A Parallel Algorithm for POMDP Solution

Larry D. Pyeatt[1] and Adele E. Howe[2]

[1] Computer Science Department, Texas Tech University, Lubbock, TX 79409
pyeatt@cs.ttu.edu
[2] Computer Science Department, Colorado State University, Fort Collins, CO 80523
howe@cs.colostate.edu

Abstract. Most exact algorithms for solving partially observable Markov decision processes (POMDPs) are based on a form of dynamic programming in which a piecewise-linear and convex representation of the value function is updated at every iteration to more accurately approximate the true value function. However, the process is computationally expensive, thus limiting the practical application of POMDPs in planning. To address this current limitation, we present a parallel distributed algorithm based on the Restricted Region method proposed by Cassandra, Littman and Zhang [1]. We compare performance of the parallel algorithm against a serial implementation Restricted Region.

1 Introduction

The process of planning can be viewed as finding a *policy* for a sequential decision task. The policy is a mapping from states to actions that maximizes some reward function. For plan execution in this framework, the agent determines its current state and chooses an action from its policy at each time step. The effects of actions need not be deterministic, since the current state is all that is necessary to select an action from the policy. If the current state can be determined at each time step, then the planning problem can be modeled as a Markov decision process. However, the agent may not be equipped to determine the current state with certainty and can only make sensor observations that hint at the true state. In this case, the planning problem is more closely modeled as a partially observable Markov decision process (POMDP) [13, 2].

A POMDP formulation of the planning problem is particularly well suited to mobile robotics, where the environment can be modeled as a state space, but the sensors may be inadequate to determine the current state. There are several methods for exactly solving a POMDP problem to find a policy, but current solution methods do not scale well and are practical only for small problems. This paper presents a parallel version of Restricted Region (RR), which is currently the best known exact solution algorithm for POMDP problems [13, 1].

A POMDP problem is specified by a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a set of observations $\mathcal{Z}$, a reward function $r(s,a)$, a set of transition probabilities $\Pr(s'|s,a)$, and a set of observation probabilities $\Pr(z|s',a)$. The transition probabilities specify the probability that the state is s' given that the previous state was s and action a was taken. The observation probabilities specify the probability of seeing observation z given that the current state is s' and action a was taken in the previous state.

The POMDP model assumes that true state cannot always be determined. Instead, a probability distribution b over $\mathcal{S}$ is maintained. This probability distribution is known as the *belief state*. At each time step an action a is taken, and an observation z is received. The belief in state s' is then updated using

$$b_z^a(s') = \frac{\Pr(z|s',a)\sum_{s\in\mathcal{S}}\Pr(s'|s,a)b(s)}{\sum_{s'\in\mathcal{S}}\left[\Pr\left(z|s',a\right)\sum_{s\in\mathcal{S}}\Pr(s'|s,a)b(s)\right]} \tag{1}$$

for all $s' \in \mathcal{S}$, and the resulting new belief state is used to select the next action from the policy. An observation is received and the process repeats.

In the next section, we will review the best known algorithm for finding POMDP policies. Section 3 describes our parallel version of the algorithm. In Section 4, we present our experiments and results, and Section 5 discusses future work.

2 Solving POMDPs Using Incremental Pruning

The goal for a POMDP solution method is to construct a policy π that specifies the action to take for every possible belief state $b \in \mathcal{B}$ so as to maximize discounted future rewards. Most POMDP solution algorithms generate the policy by finding the value function, which specifies the value of taking each possible action from every belief state. The value function can always be represented as a piecewise linear and convex surface over the belief space [11, 7]. A POMDP policy is a set of labeled vectors that are the coefficients of the linear segments that make up the value function.

In value-iteration algorithms, a new value function $\mathcal{V}'$ is calculated in terms of the current value function $\mathcal{V}$ at each iteration [1]. $\mathcal{V}'$ is the result of adding one more step of rewards to $\mathcal{V}$. The function $\mathcal{V}'$ is defined by:

$$\mathcal{V}'(b) = \max_{a \in \mathcal{A}} \left(\sum_{s \in \mathcal{S}} r(s,a)b(s) + \gamma \sum_{z \in \mathcal{Z}} \Pr(z|b,a)\mathcal{V}(b_z^a) \right). \tag{2}$$

Equation 2 defines the value for a belief state b as the value of the best action that can be taken from b where the best value is given by the expected immediate reward for that action plus the expected discounted value of the resulting belief state.

