Parallel resampling in the particle filter (appendices)

Lawrence M. Murray[*], Anthony Lee[†] and Pierre E. Jacob[‡]

June 9, 2015

A Pseudocode conventions

The algorithms presented in this work are described using pseudocode with a number of conventions. We distinguish between the **for each** and **for** constructs. The former is used where the body of the loop is to be executed for each element of a set, with the order unimportant. The latter is used where the body of the loop is to be executed for each element of a sequence, where the order must be preserved. The intended implication is that **for each** loops may be parallelised, while **for** loops cannot be. The **atomic** keyword is used to indicate that a line must be executed as if it constitutes one instruction (i.e. an atomic operation) in order to avoid read and write conflicts between concurrently running threads.

A number of primitive operations such as searches, transformations, reductions, sorts and prefix sums are used throughout pseudocode. These are specified in Code 1. Such operations will be familiar to users of, for example, the C++ standard template library (STL) or Thrust library (Hoberock and Bell, 2010), and their implementation on GPUs has been well-studied (see e.g. Harris et al., 2007; Satish et al., 2009). The advantage of describing algorithms in this way is that we can specify intent without prescribing implementation; the efficient implementation of these primitives in both serial and parallel contexts is well understood, and a single pseudocode description that uses primitives will often suffice for both serial and parallel contexts.

B Standard resampling schemes

B.1 Multinomial resampling

Multinomial resampling proceeds by drawing each a^i independently from the categorical distribution over $\mathcal{C} = \{1, \ldots, N\}$, where $P(a^i = j) = w^j / \textsc{Sum}(\mathbf{w})$. Pseudocode is given in Code 2. The algorithm is dominated by the N calls of $\textsc{Lower-Bound}$, which if implemented with a binary search, will give a serial complexity of $\mathcal{O}(N \log_2 N)$ overall.

---

[*] L.M. Murray (corresponding author) is with the Department of Statistics, University of Oxford.

[†] A. Lee is with the Department of Statistics, University of Warwick.

[‡] P.E. Jacob is with the Department of Statistics, University of Oxford.

Code 1 Pseudocode for various primitive functions.

```
Inclusive-Prefix-Sum($\mathbf{w} \in [0,\infty)^N$) $\to [0,\infty)^N$
    $W^i \leftarrow \sum_{j=1}^{i} w^j$
    return $\mathbf{W}$

Exclusive-Prefix-Sum($\mathbf{w} \in [0,\infty)^N$) $\to [0,\infty)^N$
    $W^i \leftarrow \begin{cases} 0 & i = 1 \\ \sum_{j=1}^{i-1} w^j & i > 1 \end{cases}$
    return $\mathbf{W}$

Adjacent-Difference($\mathbf{W} \in [0,\infty)^N$) $\to [0,\infty)^N$
    $w^i \leftarrow \begin{cases} W^i & i = 1 \\ W^i - W^{i-1} & i > 1 \end{cases}$
    return $\mathbf{w}$

Sum($\mathbf{w} \in [0,\infty)^N$) $\to [0,\infty)$
    return $\sum_{i=1}^{N} w^i$

Lower-Bound($\mathbf{W} \in [0,\infty)^N, u \in [0,\infty)$) $\to \{1,\ldots,N\}$
    requires
        $W$ is sorted in ascending order
    return
        the lowest $j$ such that $u$ may be inserted into
        position $j$ of $W$ and maintain its sorting.
```

The Inclusive-Prefix-Sum operation on line 1 of Code 2 is not numerically stable, as large values may be added to relatively insignificant ones during the procedure (an issue intrinsic to any large summation). With large N, assigning the weights to the leaves of a binary tree and summing with a depth-first recursion over this will help. With large variance in weights, pre-sorting may also help. While log-weights are often used in the implementation of SMC, these need to be exponentiated (perhaps after rescaling) for the Inclusive-Prefix-Sum operation, so this does not alleviate the issue.

Serially, the same approach may be used, although a single-pass approach of complexity $\mathcal{O}(N)$ is enabled by generating sorted uniform random variates (Bentley and Saxe, 1979). Code 3 details this approach. A drawback is the use of relatively expensive logarithm functions. There is scope for a small degree of parallelism in this new algorithm by dividing N among a handful of threads. Each thread must still step through all N weights, however, so that the complexity is not improved with parallelism. We find it faster than Code 2 when on CPU, but slower when on GPU.

