A Fast Parallel Particle Filter for Shared Memory Systems

Alessandro Varsi*, Jack Taylor†, Lykourgos Kekempanos‡, Edward Pyzer Knapp§, Simon Maskell*
*Department of Electrical Engineering and Electronics, University of Liverpool, Liverpool, L69 3GJ, UK
†Hartree Centre, Science and Technology Facilities Council (STFC), Sci-Tech Daresbury, Warrington, WA4 4AD, UK
‡BT Applied Research, Orion Building, Adastral Park, Martlesham, Ipswich, IP5 3RE, UK
§IBM Research UK, Sci-Tech Daresbury, Warrington, WA4 4AD, UK
{Alessando.Varsi*, S.Maskell*}@liverpool.ac.uk, jack.a.taylor@stfc.ac.uk†, lykourgos.kekempanos@bt.com‡, EPyzerK3@uk.ibm.com§

***Abstract*—Particle Filters (PFs) are Sequential Monte Carlo methods which are widely used to solve filtering problems of dynamic models under Non-Linear Non-Gaussian noise. Modern PF applications have demanding accuracy and run-time constraints that can be addressed through parallel computing. However, an efficient parallelization of PFs can only be achieved by effectively parallelizing the bottleneck: resampling and its constituent redistribution step. A pre-existing implementation of redistribute on Shared Memory Architectures (SMAs) achieves $O(\frac{N}{T}log_2N)$ time complexity over T parallel cores. This redistribute implementation is, however, highly computationally intensive and cannot be effectively parallelized due to the inherently limited number of cores of SMAs. In this paper, we propose a novel parallel redistribute on OpenMP 4.5 which takes $O(\frac{N}{T}+log_2N)$ steps and fully exploits the computational power of SMAs. The proposed approach is up to six times faster than the $O(\frac{N}{T}log_2N)$ one and its implementation on GPU provides a further three-time speed-up vs its equivalent on a 32-core CPU. We also show on an exemplary PF that our redistribution is no longer the bottleneck.**

***Index Terms*—Parallel Particle Filters, Shared Memory Architectures, OpenMP, Resampling, Redistribute.**

I. Introduction

Particle Filters (PFs) are a well-established family of algorithms to perform state estimations of a dynamic model, given a stream of measurements. The filtering technique consists of using an arbitrary proposal distribution to draw N samples (i.e. particles) which approximate the probability density function (pdf) of the state of the model. The application domain is then vast and diverse since it can range, for example, from positioning [1] to economics [2] and risk analysis [3].

Modern applications of PFs have demanding accuracy constraints which can be met in several ways: one could increase the number of particles [4] or could apply more sophisticated proposal distributions [5] or use more measurements if possible [6]. Applying any of these solutions, however, is likely to significantly slow down the run-time which could also become even more problematic if the constraint on the measurement rate is strict. In order to meet the run-time constraints without losing accuracy, PFs need to employ parallel computing [7].

PFs are parallelizable but deriving an efficient parallelization is not trivial. The resampling step, which is used to handle the particle degeneracy [8], is notoriously hard to parallelize. This is due to the problems encountered in parallelizing the constituent redistribute step whose textbook implementation achieves $O(N)$ time complexity on one core. One could also use a Multi-PF approach which, however, has accuracy, scalability and applicability limitations as shown in [9]–[11].

In [12], redistribution was parallelized in a fully-balanced fashion using divide-and-conquer to recursively sort and split the particles. Since [12] uses Bitonic Sort log_2N times, the achieved time complexity is $O((log_2N)^3)$ (not $O((log_2N)^2)$ as claimed in [12]). In [13], the time complexity has been reduced to $O((log_2N)^2)$ by proving that sort is only needed once. In [14] [15], the same idea is ported from MapReduce to MPI. Here, as in [12]–[15], we consider N to be fixed.

While sort is required on Distributed Memory Architectures (DMAs), it is optional on Shared Memory Architectures (SMAs) as it can be substituted with N Binary Searches. This idea was implemented in [4], optimized in [16] and applied to different resampling schemes in [17]. The time complexity is $O(\frac{N}{T}log_2N)$ for T parallel cores. Since N could be large, due to demanding accuracy constraints [4], modern SMAs cannot provide enough cores to optimally perform redistribution.

In this paper, we want to redesign the redistribute parallelization in order to optimize the time complexity and fully exploit the computational power of the modern CPU/GPU. The programming model we use is OpenMP 4.5 as it can be straightforwardly configured to work on either CPU or GPU.

The rest of the paper is organized as follows: in Section II we give information about SMAs and OpenMP. In Section III, we briefly describe PFs and redistribute and their parallel implementation on SMAs. In Section IV, we introduce our novel redistribute. In Section V, we provide numerical results on an exemplary PF on both CPU and GPU. In Section VI, we draw our conclusions and give suggestions for future work.