Equation 2 can be decomposed into combinations of simpler value functions:

$$\mathcal{V}'(b) = \max_{a \in \mathcal{A}} \mathcal{V}^a(b) \tag{3}$$

$$\mathcal{V}^a(b) = \sum_{z \in \mathcal{Z}} \mathcal{V}_z^a(b) \tag{4}$$

$$\mathcal{V}_z^a(b) = \frac{\sum_{s \in \mathcal{S}} r(s,a)b(s)}{|\mathcal{Z}|} + \gamma \Pr(z|b,a)\mathcal{V}(b_z^a). \tag{5}$$

Because these functions involve linear combinations, the transformations preserve piecewise linearity and convexity [11, 4].

Since the value functions $\mathcal{V}_z^a$, $\mathcal{V}^a$, and $\mathcal{V}'$ are PWL, we can represent them as sets of vectors composed of the coefficients for each PWL segment. The sets have a unique minimum size representation [4], denoted by S_z^a, S^a, and S' respectively. We use the notation $\mathsf{purge}(\mathcal{X})$ to represent the operation of reducing a set $\mathcal{X}$ to its minimum size form.

A vector of all ones is denoted as 1, a vector of all zeros is denoted as 0, and we define a vector e_s for every $s \in \mathcal{S}$ where e_s is a vector of all zeros except for one element $e_s(s) = 1$. The equivalence of vectors is defined as $\alpha = \beta \iff \alpha(s) = \beta(s) \, \forall \, s \in \mathcal{S}$. The lexicographical ordering of vectors is defined as $\alpha > \beta \iff \alpha \neq \beta$ and $\alpha(s) > \beta(s)$ where $s = \operatorname{argmin}_{i \in \mathcal{S}} \alpha(i) \neq \beta(i)$. Set subtraction is defined by

$\mathcal{X} \backslash \mathcal{Y} = \{\chi \in \mathcal{X} | \chi \notin \mathcal{Y}\}$. The *cross sum* of two vector sets is defined as $\mathcal{X} \oplus \mathcal{Y} = \{\chi + \psi | \chi \in \mathcal{X}, \psi \in \mathcal{Y}\}$.

Using the vector set notation, the value function is described as

$$S' = \mathsf{purge}\left(\bigcup_{\alpha \in \mathcal{A}} S^a\right) \tag{6}$$

$$S^a = \mathsf{purge}\left(\bigoplus_{z \in \mathcal{Z}} S^a_z\right) \tag{7}$$

$$S^a_z = \mathsf{purge}\left(\{\tau(\alpha, a, z) | \alpha \in S\}\right) \tag{8}$$

where $\tau(\alpha, a, z)$ is the $|\mathcal{S}|$-vector given by

$$\tau(\alpha, a, z) = \frac{1}{|\mathcal{Z}|} r^a(s) + \gamma \sum_{s'} \alpha(s') \Pr(z|s', a) \Pr(s'|s, a). \tag{9}$$

Equations 6 and 7 come directly from Equations 3 and 4 and basic properties of piecewise linear convex functions. Equation 8 is derived by substituting Equation 1 into Equation 5. Calculating $\tau(\alpha, a, z)$, performing the cross sum, and performing the union are straightforward; so the problem is now reduced to purging sets of vectors. In order to get a solution quickly, the purge operation must be efficient.

2.1 Purging Vector Sets

Given a set of $|\mathcal{S}|$-vectors A and a single $|\mathcal{S}|$-vector α, we define the set of belief states in which vector α is dominant compared to all the other vectors in A as

$$R(\alpha, A) = \{b | b \cdot \alpha > b \cdot \alpha', \forall \alpha' \in A \backslash \{\alpha\}\}. \tag{10}$$

The set $R(\alpha, A)$ is called the *witness region* of vector α because b can testify that α is needed to represent the piecewise linear convex function given by $A \cup \{\alpha\}$. The witness region is empty if and only if there is no part of the belief space where α has a better value than any other vector in A.

