8	-5.7	30	8	23.8	14	50	19.5	80
P;V;H;F	26	-6.0	40	10	22.9	22	40	20.8	80

TABLE IV
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.

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.

To further understand the behavior of multiple algorithms executed together, we collected various statistics from the execution of FSVI, HSVI, PBVI, and PVI together on the RockSample 7×8 domain[3]. In this experiment we stopped the execution every 10 seconds, computed the ADR and collected vector usage statistics; For each α-vector that was generated we remember the algorithm that generated it. Thus, we are able to gain understanding of the contribution of various algorithms to the value function computation.

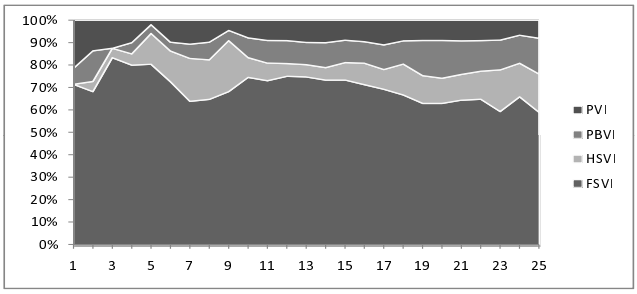
First, we counted the number of vectors that were used in policy execution (Figure 3(a)). While vectors generated by FSVI dominate the policy, all algorithms contribute vectors to the policy. To check whether all algorithms generated important vectors, we tried executing policies that exclude vectors generated by a single algorithm, and the ADR was reduced considerably. Also, while FSVI dominates the value function and policy, when only vectors computed by FSVI were executed, ADR dropped by about 25%.

It is also interesting to consider the number of times that vectors from an algorithm were used during policy evaluation (Figure 3(b)), emphasizing 'important' vectors that are more heavily used. Indeed, vectors generated by PBVI are used more often than vectors from other algorithms. To understand this, we computed when were vectors used during trials of policy execution. For each vector, we computed the average step in the trial where vectors were used (Figure 3(c)). Vectors from PBVI are mainly used in the beginning (initial steps) of a trial. This is because when a trial begins, the agent always starts with the same behavior. In the case of RockSample, the agent always begins by moving within range of the closest rock and sensing whether it is "interesting". Therefore, the same vectors are used in all trials. In the latter steps of trials, the needed behavior is more diverse, given the rock states. Thus, each trial begins by selecting the same α-vectors, but afterwards, different vectors are used in different trials.

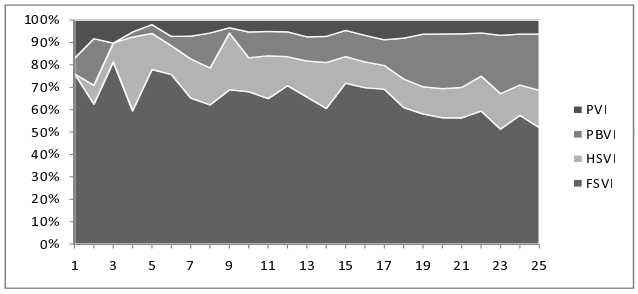
PVI vectors are used closer to the end of trials. This is because PVI executes backups on vectors with a high Bellman error. Such vectors initially lie near the rewards of the domain, and thus, closer to the end of trials. We can view PBVI as doing a breadth first search from the initial belief state, and PVI as doing a breadth first search backwards from the goal states. FSVI and HSVI, which are both trial based, are depth first search algorithms. Each algorithm contributes differently — PVI provides good values for beliefs near the end of trials, FSVI and HSVI help to propagate these values down to beliefs that are close to the initial belief state, and PBVI computes good values for these initial beliefs. In general FSVI vectors are used later in the trial than HSVI vectors. This is because FSVI executes goal oriented trials, while HSVI usually generates less direct trials that explore more beliefs, producing more vectors that are important in earlier stages.

Another interesting aspect is the influence of the algorithms on the evolution of the value function (Figure 3(d) and Figure 3(e)). We can measure this by looking at vectors that were selected in the point-based backup process (Equation 6). While in the beginning, vectors generated by FSVI dominate the selected vectors, later on vectors generated by PBVI become the dominating factor. This can be seen as an evidence that the belief expansion policy of PBVI indeed collects many important beliefs. Vectors generated by FSVI gradually become less influential, because FSVI executes focused trials, backing up only vectors that are found on the path. However, the backup process requires that we select one α-vector for each successor of the current belief. Thus, it is also important

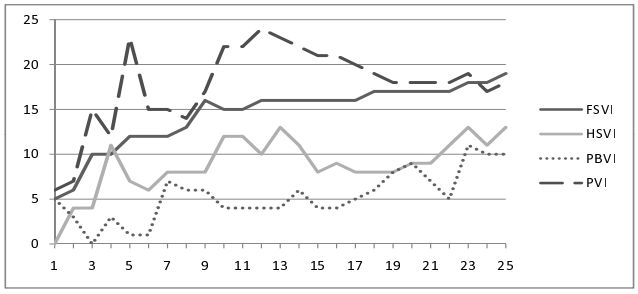
[3]We did not use the pruning thread here.



