Spring 2011 CS264: Parallel POMDP

Mark Woodward

May 8, 2011

1 Introduction

My research is on robots interactively learning from humans. During the interaction there are questions that the robot would like to know but can't directly measure. Some examples of questions are: "Where do the dishes go?", "Which object is the human pointing at?", and "Is the human finished with the teaching session?". Through its actions, the robot can elicit measurements that may provide answers to its questions. The issue is which action to take on the next time step that, in expectation, will allow the robot to learn as much as possible, as quickly as possible; formalized in the next section.

One model for action selection under uncertainty is a partially observable Markov decision process (POMDP). Solving a POMDP requries a lot of processing power. This project describes a distributed implementation of a POMDP solver.

I will describe my manipulations of the POMDP mathematics that allowed for parallelization, and I will present the results of running the Parallel-POMDP solver on my current robot models (teaser: 2.9x speedup on a 12 MPI processors).

2 Partially Observable Markov Decision Process (POMDP)

Figure 1 depicts a POMDP tree expanded one time steps into the future. Associated with each node is a probability distribution, shown as a univariate guassian but in practice the distribution is multi-modal and consists of hundreds of random variables. As each action a is taken and each measurement z is received the distribution changes.

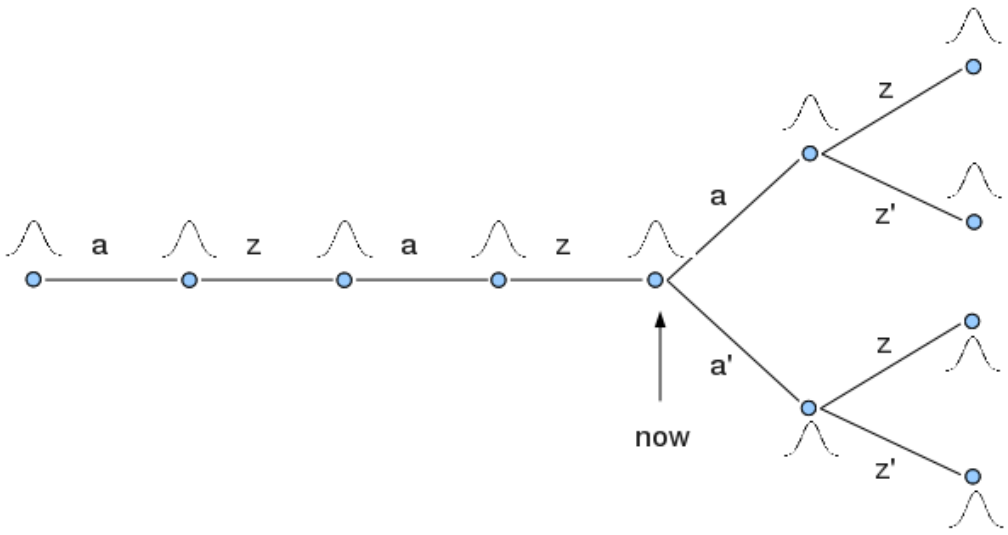

Figure 1: A POMDP tree run out 1 time step over the next action and potential next measurements

The "now" node represents the current probability distribution at the current time. Looking backward in time, the robot knows all actions that were taken and all measurements that were received. Looking forward in time from "now", the robot will take an action and receive a measurement at each time step, only one timestep is shown in the figure. The goal is to expand the tree as far as processing will allow, apply a cost function to the probability distribution at the leaf nodes, and work the cost back down the tree, taking expectations as measurement nodes and minimizations at action nodes. To formalize this, let a^t be the action taken at time step t and $a^{t:1}$ be all actions taken from timestep 1 to time step t, and use the same notation for measurements z^t and $z^{t:1}$. Then, the optimal action "now", the action that minimizes the expected cost for a time horizion H, is:

$$\pi(b_0) = argmin_{a^1} ExpectedCost(a^1)$$

with

$$ExpectedCost(a^1) = E_{z^1} \min_{a^2} E_{z^2} \min_{a^3} E_{z^3} \cdots \min_{a^H} E_{z^H} Cost(b_H | z^{H:1}, a^{H:1})$$

In my application, the cost function is entropy; meaning information entropy, not thermodynamic entropy :). Entropy is a measure of uncertainty, so choosing actions to minimize entropy will choose actions to minimize uncertainty, which is a reasonable objective for a learning robot.

3 Idea

One approach to parallelizing a POMDP solver would be to expand the tree on the master to a depth where the number of leaves equals the number of servers and then give each server a leaf node to expand from there. There are a couple problems with this approach. First the slave servers sit idle while the master is performing the initial tree expansion, which is wasteful. Second, a full probability distribution is associated with each node, and each distribution requires hundreds of megabytes to represent it, so moving the nodes to the slave servers would be slow.

My idea is to slice up the "now" distribution, hand each server a partial distribution, and allow each server to expand it's full tree. This allows all servers to start processing immediately and vastly reduces the network traffic. Below I describe the mathematical tricks that allowed for this implementation. In the end, each server accumulates partial entropies for leaf nodes and statistics for normalization coefficients and returns these to the master, who can quickly merge them to compute the expected entropy for each of its next actions.