Given this definition for the witness region R, we can define the purge function as

$$\mathsf{purge}(A) = \{\alpha | \alpha \in A, R(\alpha, A) \neq \emptyset\}, \tag{11}$$

```
purge (F,S)
    W ← ∅
    – Select the obvious winners at each corner of the belief space and put them in W
    for s ∈ S
        max ← 0
        for φ ∈ F
            if ((e_s · φ > max) or ((e_s · φ = max) and (ω_s < φ))) then
                ω_s ← φ
                max ← e_s · φ
    for s ∈ S
        if (ω_s ∈ F) then
            W ← W ∪ {ω_s}
            F ← F\{ω_s}
    – Test the remaining vectors for domination
    while F ≠ ∅
        φ ← first(F)
        x ← dominate(φ, W)
        if (x = ⊥) then F ← F\{φ}
        else
            max ← 0
            for (φ ∈ F)
                if ((x · φ > max) or ((x · φ = max) and (ω < φ))) then
                    ω ← φ
                    max ← x · φ
            W ← W ∪ {ω}
            F ← F\{ω}
    return W
```

Fig. 1. The algorithm for purging a set of vectors.

which simply returns the set of vectors in A that have non-empty witness regions. This is the minimum size set that represents the piecewise linear convex value function.

Figure 1 shows the algorithm of purge(F). This operation, developed by White [12], returns the vectors in F that do not have empty witness regions. It should be noted that dot product ties *must* be broken lexicographically [4]. The purge algorithm starts with an empty set, W, and fills it with vectors from F that have non-empty witness regions. The algorithm first checks all of the vectors at the corners of belief space. The dominant vector at each corner is obviously dominant over some region, so these vectors are added to W and removed from F immediately. The algorithm then checks each vector ϕ in F to determine if it is dominant somewhere in W. If a point x is found on vector ϕ that dominates all the vectors in W, then x is a *witness* to the fact that there is another vector that belongs in W. The correct vector $\phi \in F$ is the one that maximizes the dot product with x. The vector ϕ is added to W and removed from F, and the loop repeats. When F is empty, W contains the parsimonious set of dominant vectors.

The dominate function called by purge finds the belief state x that gives the largest value on ϕ that is not dominated. The dominate function creates a linear program and adds a constraint for each hyperplane in W. If no solution exists or if the value of the objective function is non-positive, then a null vector is returned. Otherwise, the solution to the linear program is the desired point x.

2.2 Incremental Pruning

Incremental pruning [1] sequences the purge operations to reduce the number of linear programs that have to be solved and to reduce the number of constraints in the linear programs themselves. Incremental pruning can exhibit superior performance and greatly reduced complexity compared to other methods [13, 1].

From the associative property of addition, $(A \oplus B) \oplus C = A \oplus (B \oplus C)$, and it also holds that $\mathsf{purge}((A \oplus B) \oplus C) = \mathsf{purge}(\mathsf{purge}(A \oplus B) \oplus C)$. Therefore, Equation 7 can be rewritten as

$$S^a = \mathsf{purge}(\ldots \mathsf{purge}(\mathsf{purge}(S^a_{z_1} \oplus S^a_{z_2}) \oplus S^a_{z_3}) \ldots \oplus S^a_{z_{|\mathcal{Z}|}}), \qquad (12)$$

which is the approach taken by the incremental pruning algorithm. All of the calls to purge in Incremental Pruning are of the form purge $(A \oplus B)$. This formulation allows purge to exploit the considerable regularity in the vector set passed to it.

We replace the call to dominate in Figure 1 with

$$x \leftarrow \mathsf{dominate}(\phi, D \backslash \{\phi\}) \qquad (13)$$

where D is a set of vectors that satisfy the following set of properties:

1. $D \subseteq (A \oplus B)$
2. Let $\alpha + \beta = \phi$ for $\alpha \in A$ and $\beta \in B$. For every $\alpha' \in A$ and $\beta' \in B$, if $\alpha' + \beta' \in W$, then either $\alpha' + \beta' \in D$, or $\alpha + \beta' \in D$, or $\alpha' + \beta \in D$.

