Parallel Particle Learning for Bayesian Financial Data Analysis

Hiroaki Katsura[*] Kenichiro McAlinn[†] Teruo Nakatsuma[‡]

Abstract
Posterior simulation for Bayesian inference using particle filters and particle learning algorithms have proven to be successful in various fields, including finance. However, these particle based methods for posterior simulation are, by nature, computationally strenuous and time consuming. With the recent development of fast and inexpensive devices for parallel computing, such as general purpose graphic processing units (GPGPU), in mind, we have developed a new algorithm for particle filter that is fully parallelized.

Key Words: Bayesian Inference, Particle Learning, Parallel Computing

1. Introduction

This article shows a new parallel version of the particle filter by proposing an exact resampling method that is completely parallel. In the last four decades, state-space modeling, and especially non-Gaussian state-space modeling, have become very popular between researchers and practitioners. Particle filters have been proven to be a very effective method to estimate Bayesian non-Gaussian state-space models. One problem that most researchers and practitioners (but especially practitioners) have found with particle filtering is its time consuming nature. In light of new parallel processing units that are cheap and extremely fast for parallel computing, the already parallel particle filtering methods should benefit greatly from these devises. However, as they are bottlenecks that stymie the algorithm to be completely parallel, we solve these bottlenecks in order to gain the most from parallel devices such as the GPU.

A state-space model is defined as below:

$$y_t \sim p(y_t|x_t) \tag{1}$$
$$x_t \sim p(x_t|x_{t-1}) \tag{2}$$

The particle filter algorithm by Gordon et al. (1993) approximates the posterior distribution $p(x_t|y_{1:t})$ by the following steps:

Step 0: Set the m particles $\{x_0^{(i)}\}_{i=1}^m$

Step 1: Resample $\{\tilde{x}_t^{(i)}\}_{i=1}^m$ from $\{x_t^{(i)}\}_{i=1}^m$ with weight $w_t^{(i)} \propto p(y_{t+1}|x_t^{(i)})$

Step 2: Propagate $x_{t+1}^{(i)}$ from $p(x_{t+1}|\tilde{x}_t^{(i)})$, $(i = 1, \dots, m)$

The main bottleneck of the particle filter algorithm, which is also true for other particle methods, is the resampling process (Step 1). The process requires an algorithm that searches for each particle through the cumulative distribution until it is found. Then repeats the process for each and all particles, which is extremely time consuming. In fact, upon running the particle filter algorithm above on a very simple model, we find that the resampling procedure accounts for 90% of the computation time.

[*]Graduate School of Economics, Keio University, 2-15-45 Mita, Minato-ku, Tokyo, Japan
[†]Graduate School of Economics, Keio University, 2-15-45 Mita, Minato-ku, Tokyo, Japan
[‡]Faculty of Economics, Keio University, Keio University, 2-15-45 Mita, Minato-ku, Tokyo, Japan

Other resampling procedures have been developed in order to circumvent this problem. The most simple method is to sort the particles in ascending order and then resampling the particles sequentially, so rather than counting up from zero each time, each particle can start its search from where the last particle left off. However, the possible time-consuming nature of the sorting algorithm has been pointed out by many. Other resampling methods such as the stratified and systematic resampling arbitrarily sets equally distanced variates that mimic the uniform distribution. The merit of this method is that all variates are already sorted and the number of random numbers that must be generated are limited. On the other hand, as they are not all random variates, in the sense of random sampling from the uniform distribution, these methods are not exact and cannot be parallelized.

2. Parallel Computing with GPGPUs

The key aspect of this research and the motivation for parallelization is the recent development in GPGPU computing. GPGPUs are devices that are designed with massive number of cores to solve single instruction multiple data (SIMD) processing. Originally developed for PC gamers, these GPUs are what produce the graphics for games and videos at an affordable price (even for teenagers) of around a few hundred US dollars.

Since these GPUs are designed mainly for generating graphics for games, their core aim is to calculate the millions of pixels throughout the continuous game play. Calculation per pixel is simple and easy to compute, however, as the number of pixels mount (a basic computer monitor comes with 1920x1080 pixels with 120 frames per second), the computation time can be impossible to handle for a single core architecture such as the CPU.