4 Reworking the POMDP for parallelization

4.1 Expected Entropy

In order to compute the expected entropy of a next action, I first need to calculate the entropy of the probability distribution at each leaf node of the POMDP tree. Unfortunately the standard entropy formula requires a normalized probability distribution, and in my formulation, the probability distribution is spread out across servers.

By reworking the Entropy formula, I am able to use the unnormalized probability distribution to compute a partial entropy that can be summed across nodes and merged when the normalization constant is available.

Let x^t be a potential assignment to all questions that the robot has at time step t, these would be called latent random variables in the literature. I will sometimes refer to x^t as the "state" at time t, but just think of it as assignments to questions. Let $p(x^t | z^{t:1}, a^{t:1})$ be the probability of the x^t assignment at time t, given the history of measurements $z^{t:1}$ and the history of actions $a^{t:1}$. Let $\underline{p}(x^t | z^{t:1}, a^{t:1})$ be the same probability, but unnormalized. The normalization constant that we can only know globally, but can't know on each server, is $p(z^t | z^{t-1:1}, a^{t:1})$, see the "Measurement Likelihood" section. The relationship between these probabilities is:

$$p(x^t | z^{t:1}, a^{t:1}) = \frac{\underline{p}(x^t | z^{t:1}, a^{t:1})}{p(z^t | z^{t-1:1}, a^{t:1})}$$

The classical entropy equation requires normalized probabilities. Here is the derivation of a form that is amenable to distributed computation, making use of the unnormalized probabilities:

$$
\begin{aligned}
Entropy &= -\sum_{x^t} p(x^t|z^{t:1}, a^{t:1}) log(p(x^t|z^{t:1}, a^{t:1})) \\
&= -\sum_{x_t} \frac{\underline{p}(x^t|z^{t:1}, a^{t:1})}{p(z^t|z^{t-1:1}, a^{t:1})} log\left(\frac{\underline{p}(x^t|z^{t:1}, a^{t:1})}{p(z^t|z^{t-1:1}, a^{t:1})}\right) \\
&= -\sum_{x_t} \frac{\underline{p}(x^t|z^{t:1}, a^{t:1})}{p(z^t|z^{t-1:1}, a^{t:1})} log(\underline{p}(x^t|z^{t:1}, a^{t:1})) + \sum_{x_t} \frac{\underline{p}(x^t|z^{t:1}, a^{t:1})}{p(z^t|z^{t-1:1}, a^{t:1})} log(p(z^t|z^{t-1:1}, a^{t:1})) \\
&= -\sum_{x_t} \frac{\underline{p}(x^t|z^{t:1}, a^{t:1})}{p(z^t|z^{t-1:1}, a^{t:1})} log(\underline{p}(x^t|z^{t:1}, a^{t:1})) + log(p(z^t|z^{t-1:1}, a^{t:1})) \sum_{x_t} p(x^t|z^{t:1}, a^{t:1}) \\
&= \frac{1}{p(z^t|z^{t-1:1}, a^{t:1})} \left(-\sum_{x_t} \underline{p}(x^t|z^{t:1}, a^{t:1}) log(\underline{p}(x^t|z^{t:1}, a^{t:1}))\right) + log(p(z^t|z^{t-1:1}, a^{t:1}))
\end{aligned}
$$

The quantity in parentheses can be computed independently on each server using the server's own unnormalized probabilities. Each server can pass back their partial "unnormalized entropy" as a single real number. During the merge phase the partial unnormalized entropies can be summed up, and as long as we have the normalization constant, $p(z^t|z^{t-1:1}, a^{t:1})$, we can compute the true entropy at that leaf using the equation above.

4.2 Detergent State Bayes Filter

Bayes filtering is an important component of solving POMDPs, it updates the probability distributions along the path to a leaf node and it is the step that dominates the processing. As shown here, in the special case of divergent states, the probability of a state at the horizon depends only on the probability of a single state at time 0, and an additive normalization constant. This means that we can safely split the probability distribution at the "now" node and distribute the partial distributions across the servers.

The bayes filter update for the probability of a state x^t given an action and measurement history, $a^{t:1}$ and $z^{t:1}$, with divergent states, is as follows:

$$
\begin{aligned}
p(x^t|z^{t:1}, a^{t:1}) &= \frac{p(z^t|x^t) \sum_{x^{t-1}} p(x^t|a^t, x^{t-1}) p(x^{t-1}|z^{t-1:1}, a^{t-1:1})}{p(z^t|z^{t-1:1}, a^{t:1})} \\
&= \frac{p(z^t|x^t) p(x^t|a^t, x^{t-1}) p(x^{t-1}|z^{t-1:1}, a^{t-1:1})}{p(z^t|z^{t-1:1}, a^{t:1})}
\end{aligned}
$$

The second step is due to the fact that with divergent states the previous state x^{t-1} is always known. i.e. with divergent states, a state at time $t-1$ leads probabilistically to multiple states at time t, but those states can only be reached through state x^{t-1}, not through another state at time $t-1$. Again, this equation depends on a single probability in the "now" node's distribution, which is a requirement for my proposed parallelization.