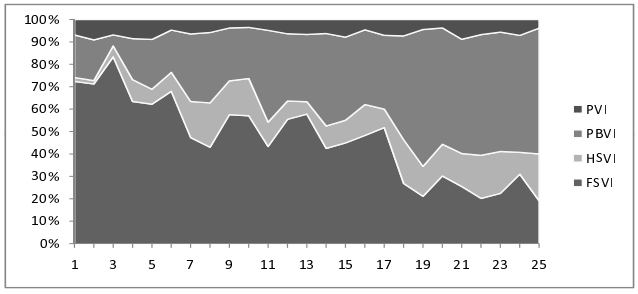
(a) Number of vectors used in the policy



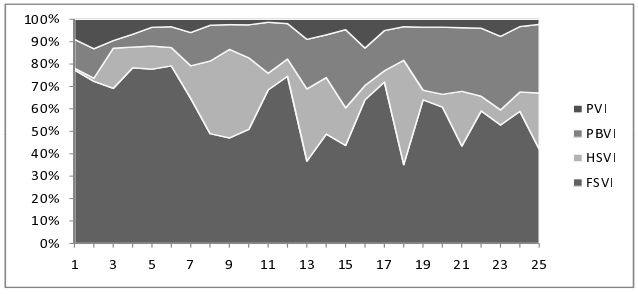
(b) Number of times vectors where used in policy execution



(c) Average trial step where vector was used in policy execution



(d) Number of vectors used by backup operations



(e) Number of times vectors were used in backups

Fig. 3. Analyzing the combination of FSVI, HSVI, PBVI, and PVI over the RockSample 7×8 domain. In all the above graphs, the x axis is time in seconds $\times 10$.

to backup these successors, in order to have good vectors for them. Here, the heuristic of PBVI, that is not value oriented, helps it again to collect important beliefs for value function computations.

2) Large Domains: Finally, we run experiments on 3 large domains — Network Management, Rock Sample, and Logistics. These models are naturally formalized as factored POMDPs (Shani et al., 2008) and allow us to scale up to large problems. We follow the model definitions of Shani et al., but used a flat definition, in order to scale up the computational effort, rather than the memory requirements. We run the experiments over machine with 16 2.8GHz AMD Opteron CPUs.

For all domains we ran HSVI, FSVI, PVI, and PBVI together, allowing all algorithm to use the parallel computation of backups. Also, PVI and PBVI were allowed 4 threads each for a parallel implementation. Table V shows the properties of the domain, the maximal ADR that was achieved within 2500 seconds, and the time until this ADR was achieved. In comparison, we tried to solve the problems using our fastest algorithm, FSVI, and a single thread. Even after 5 hours, FSVI was unable to find a policy with comparable performance to the one computed using the combination of all algorithms in less than 45 minutes.

Domain	$\|S\|$	$\|O\|$	$\|A\|$	ADR	Time (secs)
Network 12	2^{12}	2	25	56.191	1700
Rock Sample $8 \times 8 \times 14$	2^{20}	2	19	53.27	1900
Logistics $4 \times 2 \times 5$	2^{19}	7	18	21.65	2800

TABLE V
EXECUTING ALGORITHMS IN PARALLEL ON A 16 CORE MACHINE.

V. DISCUSSION AND RELATED WORK

The machine learning community has previously acknowledged the possible scaling up using multi-core architecture. This was mostly explored in the context of Support Vector Machines (e.g. (Zanni et al., 2006)). The parallel PBVI and PVI algorithms are variations of this approach, but the rest of the modifications that we suggest do not fall under this framework.

In view of this, we propose several enhancements to point-based methods, that exploit the structure point-based methods. Our pruning method is such an example, and the combinations of different algorithms is another enhancement that doesn't fit the map-reduce framework.

Our algorithm combination allows us to exploit the advantages of various algorithms. For example, we are now able to combine the rapid improvements that FSVI offers with the convergence guarantees of HSVI. It is also likely that in different domains different algorithms have an advantage. In such cases, a combination of algorithms will allow us to always use the best algorithm for the domain. When combining algorithms the number of α vector is likely to increase more rapidly, using our pruning technique is important.

When designing multi thread algorithms, it is important to reduce the amount of synchronization. Each point of thread

synchronization reduces the performance of the algorithm considerably (Sutter & Larus, 2005). The modifications we suggest that preserve the algorithm semantics need synchronization only in that all threads must finish an operation before a new operation is launched. However, the threads do not synchronize any operations while running.

Most of the modifications we suggest require a shared value function. Many of the operations over the value function read vectors, thus not requiring any synchronization. However, changes to the value function are also needed, such as adding a new vector or pruning dominated vectors. Our implementation minimizes the use of locks when changes are required. We use a type of transactional operation (Sutter & Larus, 2005) where changes are done over a copy of the value function, that then replaces the original. At the worst, an algorithm might use vectors that were already found to be dominated, but are nevertheless valid candidates. This will slow down an algorithm, as such vectors can be ignored, but will not compromise the validity of the value function.