Many choices for D will satisfy the given properties, such as:

$$D = A \oplus B \tag{14}$$
$$D = (\{\alpha\} \oplus B) \cup (\{\beta\} \oplus A) \tag{15}$$
$$D = W \tag{16}$$
$$D = (\{\alpha\} \oplus B) \cup \{\alpha' + \beta | (\alpha' + \beta) \in W\} \tag{17}$$
$$D = (\{\beta\} \oplus A) \cup \{\beta' + \alpha | (\beta' + \alpha) \in W\} \tag{18}$$

Any such choice of D allows us to use the domination check of Equation 13 to either remove ϕ from consideration or to find a vector that has not yet been added to W.

Incremental pruning is actually a *family* of algorithms. The particular choice for D determines how efficient the algorithm is. In general, small sets make a better choice, since this reduces the number of constraints in the linear program created by the `dominate` function. Incremental Pruning using a combination of Equations 17 and 18 is referred to as *restricted region* (RR) and is currently deemed to be the best POMDP solution algorithm [1].

3 Parallel Restricted Region

Incremental Pruning decomposes the problem into independent parts. The easiest way to parallelize RR would be to perform the cross sum and filter operations on the S_z^a sets in parallel. The number of threads using this approach would be limited to $|\mathcal{A}|$.

Our approach is to perform the individual domination tests in parallel. Each domination test is performed by a domination server which may reside on another computer. The sequential algorithm shown in Figure 1 assumes that W and F will be updated at the end of each domination check and that W and F do not change during the domination check. However, this cannot be guaranteed when several domination tests are occurring simultaneously, and synchronizing the data between servers would be a significant problem. We change the algorithm slightly, so that the vectors are not moved from F to W, but are simply marked as dominant. After all of the vectors in F have been checked, the ones marked as dominant are moved to W. This approach allows the initial sets to be sent to all of the servers

```
purge (F,S)
    W ← ∅
    – Select the obvious winners at each edge of the belief space and put them in W
    for s ∈ S
        max ← 0
        for φ ∈ F
            if ((e_s · φ > max) or ((e_s · φ = max) and (ω_s < φ))) then
                ω_s ← φ
                max ← e_s · φ
    for s ∈ S
        if (ω_s ∈ F) then
            W ← W ∪ {ω_s}
            F ← F\{ω_s}
    – Test the remaining vectors for domination
    current ← 0
    for i ∈ NUMSERVERS
        child_i ← CreateThread(ThreadProc, F, W, current, i)
    for i ∈ NUMSERVERS
        join(child_i)
    for (ω ∈ F)
        if (marked(ω)) then
            W ← W ∪ ω
    return W
```

Fig. 2. Parallel algorithm for purging a set of vectors. The algorithm splits into `NUMSERVERS` threads, which mark the vectors in F that belong in W.

at the beginning of each parallel merge which minimizes subsequent communication.

Our goal is to distribute the linear programs over as many processors as possible and do as many in parallel as we can. In order to do so, we must modify the purge function. We modified the cross sum operator $\alpha \oplus \beta$ so that the resulting vector is "tagged" to indicate α and β. We then added an operation `tagged`$\alpha(\alpha, X)$ to return the set of vectors in X that are tagged with the given α and an operation `tagged`$\beta(\beta, X)$ to return the set of vectors in X that are tagged with the given β. The complete F and W sets are sent to the servers, which use these operations to extract the subsets required for each application of Equations 17 and 18. We then modified the purge algorithm to perform domination checks in parallel.