Code 2 Pseudocode for parallel multinomial resampling.

```
MULTINOMIAL-ANCESTORS($\mathbf{w} \in [0,\infty)^N$) → $\{1,\dots,N\}^N$
  $\mathbf{W} \leftarrow$ INCLUSIVE-PREFIX-SUM($\mathbf{w}$)
  for each $i \in \{1,\dots,N\}$
      $u^i \sim \mathcal{U}[0, W^N)$
      $a^i \leftarrow$ LOWER-BOUND($\mathbf{W}, u^i$)
  return a
```

Code 3 Pseudocode for serial, single-pass multinomial resampling.

```
MULTINOMIAL-ANCESTORS($\mathbf{w} \in [0,\infty)^N$) → $\{1,\dots,N\}^N$
  $\mathbf{W} \leftarrow$ EXCLUSIVE-PREFIX-SUM($\mathbf{w}$)
  $W \leftarrow W^N + w^N$ // sum of weights

  $lnMax \leftarrow 0$
  $j \leftarrow N$
  for $i = N,\dots,1$
      $u \sim \mathcal{U}[0,1)$
      $lnMax \leftarrow lnMax + \ln(u)/i$
      $u \leftarrow W \exp(lnMax)$
      while $u < W^j$
          $j \leftarrow j - 1$
      $a^i \leftarrow j$
  return a
```

B.2 Stratified resampling

The variance in outcomes produced by the multinomial resampler may be reduced (Douc and Cappé, 2005) by stratifying the cumulative probability function of the same categorical distribution, and randomly drawing one particle from each stratum. This stratified resampler (Kitagawa, 1996) most naturally delivers not the ancestry vector **a** or offspring vector **o**, but the *cumulative offspring* vector, which we denote **O**, and define as $\mathbf{O} = \textsc{Inclusive-Prefix-Sum}(\mathbf{o})$. Pseudocode is given in Code 4. The algorithm is of serial complexity $\mathcal{O}(N)$.

As for multinomial resampling, the INCLUSIVE-PREFIX-SUM operation on line 2 of Code 4 is not numerically stable. The same strategies to ameliorate the problem apply. Line 6 of Code 4 is more problematic. Consider that there may be a j such that, for $i \geq j$, u^{k^i} is not significant against r^i under the floating-point model, so that the result of $r^i + u^{k^i}$ is just r^i. For such i, no random sample is being made within the strata. Furthermore, rounding up on the same line might easily deliver $O^N = N + 1$, not $O^N = N$ as required, if not for the quick-fix use of min. Given that single precision has about seven significant figures in decimal, consider that, with N around

Code 4 Pseudocode for stratified resampling.

```
Stratified-Cumulative-Offspring($\mathbf{w} \in [0, \infty)^N$) → $\{0, \ldots, N\}^N$
u ~ i.i.d. $\mathcal{U}[0, 1)$
W ← Inclusive-Prefix-Sum(w)
for each $i \in \{1, \ldots, N\}$
    $r^i \leftarrow \frac{NW^i}{W^N}$
    $k^i \leftarrow \min(N, \lfloor r^i \rfloor + 1)$
    $O^i \leftarrow \min\left(N, \left\lfloor r^i + u^{k^i} \right\rfloor\right)$
return O
```

one million, almost certainly no $u^{k^i} \in [0, 1)$ is significant against r^i at high i. Note that while pre-sorting weights and summing over a binary tree can help with the numerical stability of the Inclusive-Prefix-Sum operation, it does not help with this latter issue.

B.3 Systematic resampling

The variance in outcomes of the stratified resampler may often, but not always (Douc and Cappé, 2005), be further reduced by using the same random offset within each stratum. This is the *systematic resampler* (equivalent to the *deterministic* method described in the appendix of Kitagawa (1996)). Pseudocode is given in Code 5, which is a simple modification to Code 4. The same complexity and numerical caveats apply to the systematic resampler as for the stratified resampler.