Scientist quickly caught on to this device especially as NVIDIA, the largest GPU maker and designer in the world, started to shift their development architecture to the more broad general scientific community. The key to this innovation has been the computation unified device architecture (CUDA), which is an extension of the C programming language (NVIDIA, 2011).

The key strength of the GPU is its power over SIMD processing. Table. 1 below shows the comparison between the CPU (the host) and the GPU (the device). GPUs, in this case the GTX580, which is at this point a generation behind, has 128x the core of the CPU but has roughly one third of the computational power.

An additional point that must be mentioned is the price at which these GPU cards are sold on the market. As GPUs are designed and manufactured for gamers all around the world, the price of one is extremely cheap and easy to install. Another appealing point is that there can be more that one GPU on each motherboard (for example, four GPU cards on one motherboard), easily quadrupling the computation power.

	CPU (Intel i7 Extreme)	GPU (GTX580)
Memory	32GB	1.5GB
Cores	4	512
Core clock	2.93GHz	772MHz
Cost(USD)	1,000	100-200

Table 1: Comparing the CPU and GPU hardware

As mentioned earlier, GPUs are designed for SIMD processes, however, not all processes are SIMD. Thus, the key to creating successful algorithms on GPUs is to develop algorithms that are SIMD (such as Durham and Geweke, [2011]) or parallelizing non-SIMD algorithms. This paper takes the later approach by modifying (partly) non-SIMD algorithms to completely fit into the SIMD framework.

There are a few points that one should avoid when developing algorithms on the GPU. Processing sequential algorithms on the GPU can be costly because of the GPU's memory architecture. In general, there are two major types of memories on a GPU; memory that is assigned to each core and memory that is shared between all cores. As parallel devices, access and calculation on each core-linked memory is fast while sharing memory is extremely costly for the computational time, so one should try as much to keep all calculation on each core without communication between cores. Another aspect of the GPU that is costly is the memory transfer between the host and the device. The developed code should keep all calculations on the GPU rather than transferring processes back and forth. An ideal algorithm for GPGPU devices would be to calculate everything in parallel (without communication between cores) and completing all calculations on the GPU, which we succeed in doing for the parallel particle filter algorithm.

The actual programming on CUDA is exactly like programming in C. The only exception is that there are few additional lines that send the code and data from the host to the device and then to retrieve the results from the device to the host. Libraries for CUDA have been developed, which are sufficient for statistical computing (e.g. number of random number generators), and MATLAB also has a parallel computing tool box, not to mention third party jackets that can be used with conjecture to other statistical software.

3. Parallel Particle Filtering

As everything about the particle filtering algorithm is SIMD except for the resampling procedure, modifying the resampling procedure (the bottleneck) to fit the SIMD framework makes the whole algorithm parallel. With this in mind, we have succeeded in developing an algorithm that enables the resampling procedure to be processed in parallel using the cut-point method by Chen and Asau (1974).

The goal of resampling is to generate m random integers with replacement from a discrete distribution on $\{1, \ldots, m\}$ with the cumulative distribution function

$$q(i) = \Pr\{X \leq i\}, \quad (i = 1, \ldots, m) \tag{3}$$

which is proportional to the posterior density. Direct application of the inverse transform method to resampling is rather time-consuming when the number of particles m is large. Instead, we propose to use the cut-point method. A *cut-point* I_j for given $j = 1, \ldots, m$ is the smallest index i such that the corresponding probability (3) should be greater than $(j-1)/m$. Alternatively, I_j is defined as

$$q(I_j) = \min_{1 \leq i \leq m} q(i) \quad \text{subject to} \quad mq(i) > j - 1, \quad (j = 1, \ldots, m) \tag{4}$$

Then the resampling algorithm with the cut-point method is given as follows.

Step 0: Let $j = 1$

Step 1: Generate u from the uniform distribution on the interval $(0, 1)$.

Step 2: Let $k = I_{\lceil mu \rceil}$ where $\lceil x \rceil$ stands for the smallest integer greater than or equal to x.

Step 3: If $u > q(k)$, let $k \leftarrow k + 1$ and repeat Step 3; otherwise, go to Step 4.

Step 4: Store k as $i_{(j)}$.