The parallel purge algorithm is shown in Figure 2. As with RR, the algorithm first finds the dominant vectors at each corner of the

```
ThreadProc(F, W, current, i)
    SendToServer(i, "NewProblem")
    SendToServer(i, F, W)
    while current < |F|
        lock current
        φ ← F_current
        current ← current + 1
        unlock current
        SendToServer(i, "TestDomination")
        SendToServer(i, φ)
        x ← RecieveFromServer(i)
        if (x ≠ ⊥) then
            max ← 0
            for (φ ∈ F)
                if ((x · φ > max) or ((x · φ = max) and (ω < φ))) then
                    ω ← φ
                    max ← x · φ
            lock F
            F_ω ← mark(F_ω)
            unlock F
```

Fig. 3. Each thread communicates with one server. The thread sends F and W to the server, then begins processing vectors in F.

belief space and moves those vectors into W. Then it starts a number of threads to perform the domination checks on the remaining vectors in F. The threads all execute the same `ThreadProc` code (shown in Figure 3) locally and each thread is responsible for communicating with one server. Each thread sends F to the remote server and then enters a loop. On each iteration, the thread takes the next available vector in F and instructs the remote server to test that vector for domination. If the remote server returns a witness point, then the thread finds the largest vector in F at that point and marks it for addition to W. When all of the vectors in F have been tested, the threads terminate, and the main program continues.

Figure 4 shows the server algorithm for the parallel purge algorithm. The server understands two commands. The `NewProblem` command instructs it to receive the following F and W sets. The `TestDomination` command instructs the server to test a given vector in F for domination. When a `TestDomination` command arrives, it extracts the α and β tags from the vector and extracts the sets

```
Server()
    loop
        ReceiveFromThread(command)
        if (command = "NewProblem") then
            ReceiveFromThread(F, W)
        elseif (command = "TestDomination") then
            ReceiveFromThread(φ)
            α = taga(φ)
            A = taggedα(α, F)
            A^W = taggedα(α, W)
            β = tagb(φ)
            B = taggedβ(β, F)
            B^W = taggedβ(β, W)
            if (|A| + |B^W| < |B| + |A^W|) then
                D = A ∪ B^W
            else
                D = B ∪ A^W
            x = dominate(φ, D\{φ})
            SendToThread(x)
```

Fig. 4. Server algorithm for parallel purge.

$A = \{\alpha\} \oplus B$, $B = \{\beta\} \oplus A$, $B^W = \{\alpha' + \beta | (\alpha' + \beta) \in W\}$, and $A^W = \{\beta' + \alpha | (\beta' + \alpha) \in W\}$. D is calculated as $A \cup B^W$ or $B \cup A^W$, whichever is smaller. Finally, ϕ is tested for domination over D, and the result is returned.

4 Experiments and Results

We implemented the parallel RR algorithm in C++ and ran it on a network of Sun Ultra I workstations. For comparison, we also implemented a serial version of the RR algorithm derived from the same code base. Thus, the only difference in the two versions is in the incprune algorithm as outlined in this paper.

We selected four problems from the literature and one of our own. The problems from the literature are representative of the problems that can be solved with current POMDP solution methods. Our Robot problem is significantly more difficult, and the advantage of our parallel algorithm is more pronounced on this problem. The problems are listed in Table 1. Descriptions of the Tiger, Part Painting, and 4×3 problems are available from the references. Robot

Table 1. Test problem parameters.

Problem	$\|\mathcal{S}\|$	$\|\mathcal{A}\|$	$\|\mathcal{Z}\|$	Iterations	Reference
Tiger	2	3	2	273	Cassandra, Kaelbling and Littman [3]
Shuttle	8	3	5	283	Chrisman [5]
Part Painting	4	4	2	100	Kushmerick, Hanks, and Weld [6]
4×3	11	4	6	100	Parr and Russell [9]
Robot	64	3	16	25	Pyeatt and Howe [10]

Table 2. Results of experiments with run times given in seconds. The column headings for the Parallel RR algorithm indicate the number of servers.

Problem	RR	Parallel RR				
		1	2	3	4	5
Tiger	209	220	177	139	103	70
Shuttle	415	566	473	453	448	444
Part Painting	137	156	103	89	81	76
4×3	1359	1474	1156	1100	1079	1083
Robot	>86400	>86400	>86400	83465	62034	41648

is a small robot navigation problem where the goal is to reach a specified location.