II. Shared Memory Architectures

SMAs are a type of parallel system which are fundamentally different to DMAs. In these architectures, the cores can simultaneously access the same memory which can be used to share information between the threads.

The main advantage over DMAs is therefore fast communication between the cores since SMAs achieve this by simply reading from/writing to the system-wide shared memory. On the other hand, SMAs have two main disadvantages relative to DMAs: memory access increases with the number of cores and the largest systems that use DMAs are bigger than the largest that use SMAs alone.

Any shared memory API is suitable for this research. We choose OpenMP for its simple and intuitive syntax which makes it one of the most popular programming models for SMAs. OpenMP applies the fork-join model to set up T parallel threads which are uniquely identified by an $id \in \mathbb{Z}$. Once created, the threads can concurrently execute bodies of instructions which are coded within directives for the compiler called pragmas. OpenMP also supports "reduction" operations and provides a SIMD clause for explicit vectorization [18]. For the purposes of this paper, we prefer OpenMP 4.5 which is the first OpenMP release to support mainstream CPUs and GPU offload. We use Clang 7.0 as in [19].

III. Particle Filters

A wide range of PF methods exist. In this section, we briefly explain Sequential Importance Resampling (SIR) PF whose pseudo-code is described by Algorithm 1. A detailed explanation can be found in [8] [9].

Algorithm 1 SIR Particle Filter

Input: T_{PF}, N, N^*
Output: $\mathbf{f}_t$

$\mathbf{x}_0, \mathbf{w}_0 \leftarrow$ `Initialize()`, $\mathbf{x}_0 \sim p(\mathbf{x}_0)$ and $\mathbf{w}_0^i \leftarrow \frac{1}{N}\ \forall i$
for $t \leftarrow 1$; $t \leq T_{PF}$; $t \leftarrow t+1$ **do**
  $\mathbf{Y}_t \leftarrow$ `New Measurement()`
  $\mathbf{x}_t, \mathbf{w}_t \leftarrow$ `Importance Sampling()`, see (1) and (2)
  $\tilde{\mathbf{w}}_t \leftarrow$ `Normalize`$(\mathbf{w}_t)$, see (3)
  $N_{eff} \leftarrow$ `ESS`$(\tilde{\mathbf{w}}_t)$, see (4)
  if $N_{eff} < N^*$ **then** perform Resampling
    $\mathbf{ncopies} \leftarrow$ `MVR`$(\tilde{\mathbf{w}}_t)$, Minimum Variance Resampling
    $\mathbf{x}_t \leftarrow$ `Redistribute`$(N, \mathbf{ncopies}, \mathbf{x}_t)$
    $\mathbf{w}_t \leftarrow$ `Reset`$(\mathbf{w}_t)$, $\mathbf{w}_t^i \leftarrow \frac{1}{N}\ \forall i$
  end if
  $\mathbf{f}_t \leftarrow$ `Mean`$(\mathbf{x}_t)$, see (7)
end for

Algorithm 2 Sequential Redistribute (S-R)

Input: N, $\mathbf{ncopies}$, $\mathbf{x}$, $base$
Output: $\mathbf{x}_{new}$

$i \leftarrow base$, $j \leftarrow 0$, $base$ must be 0 on a single core run
while $i < N$ **do**
  for $k \leftarrow 0$; $k < \mathbf{ncopies}^j$; $k \leftarrow k+1$ **do**
    $\mathbf{x}_{new}^i \leftarrow \mathbf{x}^j$
    $i \leftarrow i+1$
  end for
  $j \leftarrow j+1$
end while

A. SIR Particle Filter

PFs employ the Importance Sampling (IS) principle to make Bayesian inferences on the state $\mathbf{X}_t \in \mathbb{R}^M$ of a dynamic model. To do that, N statistically independent particles $\mathbf{x}_t \in \mathbb{R}^{N \times M}$ are drawn from a user-defined proposal distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ at every given iteration t. The population of particles represents the pdf of the state of the model. Each particle $\mathbf{x}_t^i$ is then assigned an unnormalized importance weight $\mathbf{w}_t^i$, such that the array of weights $\mathbf{w}_t \in \mathbb{R}^N$ gives information on which particle best resembles the true $\mathbf{X}_t$. Details of how to proceed at each time step follow.