Step 5: If $j < m$, let $j \leftarrow j + 1$ and go back to Step 1; otherwise, exit the loop.

The resultant $\{i_{(1)}, \dots, i_{(m)}\}$ are random indices independently drawn from the discrete distribution $\{q(1), \dots, q(m)\}$.

Once the cut-points $\{I_1, \dots, I_m\}$ are given, parallel execution of the above algorithm is straightforward because the execution of Step 1 – Step 3 does not depend on the index j. On the other hand, the standard algorithm for computation of the cut-points is not friendly to parallel execution. In this paper, we propose an alternative approach for parallel execution of the cut-point method. First, let us define

$$L_j = \lceil mq(j) \rceil, \quad (j = 1, \dots, m)$$

and $L_0 = 0$ for convention. Due to the monotonicity of the cumulative distribution function, we observe

1. $0 = L_0 < L_1 \leq \cdots \leq L_m = m$.
2. If $L_{j-1} < L_j$, a cut-point such that

   $$q(I_k) = \min_{1 \leq i \leq m} q(i) \quad \text{subject to} \quad mq(i) > k - 1, \quad (k = L_{j-1} + 1, \dots, L_j)$$

   is given as $I_k = j$.
3. If $L_{j-1} = L_j$, j does not correspond with any cut-points.

The above properties give us a convenient criterion to check whether a particular L_j is a cut-point or not, and it leads to the following multi-thread algorithm to find all cut-points.

Step 0: Initiate the j-th thread.

Step 1: Compute $L_j = \lceil mq(j) \rceil$.

Step 2: If $L_{j-1} < L_j$, let $k = L_j$ and $I_k = j$; otherwise, end the thread.

Step 3: Let $k \leftarrow k - 1$.

Step 4: If $k > L_{j-1}$, let $I_k = j$ and go to Step 3; otherwise, end the thread.

The number of iterations in the loop Step 3 – Step 4 will be modest unless $q(j) - q(j-1)$ is extremely high.

4. Results

In our computational example, we use 100 simulated data sets and compute a very simple state space model shown below:

$$\begin{aligned} y_t &= x_t + \nu_t, \quad \nu_t \sim \mathcal{N}(0, \sigma^2) && (5) \\ x_t &= x_{t-1} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \tau^2) && (6) \end{aligned}$$

In order to estimate the above model, we use the particle learning method suggested by Carvalho et al.(2010). The results of three types of simulation, GPU (parallel resampling), CPU (ordinary resampling), and CPUs(sorting the i.i.d. uniform variates) are shown below in Table 2. and Figure 1.

The results are straightforward. It is clear that by developing an algorithm that completely runs in parallel and keeps all calculations on the device can be extremely effective

Table 2: Computational Time of the Particle Filter with Parallel Resampling on the GPU, Ordinary Resampling on the CPU, and Ordinary Resampling with Sort on the CPU

Particles ($\times$100)	1,024	4,096	16,384	131,072	1,048,576	8,388,608
GPU(msec)	10.879	16.107	63.333	257.063	2,110.560	19,876.252
CPU(msec)	30.000	80.000	390.000	7,050.000	307,600.000	19,855,870.000
CPUs(msec)	60.000	210.000	850.000	6,960.000	58,170.000	528,800.000
CPU$\times$GPU	2.758	4.967	6.158	27.425	145.743	998.975
CPUs$\times$GPU	5.515	13.038	13.421	27.076	27.561	26.605
CPU$\times$CPUs	0.500	0.381	0.459	1.013	5.288	37.549

compared to sequential algorithms. As the particles increase (and the precision of the estimate increase), the sheer parallel power of the GPU overpowers that of the CPU by ten to the hundreds. Even when compared to the ordinary resampling with ordered variates, the GPU is faster than the CPU by ten to twenty times. What should be noted is that the GPU results can be easily doubled or tripled with an investment of a few hundred USD while there is little room for the CPU to run any faster.

4.1 Conclusion

By fully parallelizing the resampling procedure, using the parallel cut-point method, we have succeeded in achieving computational differences in the hundreds. This in mind, researchers are now able to compute real-time posterior distributions using exact resampling in complete parallel for many different applications; ranging from high-frequency trading to motion sensing technology.