An interesting feature of the bayes filter update equation in general is that the denominator does not depend on x^t, so it is a constant for $p(x^t|z^{t:1}, a^{t:1})$, since $p(x^t|z^{t:1}, a^{t:1})$ only varies with x^t. In the non-parallel case we can safely ignore this constant, since, if we want to later normalize the distribution, we can always do so by dividing the probability by the sum of all unnormalized $p(x_i^t|z^{t:1}, a^{t:1})$. But, in the parallel case, the unnormalized $p(x_i^t|z^{t:1}, a^{t:1})$ are spread across servers, so we can't normalize when we need it. Thus we need to efficiently keep around statistics that allow us to reconstruct the normalizing constant $p(z^t|z^{t-1:1}, a^{t:1})$ during the merging step. This is the focus of the next section.

4.3 Measurement Likelihood

We need the measurement likelihoods, $p(z^t|z^{t-1:1}, a^{t:1})$, for two reasons: 1) as mentioned in the POMDP section, expected cost is back propagated through measurement nodes by taking expected values, and the $p(z^t|z^{t-1:1}, a^{t:1})$ are the probabilities needed for that expectation, and 2) we need the measurement likelihoods

as the normalization constants in the modified Entropy calculation above. To find a solution we expand the measurement likelihood equation:

$$\begin{aligned} p(z^t|z^{t-1:1}, a^{t:1}) &= \sum_{x^t} p(z^t|x^t)p(x^t|z^{t-1:1}, a^{t:1}) \\ &= \sum_{x^t} p(z^t|x^t) \sum_{x^{t-1}} p(x^t|a^t, x^{t-1})p(x^{t-1}|z^{t-1:1}, a^{t-1:1}) \\ &= \sum_{x^t} p(z^t|x^t)p(x^t|a^t, x^{t-1})p(x^{t-1}|z^{t-1:1}, a^{t-1:1}) \end{aligned}$$

The third line follows from the second line and the divergent states property. Also, from the "Divergent State Bayes Filter" section we know that $p(x^{t-1}|z^{t-1:1}, a^{t-1:1})$ does not consist of any summations. This means that $p(z^t|z^{t-1:1}, a^{t:1})$ is a summation of terms, where each term depends only on one x^t. Thus, we can compute partial sums for $p(z^t|a^t, z^{t-1:1}, a^{t-1:1})$ on each server, based on the x^t's that that server is computing, then transfer one real number per branch per server back to the master, and sum them up during the merge step to get the full normalization coefficients $p(z^t|a^t, z^{t-1:1}, a^{t-1:1})$.

5 Results

Figure 2 shows the speedup over the original serial code for one through 12 processes on the resonance cluster. The speedup on 12 processes is 2.9x. The mean processing time of the original serial code is 4.64 seconds. The mean processing time of the parallel code on 12 processes is 1.54 seconds.

One of the reasons that the speedup is not linear is that the amount of network traffic increases with each new process; each process returns the same number of normalization constant statistics, roughly 250,000, so more processes means more network traffic (note that this is still substantially less traffic than passing around full distributions).

I was very happy with this speedup, as my goal was a speedup of 2x, and it is likely that more processes would show more speedup.

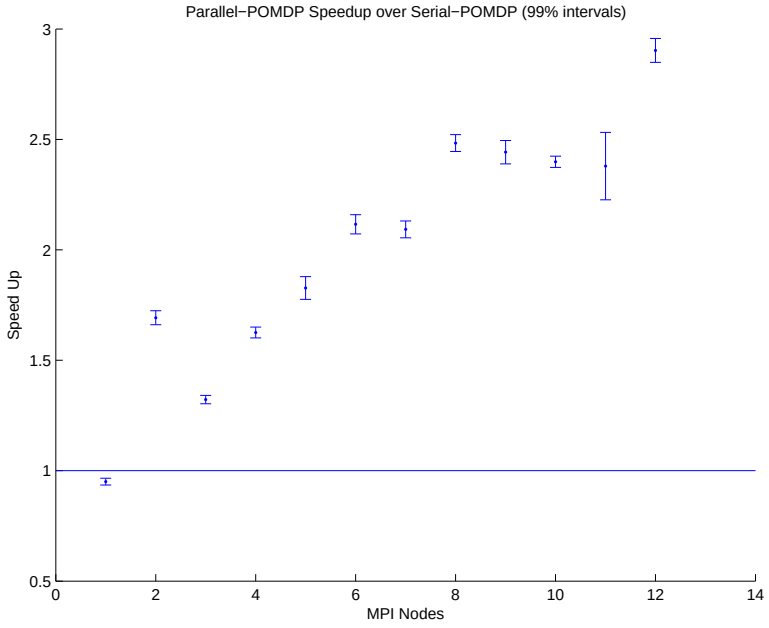

Figure 2: Speed Up

6 Conclusions

I am happy with the results and excited about solving POMDPs by splitting the distribution, as apposed to distributing branches of the tree. In the limit, this might be a good fit for Map-Reduce, where the map would send out each element of the "now" distribution. But that's for future work!