Code 5 Pseudocode for systematic resampling.

```
Systematic-Cumulative-Offspring($\mathbf{w} \in [0, \infty)^N$) → $\{0, \ldots, N\}^N$
$u \sim \mathcal{U}[0, 1)$
W ← Inclusive-Prefix-Sum(w)
for each $i \in \{1, \ldots, N\}$
    $r^i \leftarrow \frac{NW^i}{W^N}$
    $O^i \leftarrow \min(N, \lfloor r^i + u \rfloor)$
return O
```

The resampling algorithms presented here do not constitute an exhaustive list of those in use, for instance *residual resampling* has been omitted (Liu and Chen, 1998). However they are reasonably representative, and can form the building blocks of more elaborate schemes.

C Ancestor permutation for in-place propagation

An ancestry vector may be permuted to satisfy (9) in the main article. A serial algorithm to achieve this is straightforward and given in Code 6. This $\mathcal{O}(N)$ algorithm makes a single pass through the ancestry vector with pair-wise swaps to satisfy the condition.

Code 6 Serial algorithm for permuting an ancestry to ensure condition (9) in the main article.

```
PERMUTE($\mathbf{a} \in \{1, \ldots, N\}^N$)
for $i = 1, \ldots, N$
    if $a^i \neq i$ and $a^{a^i} \neq a^i$
        swap($a^i, a^{a^i}$)
        $i \leftarrow i - 1$ // repeat for new value
ensures
    $\forall i (i \in \{1, \ldots, N\} : o^i > 0 \implies a^i = i)$
```

The simple algorithm is complicated in a parallel context as the pair-wise swaps are not readily serialised without heavy-weight mutual exclusion. In parallel we propose Code 7. This algorithm does not perform the permutation in-place, but instead produces a new vector $\mathbf{c} \in \{1, \ldots, N\}^N$ that is the permutation of the input vector $\mathbf{a}$. It introduces a new vector $\mathbf{d} \in \{1, \ldots, N+1\}^N$, through which, ultimately, $c^i = a^{d^i}$. In the first stage of the algorithm, PREPERMUTE, the thread for element i attempts to claim position a^i in the output vector by setting $d^{a^i} \leftarrow i$. By virtue of the min function on line 3, the element of lowest index always succeeds in this claim while all others contesting the same place fail, and the outcome of the whole permutation procedure is deterministic. This is desirable so that the results of a particle filter are reproducible for the same pseudorandom number seed. For each element i that is not successful in its claim, the thread for i instead attempts to claim d^i, if unsuccessful again then d^{d^i}, then recursively $d^{d^{d^i}}, \ldots$ etc, until an unclaimed place is found.

We offer a proof of the termination of Code 7. First note that PREPERMUTE leaves $\mathbf{d}$ in a state where, excluding all values of $N + 1$, the remaining values are unique. Furthermore, in PERMUTE the conditional on line 4 means that the loop on line 6 is only entered for values of i that are not represented in $\mathbf{d}$.

For each such i, the while loop traverses the sequence $x_0 = i$, $x_n = d^{x_{n-1}}$, until $d^{x_n} = N + 1$. For the procedure to terminate this sequence must be finite. Because each x_n is an element of the finite set $\{1, \ldots, N\}$, to show that the sequence is finite it is sufficient to show that it never revisits the same value twice. The proof is by induction.

Proof.

1. As no value of $\mathbf{d}$ is i, the sequence cannot revisit its initial value $\mathbf{x}_0 = i$. The element x_0 is therefore unique.
2. For $k \geq 1$, assume that the elements of $x_{0:k-1}$ are unique.
3. Now, the elements of $x_{0:k}$ are *not* unique if there exists some $j \in \{1, \ldots, k-1\}$ such that $x_k = d^{x_{k-1}} = x_j = d^{x_{j-1}}$, with $x_{j-1} \neq x_{k-1}$ by the uniqueness of $x_{0:k-1}$. But this contradicts

Code 7 Parallel algorithm for the permutation of an ancestry vector to ensure condition (9) in the main article.

```
PREPERMUTE($\mathbf{a} \in \{1,\ldots,N\}^N$) $\rightarrow \{1,\ldots,N+1\}^N$
Let $\mathbf{d} \in \{1,\ldots,N+1\}^N$ and set $d^i \leftarrow N+1$ for $i = 1,\ldots,N$.
for each $i \in \{1,\ldots,N\}$
    atomic $d^{a^i} \leftarrow \min(d^{a^i}, i)$ // attempt to claim this slot, minimum is winner
ensures
    $\forall i (i \in \{1,\ldots,N\} : o^i > 0 \implies d^i = \min_j (a^j = i))$
return d

PERMUTE($\mathbf{a} \in \{1,\ldots,N\}^N$) $\rightarrow \{1,\ldots,N\}^N$
d $\leftarrow$ PREPERMUTE(a)
for each $i \in \{1,\ldots,N\}$
    $x \leftarrow d^{a^i}$
    if $x \neq i$ // if claim was unsuccessful in PREPERMUTE
        $x \leftarrow i$
        while $d^x \leq N$
            $x \leftarrow d^x$
        $d^x \leftarrow i$

for each $i \in \{1,\ldots,N\}$
    $c^i \leftarrow a^{d^i}$

ensures
    $\forall i (i \in \{1,\ldots,N\} : o^i > 0 \implies c^i = i)$
return c
```