The changes we suggest here are also applicable only to multi-core machines where threads share their memory. A different type of parallel computing is the cluster framework, where several machines are combined in a single cluster. Clusters offer thousands of processors to use, but add an additional overhead in communication between the machines. In that context our modifications are likely to be less useful, and future research should identify how point-based solvers can still benefit from multiple cores while minimizing communications. Our algorithms require a shared value function and the communication cost of sending α-vectors between machines reduces the effectiveness of our modifications considerably.

In this paper we focus on exploiting the properties of point-based algorithms for parallel execution. An orthogonal approach is to use the properties of the domain. For example, some domains can be decomposed into a set of loosely coupled small POMDPs arranged in an hierarchy (Charlin et al., 2007). Then, we can solve all the sub-POMDPs in parallel and combine the resulting policies afterwards. Still, when computing policies for the sub-POMDPs, we can still leverage improvements at the algorithmic level to speed up execution.

Boosting (Schapire, 1999) is another relevant method from the Machine Learning literature, where several instances of the same algorithm are executed, and the combined solution is better than the results of each separated algorithm. Boosting usually executes the algorithms in a sequential order, where each algorithm focuses on the data where the previous algorithms did not do as well. Indeed, it is difficult to consider running a set of algorithms (instances of the same algorithm or different algorithms) in parallel, as we do, to be a boosting method. Nevertheless, we can borrow ideas from boosting by directing different instances of the algorithms towards different areas of the belief space. For example, we can manipulate the heuristic of HSVI or FSVI to direct the belief space traversals towards different areas, thus reducing the number of overlapping backups for the same belief state. Additional research is required in order to identify good strategies for traversal heuristics, and to evaluate the advantage from using such an approach.

VI. Conclusions

This paper evaluates 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.

In the future, we intend to test our approaches on machines with more cores, such as the AMD FireStorm. On such machines, where all the parallel modifications can be evaluated together, we expect that the scaling up will be much more pronounced. We also intend to try our methods on machines with multiple GPUs, which are now becoming widely available. It is likely that not all of our suggestions will be applicable to GPUs, that support a smaller set of instructions, but this requires further investigation. We will first try to use GPUs for the low-level backup parallelizing, which contains arithmetic operations that are especially suitable for GPUs.

After fully understanding the usability of our approaches to multi-core machines with shared memory, the next step would be to design algorithms that can be executed in distributed environments, using techniques such as map-reduce, or cloud computing, to scale up POMDP solvers. While more challenging, distributed environments can offer many more computational power, at the cost of moving data through a network. This tradeoff should be studied and we intend to design algorithms that will optimize for this tradeoff.

References

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., & Yelick, K. A. (2006). *The landscape of parallel computing research: A view from berkeley* (Technical Report UCB/EECS-2006-183). EECS Department, University of California, Berkeley.

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

Charlin, L., Poupart, P., & Shioda, R. (2007). Automated hierarchy discovery for planning in partially observable environments. *Advances in Neural Information Processing Systems NIPS 19* (pp. 225–232).

Chu, C., Kim, S. K., Lin, Y., Yu, Y., Bradski, G. R., Ng, A. Y., & Olukotun, K. (2006). Map-reduce for machine learning on multicore. *NIPS* (pp. 281–288).

Dean, J., & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*.

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

Hauskrecht, M., & Fraser, H. (1998). Planning medical therapy using partially observable markov decision processes. *ninth international workshop on principles of diagnosis , (DX-98)*.

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

Lovejoy, W. S. (1991). Computationally feasible bounds for partially observable markov decison processes. *Operations Research*, *39*, 175–192.

Pineau, J., Gordon, G., & Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. *IJCAI*.

Schapire, R. E. (1999). A brief introduction to boosting. *Sixteenth International Joint Conference on Artificial Intelligence, (IJCAI)*.

Shani, G., Brafman, R., Poupart, P., & Shimony, S. (2008). Efficient add operations for factored pomdps. *ICAPS-08*.

Shani, G., Brafman, R., & Shimony, S. (2006). Prioritizing point-based pomdp solvers. *ECML*.

Shani, G., Brafman, R., & Shimony, S. (2007). Forward search value iteration for pomdps. *IJCAI-07*.

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

Smith, T. (2007). *Probabilistic planning for robotic exploration*. Doctoral dissertation, The Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Smith, T., & Simmons, R. (2004). Heuristic search value iteration for pomdps. *UAI 2004*.

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

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

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

Williams, J. D. (2008). The best of both worlds: Unifying conventional dialog systems and pomdps. *Intl Conf on Speech and Language Processing (ICSLP)*.

Zanni, L., Serafini, T., & Zanghirati, G. (2006). Parallel software for training large scale support vector machines on multiprocessor systems. *JMLR*, *7*.