Passage: A Parallel Sampler Generator for Hierarchical Bayesian Modeling

Chad Scherrer
Independent Consultant
Yakima, WA
chad.scherrer@gmail.com

Iavor Diatchki
Galois, Inc.
Portland, OR
diatchki@galois.com

Levent Erkök
Intel
Portland, OR
erkokl@gmail.com

Matthew Sottile
Galois, Inc.
Portland, OR
mjsottile@gmail.com

Abstract

We introduce Passage, an EDSL (Embedded Domain Specific Language) for hierarchical Bayesian modeling. Passage is hosted by the functional programming language Haskell, and inherits Haskell's infrastructure for rapid code development and higher-order functions. We expect Passage to be interesting to both casual users interested in Bayesian modeling, and also to researchers in the field who want to explore computational aspects of statistical modeling. In particular, our framework allows easy manipulation of the model and the generated code to explore alternative execution strategies, including targets that can make use of the heavy parallelism as afforded by many modern machines. Passage is open-source software, available for free download.

1 Introduction

Passage is a PArallel SAmpler GEnerator; it builds parallel C/OpenMP code from a high-level model specification and observed data. Passage is implemented as a EDSL (*Embedded Domain Specific Language*) hosted by the Haskell language. Passage programs are therefore simply Haskell programs that call some Passage-specific language constructs.

In particular, Passage makes it easy to describe hierarchical Bayesian models in terms of higher-order abstractions. Given a model, Passage performs symbolic calculation to find a form of the log-density suitable for sampling. It then calculates a coloring of the dependency graph, and uses this to construct parallel sampling code using C/OpenMP.

Due to space limitations, we will assume a working understanding of Bayesian analysis and the Haskell programming language.

2 A simple example

```
1 simpleModel n = do
    mu  ← improperUniform
3   tau ← improperScale
    x   ← iid n (normal mu tau)
5   return (mu, tau, x)
```

Figure 1: Passage code for a simple example.

Suppose we'd like to model a set of n observations as iid normal. Figure 1 gives a function `simpleModel` that takes a single parameter `n` corresponding to the number of observations. Lines 2 and 3 specify improper prior distributions for both the mean and the precision [We use the term "improper scale" to indicate that $\log \tau$ is given an improper uniform prior], and Line 4 gives the distribution for each component of `x`. Finally, Line 5 returns the triple `(mu,tau,x)`, allowing any of these variables to be `observed` (bound to a known constant value) or `monitored` (output during sampling).

With the model specified, we can build the simulation driver, which indicates which variables we would like to monitor, as well as those for which we have observed values; see Figure 2. Given a list of doubles x0, evaluation of sim x0 will now result in creation of a simpleExample directory containing C/OpenMP sampler code. This sampler performs parallel Gibbs sampling (here using 2 threads) from the posterior distribution.

```
sim x0 = genSimulator "simpleExample" $ do
  setThreadNum 2
  let n = length x0
  (mu, tau, x) ← runModel (simpleModel n)
  zipWithM_ observe x x0
  monitor "mean" mu
  monitor "precision" tau
```

Figure 2: The simulation driver for our simple example.

3 Programming abstractions

Haskell is well-suited to expressing higher-order abstractions; Passage takes advantage of this to allow higher-order specification of Bayesian models. For example, a Markov chain can be specified by a length n, an initial distribution d0, and a function next that takes a value from the jth point in the chain and returns a distribution at the $(j+1)$st. This is easily expressed in Passage, as shown in Figure 3.

```
markovChain n d0 next = mc n d0
  where
  mc 0   _     = return []
  mc n distr = do
    x  ← distr
    xs ← mc (n-1) (next x)
    return (x:xs)
```

Figure 3: A higher-order Markov chain function.

Now given the markovChain function, we can use it to define other models that are given in these terms. For example, consider a simple random walk consisting of a discrete stochastic process (x_k), where

$$x_0 \sim \mathcal{N}(0,1)$$
$$x_k \sim \mathcal{N}(x_{k-1},\tau) \ .$$

```
randomWalk n = do
  tau ← improperScale
  x   ← markovChain n (normal 0 1)
    (λxk → normal xk tau)
  return (tau, x)
```

Figure 4: Using the markovChain function to define a random walk.

The Passage implementation is shown in Figure 4. Note that there need not be any distinction whether the markovChain function referred to is built-in or user-defined.

4 Building distributions

```
logisticBernoulli r = makeDist (discrete 1) f
  where
  f x = r * x - r - log (1 + exp (-r))
```

Figure 5: A reparameterized Bernoulli distribution.

Passage comes with a wide variety of predefined distributions, both discrete and continuous. In addition, this can be easily extended using the makeDist function, which requires specification of a support (discrete with some max, nat, real, or posReal) and a log-density function. For example, Figure 5 gives a simple example of a Bernoulli distribution, parametrized by the logit of the probability of success.

