Distributed Parallel Particle Filter in Julia

Final Project Report for 18.337J/6.338J Parallel Computing

Gustavo Goretkin

1 Introduction

The goal of this final project is to use the Julia REF language to implement a particle filter that can take advantage of distributed-memory parallelism. The implementation is not yet complete, but this report explains the intended approach and progress.

The Julia language is a technical computing language that is currently in development. It has resources for different paradigms such as vectorized/matrix programming (like MATLAB) and systems-level programming (like C) and has a rich type system.

A particle filter is an implementation of the general recursive state estimation Bayes filter. The essence of recursive state estimation is that there is some dynamical system whose state you want to estimate. The filter has some model of the dynamical system. For clarity, take the dynamical system to be discrete time and in continuous space:

$$\begin{aligned} x_{t+1} &= f(x_t, u_t, p_t) \\ z_t &= g(x_t, u_t, m_t) \end{aligned}$$

x_t, u_t, and z_t are the state, control input, and observation at time t. Furthermore, p_t and m_t are random variables, corresponding to the process noise and measurement noise.

The recursive state estimator keeps a representation of the hypothesis – which is a probability distribution over states – at time step x_{t-1}. It updates that estimate by predicting where the system will be, given the control input and previous state (using the system function f) and corrects that prediction using Bayes' rule applied to the observation z. The corrected prediction is the hypothesis at time step x_t, and is used as the prior distribution at the next time step, which is why it is a *recursive* state estimator. A clear exposition of the Bayesian filter framework is provided in Probabilistic Robotics REF.

The general Bayesian filter can be realized in many ways, depending on the chosen representation of the hypothesis. A Kalman filter is also a Bayesian filter where the hypothesis is a multivariate Gaussian distribution. The Kalman filter, in its original formulation, also makes the assumption that f and g are multi-linear functions and that the process noise and measurement noise are normal. The particle filter is another way to realize a Bayesian filter. It does not make these assumptions and allows a richer class of state hypotheses, not just Gaussian hypotheses.

The way I described the Bayesian filter suggests that the filter is only useful for providing real-time best estimates at any point in time. In fact, the filter can be used in a *non-causal* way by using observations at later time steps to improve the state hypothesis at the current time step. Essentially the filter can be run forward, backward, or both.

2 Particle Filter

The particle filter maintains a hypothesis on the state as a fixed set of particles in the state space. Each particle has a weight, which is a scalar value. Initially all the weights are uniform. The particles approximate a continuous probability distribution over the state space. If there is a region of the state space with a high concentration of particles, then the true state has a high probability of being in that region.

To predict the hypothesis at the next time step, each particle is individually propagated according to f. Then the likelihood of each particle – the probability of the observed o given the state of the system is that particle – is used to update each particles weight multiplicatively. Therefore, particles that are more likely are given higher weights.

It is important to ensure that there are more particles in the likely regions of the state space to provide finer representation. Otherwise, computation effort is spent on unlikely believes. To redistribute the particles, they are resampled according to their weights. The new set of particles is then given uniform weights.

There are many different resampling methods – with differing statistical properties and computational complexities.

3 Parallel Implementation in Julia

The approach is to distribute particles across different machines. The re-weighting step and the particle propagation step are embarrassingly parallel and can be done without any communication between the nodes.

The resampling step is not so simple. Depending on the resampling method, there are different approaches. For sampling with replacement, ideally the particles distributed across the nodes should be treated as belong all to one pool from which the sampling is done.

One attempt to decouple the problem is to sample with replacement on each node separately. The number of particles that a node should resample is

$$\frac{\text{sum of weights of particles on that node}}{\text{sum of weights of all particles}} \times \text{number of particles}$$

That value might not be an integer, so rounding is necessary. This requires a global sum over all the weights, which can be done in parallel with little communication (each node broadcasts one scalar corresponding to the sum of its local particles).

It is ideal to keep the number of particles per node balanced. This will keep the embarrassingly parallel parts of the computation balanced. After a re-weighting step, there is no guarantee that the particles will be balanced between the nodes. Rebalancing the particles is an expensive communication step.

I intend to use the Single Instruction Multiple Data paradigm. Each node knows how many particles every (other) node will end up with after the parallel resampling step and each node runs the same rebalancing algorithm and therefore each node knows the other nodes it should accept particles from or the other nodes it should deliver particles to.

There are many ways to solve the problem of rebalancing. For example a complete rebalancing optimal with respect to communication can be found by solving a minimum-cost graph flow problem REF. Or perhaps all that is necessary is to exchange particles between the node with the largest surplus and the node with the largest deficit. Once the code is in place, it will be important to compare these methods.

4 Concerns and Future Work

One concern is that the particle filter begins having gains from distributed-memory parallelization only for very large number of particles. Even if that is the case, the code should serve as a research platform to compare different techniques for parallelizing the parallel particle filter.

5 Contributions to Julia

The hidden agenda I had for this project was to expose myself to the Julia environment. I have filed some issues and will hopefully provide an important use case of the distributed array. I have also written a few simple functions to incorporate into the Julia standard library, like cumsum on distributed arrays and argmin/argmax and binary search and resampling functions.

6 Acknowledgement

Thank you to Professor Alan Edelman and T.A. Jeff Bezanson for their help throughout the course. Jeff was especially helpful in fleshing out Julia code and helping debug during his long office hours and through instant messaging. Many of the Julia issues that I filed, Jeff resolved within a couple of minutes.