Additionally, as CPU computation speed is heading towards its limit, new, cheap, and faster hardware, like the GPGPU, is now looked upon as a resolution for solving complex problems in a reasonable time. By solving (successfully parallelizing) bottlenecks in algorithms, this paper has shown that the results can be staggering compared to traditionally effective sequential algorithms.

Future research includes extending the state-space model to more complex models, such as the multi-factor stochastic volatility model, and apply it to actual high-frequency financial data to examine its effectiveness.

REFERENCES

C.Carvalho, M.Johannes, H.Lopes, N.Polson. ”Particle learning and smoothing.” Statistical Science, 25 (2010), pp. 88-106.

H.C.Chen, Y.Asau. ”On generating random variates from an empirical distribution.” American Institute of Industrial Engineers (AIIE) Transactions, 6(1974), pp.163-166.

J.Geweke, G.B.Durham. ”Massively Parallel Sequential Monte Carlo for Bayesian Inference.” Working paper series, 2011.

N.Gordon, D.Salmond, A.Smith. ”Novel approach to nonlinear/non-Gaussian Bayesian state estimation.” IEEE Proceedings-F, 140(1993), pp.107-113.

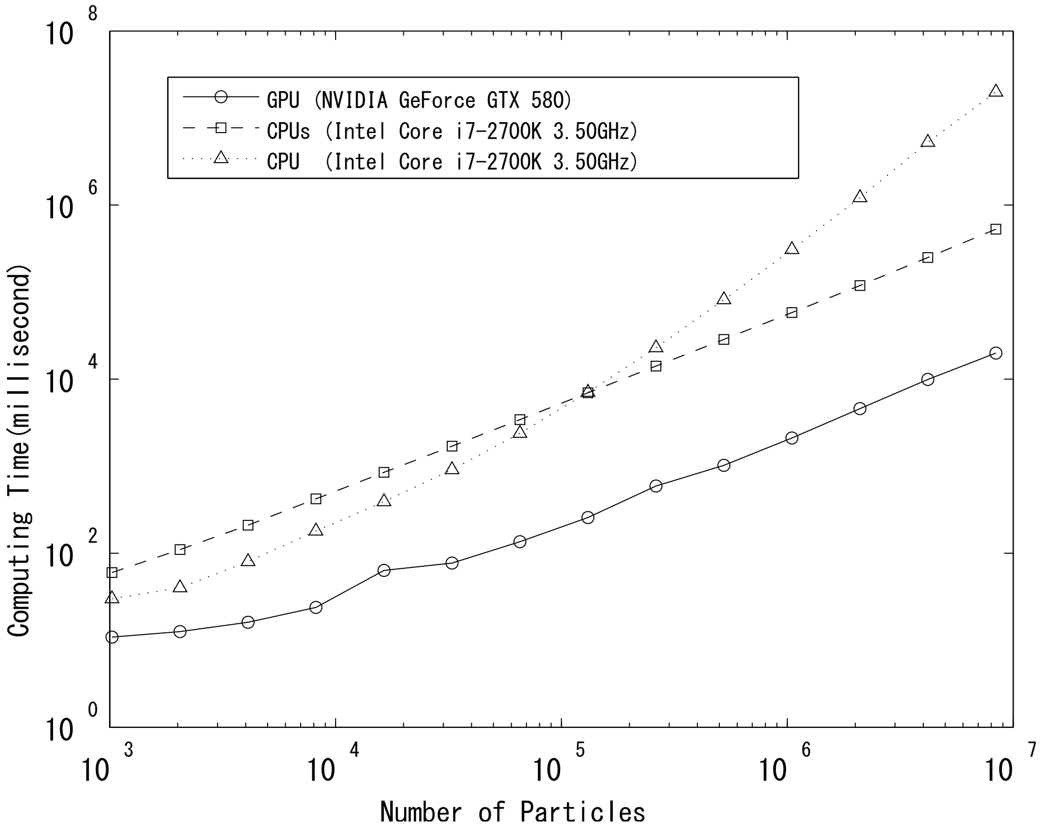

Figure 1: Computational Time of the Particle Filter with Parallel Resampling on the GPU, Ordinary Resampling on the CPU, and Ordinary Resampling with Sort on the CPU