Alternatively, Passage allows distributions to be defined in terms of auxiliary variables. Figure 6 shows how this approach can be used to easily define Student's t distribution as a mixture of normals, and a symmetric Dirichlet distribution as a normalized vector of gammas.

```
studentT df = do
  v ← chiSquare df
  normal 0 v

symDirichlet n alpha = do
  gs ← iid n (gamma alpha 1)
  return [g / sum gs | g ← gs]
```

Figure 6: Student's t, and a symmetric Dirichlet.

5 Symbolic manipulations

Introduction to the model of the relation $x \sim \mathcal{N}(\mu,\tau)$ corresponds to an increase in the joint log-density in the amount of

$$\ell = \tfrac{1}{2}\log\tau - \tfrac{1}{2}\tau x^2 + \tau\mu x - \tfrac{1}{2}\tau\mu^2 \ .$$

Internally, Passage transforms this into a set of functions, each of which is a sum of products of pairs:

$$\ell_x = \left(x^2\right)\left(\tfrac{1}{2}\tau\right) + (x)(\tau\mu)$$
$$\ell_\mu = (\mu)(\tau x) + \left(\mu^2\right)\left(-\tfrac{1}{2}\tau\right)$$
$$\ell_\tau = (\log\tau)\left(\tfrac{1}{2}\right) + (\tau)\left(-\tfrac{1}{2}x^2 + \mu x - \tfrac{1}{2}\mu^2\right)$$

Note that in each pair, the first factor is a function of the particular variable of interest, and the second factor is constant relative to this variable. Successive variable specifications lead to aggregation of similar terms in all such functions. Thus similar terms are collected easily and effectively, yielding an efficient functional representation fit for a variety of sampling approaches.

Finally, note that values that are constant relative to the variable of interest are discarded; we work with relative values only and do not calculate normalization factors for any distribution.

6 Parallel thread scheduling

In addition to yielding a convenient functional form, the above analysis of the joint log-density yields an easy way to arrive at the dependency graph mapping each node to its Markov blanket. Sampling a given variable requires reading values at the corresponding node's neighbors, and writing over the value at node for the given variable. Thus in a parallel context, adjacent nodes cannot be sampled simultaneously, and the dependency graph corresponds to an *interference graph* in the usual compiler sense. Graph coloring provides a guide we can use to sample nodes in parallel when possible, while avoiding sampling dependent nodes simultaneously.

Figure 7 shows a simple example. First, we color the nodes of the dependency graph. Next, we construct the desired number of OpenMP threads, with a number of *barriers* one greater than the number of colors (to force synchronization before each *write*). We then populate this structure with the sampler code from the various nodes.

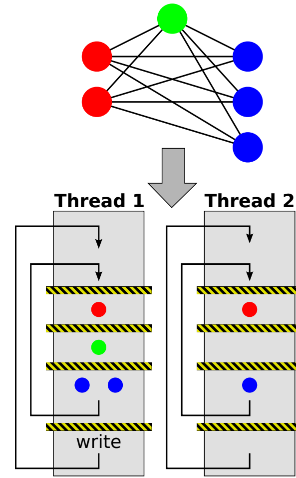

Figure 7: Thread scheduling using graph coloring.

7 Conclusion and Future Work

We see great promise in Passage for its approach to abstractions within Haskell, symbolic processing of the joint log-density, and automatic parallel code generation. But Passage is still very young, and there is clearly lots of work still ahead. A number of available optimizations have not yet been implemented, including marginalization and conjugacy. Also, to this point Passage is inherently univariate, with multivariate distributions expressed simply as a joint distribuition, but without any fundamental distinction.

We hope to continue to improve Passage in these ways and others, to provide a laboratory for exploring new ideas in modeling and parallel sampling.

Passage is available for free download from `http://hackage.haskell.org/package/passage`.

Acknowledgments

The authors are grateful to DOE's ASCR Applied Mathematics Program for funding, to Andrew Gelman for arranging subcontract funding for Galois, and to Pacific Northwest National Laboratory for releasing Passage into the public domain.

Special thanks also go to Andy Adams-Moran, Bob Carpenter, Aleks Jakulin, Don Stewart, and Rob Zinkov for helpful discussions and planning.

References

[1] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries*. Cambridge Press, 2003.

[2] David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS: A bayesian modelling framework: Concepts, structure, and extensibility. *Statistics and Computing*, 10:325–337, October 2000.

[3] Martyn Plummer. JAGS: A program for analysis of bayesian graphical models using Gibbs sampling, 2003.

[4] Hal Daume. HBC: Hierarchical Bayes Compiler. `http://www.umiacs.umd.edu/~hal/HBC/`, 2008.

[5] George Marsaglia. `http://programmingpraxis.com/2010/10/05/george-marsaglias-random-number-generators/`, 2010.

[6] Radford Neal. Slice sampling. *Annals of Statistics*, 31:705–767, 2000.

[7] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[8] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.