the uniqueness of the (non $N+1$) values of $\mathbf{d}$. Thus the elements of $x_{0:k}$ are unique, the sequence is finite, and the program must terminate.

□

D Auxiliary functions

The multinomial, Metropolis and rejection resamplers most naturally return the ancestry vector $\mathbf{a}$, while the stratified and systematic resamplers return the cumulative offspring vector $\mathbf{O}$. Conversion between these is reasonably straightforward. An offspring vector $\mathbf{o}$ may be converted to a cumulative offspring vector $\mathbf{O}$ via the INCLUSIVE-PREFIX-SUM primitive, and back again via ADJACENT-DIFFERENCE. A cumulative offspring vector may be converted to an ancestry vector via Code 8, and an ancestry vector to an offspring vector via Code 9. These functions perform well on both CPU and GPU. An alternative approach to CUMULATIVE-OFFSPRING-TO-ANCESTORS, using a binary search for each ancestor, was found to be slower.

Code 8 Pseudocode conversion of an offspring vector **o** to an ancestry vector **a**.

```
CUMULATIVE-OFFSPRING-TO-ANCESTORS($\mathbf{O} \in \{0, \dots, N\}^N$) $\rightarrow \{1, \dots, N\}^N$
for each $i \in \{1, \dots, N\}$
    if $i = 1$
        $start \leftarrow 0$
    else
        $start \leftarrow O^{i-1}$

    $o^i \leftarrow O^i - start$
    for $j = 1, \dots, o^i$
        $a^{start+j} \leftarrow i$
return a
```

Code 9 Pseudocode conversion of an ancestor vector **a** to an offspring vector **o**.

```
ANCESTORS-TO-OFFSPRING($\mathbf{a} \in \{1, \dots, N\}^N$) $\rightarrow \{0, \dots, N\}^N$
$\mathbf{o} \leftarrow \mathbf{0}$
for each $i \in \{1, \dots, N\}$
    atomic $o^{a^i} \leftarrow o^{a^i} + 1$
return o
```

E Implementation

All the algorithms described in the article and the appendices have been implemented as part of the LibBi software (`www.libbi.org`, Murray (2013)) for performing methods such as the particle filter on high-performance computing devices. We enumerate the most important considerations of the implementation here, and avoid painstaking detail of the remainder so as not to oversell their importance relative to these. It is worth emphasising that some important decisions, such as the choice of pseudorandom number generator (PRNG), depend on the particular problem at hand.

The weight vector may contain many very small values. Because of this, a typical implementation will store *log-weights* rather than *weights* for numerical accuracy. The log-weights may be large and negative, and one should avoid taking a floating point exponential of these large negative numbers, which is often zero. All of the algorithms presented are robust to the scaling of weights by a constant factor, however. When computing sums or prefix sums, a vector of log-weights can therefore be renormalised using, say, the maximum value, denoted $\log w_{\max}$. For example, the logarithm of the sum of weights, stored as log-weights, is accurately computed using the identity:

$$\log \sum_{i=1}^{N} w^i = \log w_{\max} + \log \sum_{i=1}^{N} \exp(\log w^i - \log w_{\max}).$$

Renormalisation is not required for the Metropolis and rejection algorithms, as they feature only pairwise ratios between weights, or pairwise differences between log-weights.

The performance of the multinomial, stratified and systematic resamplers depends largely on the implementation of the prefix sum operation. We defer to existing work for these operations, in particular to that invested in the Thrust library (Bell and Hoberock, 2012), which the implementation uses. Conceptually, the implementation in the Thrust library is based on up- and down-sweeps of a balanced binary tree (Harris et al., 2007).