Table 2 shows the performance of the two algorithms on the selected problems. The parallel IP algorithm was run with a varying number of servers; the algorithm was terminated when run time exceeded 24 hours. This occurred in three cases on the Robot problem. On the Tiger and Part Painting problems, each additional processor results in successively smaller performance increase. On the 4×3 problem, the performance actually degraded when a fifth processor was added. This diminishing return results from the communications overhead increasing in relationship to the amount of computation that is performed. On the Shuttle problem, the serial algorithm outperformed the parallel algorithm due to the time required to send the S_z^a sets over the network. On the Robot problem, which requires considerably more computation, the relationship of run-time to number of processors is more linear. Note that this relationship does not appear to correlate with the number of states, actions, or observations in the POMDP problem, but appears to be related to the size of the S_z^a sets that are created. Thus, $|\mathcal{S}|$, $|\mathcal{A}|$, and $|\mathcal{Z}|$ are not good indicators of the difficulty of POMDP problems.

5 Future Work

Our current implementation is really a distributed algorithm that spreads the work across several computers connected with 10Mbit/s Ethernet. The major drawback to the current implementation is the network bottleneck created when all of the threads send the F set to the servers at once. This bottleneck could be alleviated in several ways. The main program could broadcast the F set to all servers with a single transmission. This approach was not taken because TCP streams do not support broadcast. UDP supports broadcast, but implements unreliable communication using datagrams instead of streams. Broadcasting large vector sets using UDP would require breaking the data into packets and providing some mechanism to ensure reliability. This approach, though feasible, would require an added layer of complexity. Another way to reduce the bottleneck is to use multiple high speed network interfaces instead of a single Ethernet. Linux Beowulf and IBM SP systems both provide multiple high speed networks and would be ideal for this application. Third, and most obviously, the parallel IP algorithm could be implemented on a shared memory multiprocessor system.

This algorithm only parallelizes Equation 4, but could be extended to perform the incprune in Equation 3 and the cross sum operation in Equation 5 in parallel as well. Currently, these operations account for the majority of the time used by the parallel algorithm on the Tiger and Robot problems.

The parallel algorithm is most useful on problems with large S_z^a sets, where the overhead involved in sending the data to the servers is much less than the time required for computation in the incprune operation. Thus, for small problems, parallel solution does not present an advantage, while the most difficult POMDP problems can benefit greatly from parallelization.

6 Acknowledgments

This research was supported in part by National Science Foundation Career Award IRI-9624058. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation herein. We thank the

reviewers for catching some minor errors in the notation and description.

References

1. Anthony Cassandra, Michael L. Littman, and Nevin L. Zhang. Incremental pruning: A simple, fast, exact algorithm for partially observable markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence*, 1997.
2. Anthony R. Cassandra. A survey of POMDP applications. In Michael Littmann, editor, *Working Notes: AAAI Fall Symposium on Planning with Partially Observable Markov Decision Processes*, pages 17–24. AAAI, October 1998.
3. Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, 1994.
4. Anthony R. Cassandra, Michael L. Littman, and Leslie Pack Kaelbling. Efficient dynamic-programming updates in partially observable Markov decision processes. Technical Report CS-95-19, Brown University, Providence, RI, 1996.
5. Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 183–188. AAAI, AAAI Press/MIT Press, July 1991.
6. N. Kushmerick, S. Hanks, and D. S. Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1–2):239–286, 1995.
7. G. E. Monahan. A survey of partially observable Markov decision processes. *Management Science*, 28(1):1–16, 1982.
8. Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987.
9. R. Parr and Stuart Russell. Approximating optimal palicies for partially observable stochastic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.
10. Larry D. Pyeatt and Adele E. Howe. Integrating POMDP and reinforcement learning for a two layer simulated robot architecture. In *Third International Conference on Autonomous Agents*, Seattle, Washington, May 1999.
11. Richard D. Smallwood and Edward J. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, 21:1071–1088, 1973.
12. C. C. White III. Partially observed Markov decision processes: A survey. *Annals of Operations Research*, 32, 1991.
13. Nevin L. Zhang and Wenju Liu. Planning in stachastic damains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1996.