The performance of the Metropolis and rejection resamplers is dependent mostly on the selection of PRNG. Performance is not the only consideration in this selection, however. PRNGs are assessed both on execution speed and the statistical quality of the pseudorandom number sequence that they produce, typically using test suites such as DIEHARD (Marsaglia, 1996) or TestU01 (L'Ecuyer and Simard, 2007). Among clients of PRNGs, Monte Carlo algorithms, such as the particle filter, have high demands for statistical quality. To this end, our CPU code uses the Mersenne Twister PRNG (Matsumoto and Nishimura, 1998) as implemented in the Boost.Random library (`www.boost.org`). This is standard for Monte Carlo applications. Our GPU code uses the XORWOW PRNG (Marsaglia, 2003) from the CURAND library (NVIDIA Corporation, 2012). This particular PRNG belongs to a family that is readily shaped to the GPU architecture (Nandapalan et al., 2012). Faster but lower quality PRNG may be used. This would constitute a relaxing of the unbiasedness condition (2). As any such decision is problem-specific, it is not investigated in this work.

The Metropolis and rejection algorithms use random access patterns to memory. Spatiotemporally local access patterns are preferred for good cache performance on CPU, and streaming, or at least coalesced access, is preferred on GPU. The random access pattern is, unfortunately, inherent to the algorithms, and we can only rely on the presence of a large cache to mitigate associated latencies. On GPU, juditious use of shared memory may help, but there is no reason to believe that this can achieve better results than the hardware-controlled cache found on more recent architectures; we rely on the latter. As such, the GPU is configured to use 48 KB of L1 cache and 16 KB of shared memory. This maximises the size of the cache for random access patterns, but still provides sufficient shared memory for all kernels.

Our Metropolis and rejection resampler kernels compile to 32 registers per thread, as reported by the CUDA compiler. This is satisfactory with respect to occupancy of the device, and we do not seek further reductions.

Finally, the auxiliary algorithms presented in §D pose little challenge. Implemented using CUDA, they compile to kernels using no shared memory and fewer than 16 registers per thread, which is of no hindrance to occupancy of the device. On GPU, we append the PREPERMUTE procedure of Code 7 to the end of any procedure that produces an ancestry vector. This saves the launch of a separate kernel and the associated overhead of doing so.

References

Bell, N. and J. Hoberock (2012). Chapter 26 - Thrust: A productivity-oriented library for CUDA. In W. W. Hwu (Ed.), *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pp. 359–371. Boston: Morgan Kaufmann.

Bentley, J. L. and J. B. Saxe (1979). Generating sorted lists of random numbers. Technical Report 2450, Carnegie Mellon University, Computer Science Department.

Douc, R. and O. Cappé (2005, 15-17). Comparison of resampling schemes for particle filtering. In *Image and Signal Processing and Analysis, 2005. ISPA 2005. Proceedings of the 4th International Symposium on*, pp. 64 – 69.

Harris, M., S. Sengupta, and J. D. Owens (2007). *GPU Gems 3*, Chapter Parallel Prefix Sum (Scan) with CUDA. NVIDIA.

Hoberock, J. and N. Bell (2010). Thrust: A parallel template library.

Kitagawa, G. (1996). Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics 5*, 1–25.

L'Ecuyer, P. and R. Simard (2007). TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software 33.*

Liu, J. S. and R. Chen (1998). Sequential Monte-Carlo methods for dynamic systems. *Journal of the American Statistical Association 93*, 1032–1044.

Marsaglia, G. (1996). DIEHARD: a battery of tests of randomness.

Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software 8*(14), 1–6.

Matsumoto, M. and T. Nishimura (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation 8*, 3–30.

Murray, L. M. (2013). Bayesian state-space modelling on high-performance hardware using LibBi. In review.

Nandapalan, N., R. P. Brent, L. M. Murray, and A. P. Rendell (2012). High-performance pseudo-random number generation on graphics processing units. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski (Eds.), *Parallel Processing and Applied Mathematics*, Volume 7203 of *Lecture Notes in Computer Science*, pp. 609–618. Springer–Verlag.

NVIDIA Corporation (2012, July). *CUDA Toolkit 5.0 CURAND Library.* NVIDIA Corporation.

Satish, N., M. Harris, and M. Garland (2009, May). Designing efficient sorting algorithms for manycore GPUs. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–10.