A new measurement $\mathbf{Y}_t \in \mathbb{R}^D$ is collected at each time step t. In SIR Filter, the particles are initially generated from the prior distribution $q(\mathbf{x}_0) = p(\mathbf{x}_0)$ and then sampled from the proposal distribution as follows:

$$\mathbf{x}_t^i \sim q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{Y}_t) \tag{1}$$

The weights are initialized to $1/N$ and then computed as

$$\mathbf{w}_t^i = \mathbf{w}_{t-1}^i \frac{p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)p(\mathbf{Y}_t|\mathbf{x}_t^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{Y}_t)} \tag{2}$$

After that, they are normalized as follows:

$$\tilde{\mathbf{w}}_t^i = \mathbf{w}_t^i / \sum_{j=0}^{N-1} \mathbf{w}_t^j \tag{3}$$

The particles, however, suffer from a phenomenon called degeneracy which (within a few iterations) makes all weights but one decrease towards 0. This is because the variance of the weights is proven to increase at every iteration [8] [9]. Degeneracy can be addressed in different ways. The most common is to perform a resampling step which repopulates the particles by eliminating the most negligible ones and duplicating the most important ones. In the SIR PF, resampling is only triggered when the effective sample size

$$N_{eff} = 1/\sum_{i=0}^{N-1} (\tilde{\mathbf{w}}_t^i)^2 \tag{4}$$

decreases below a threshold N^* (which is usually set to $N/2$).

Different (biased or unbiased) resampling schemes exist [17] [20]–[22] but they mostly follow a three-step approach. The first step is to process the normalised weights $\tilde{\mathbf{w}}_t$ to generate $\mathbf{ncopies} \in \mathbb{Z}^N$ such that $\mathbf{ncopies}^i$ indicates how many copies of the i-th particle must be created. Therefore, it is easy to infer that

$$\sum_{i=0}^{N-1} \mathbf{ncopies}^i = N \tag{5}$$

The second step is redistribution which all resampling algorithms have in common and is in charge of duplicating each particle the right number of times. A textbook sequential redistribute (S-R) can be found in Algorithm 2 which takes $O(N)$ steps as (5) holds. After redistributing, all weights are reset to $1/N$. Previous referenced work has used Minimum Variance Resampling (MVR) to perform the first step [12] [14] [16]. Since our focus is mostly on redistribution, MVR will be the only variant we consider. MVR first computes $\mathbf{cdf} \in \mathbb{R}^{N+1}$, the Cumulative Density Function (CDF) of the weights, then it draws a random sample $u \sim [0, 1)$ from a uniform distribution and then computes each $\mathbf{ncopies}^i$ as follows:

$$\mathbf{ncopies}^i = \lceil \mathbf{cdf}^{i+1} - u \rceil - \lceil \mathbf{cdf}^i - u \rceil \tag{6}$$

where the bracket operator represents the ceiling function (e.g. $\lceil 5.1 \rceil = 6$). At the end of each time step, estimates are produced as follows:

$$\mathbf{f}_t = \frac{1}{N} \sum_{i=0}^{N-1} \mathbf{x}_t^i \tag{7}$$

B. Parallel Particle Filters

Weight reset, (1), (2) and (6) are element-wise operations and hence trivially parallelizable. On OpenMP they can be parallelized by `#pragma omp parallel for` directives.

The sum is computed in (3), (4) and (7). Reduction is the classic way to parallelize Sum and takes $O(\frac{N}{T} + log_2 T)$ operations. On OpenMP, reduction can be used by adding a `reduction` clause applied to the variable to reduce.

The CDF of the weights requires Cumulative Sum which is also commonly known as Prefix Sum or Scan. Cumulative Sum is parallelizable and achieves $O(\frac{N}{T} + log_2 T)$ time complexity. A recent implementation of parallel Cumulative Sum on GPU can be found in [23].

S-R has a low constant time (it only consists of N memory writes) but is impossible to parallelize in an element-wise fashion. Each $\mathbf{ncopies}^i$ could randomly be equal to any integer between 0 and N and hence, the workload could be highly unbalanced. An alternative advanced parallel redistribution for SMAs on CPU and GPU can be found in [4] [16]. The idea is to make use of $\mathbf{csum} \in \mathbb{Z}^N$, the Cumulative Sum of $\mathbf{ncopies}$, to search for the particles to duplicate. Each i-th particle to copy can indeed be found by using Binary Search over $\mathbf{csum}$ to search for the first index j such that $\mathbf{csum}^j \geq i$. While it is not explicit in [4] [16], to infer $\mathbf{csum}$ it is unnecessary to perform again parallel Cumulative Sum, as one could recycle the information in $\mathbf{cdf}$ and apply (6) as follows:

$$\mathbf{csum}^i = \sum_0^i \mathbf{ncopies}^i = \lceil \mathbf{cdf}^{i+1} - u \rceil - \lceil \mathbf{cdf}^0 - u \rceil \quad (8)$$

Algorithm 3 briefly summarizes these steps. The time complexity is $O(\frac{N}{T} log_2 N)$ as each core needs to perform up to N Binary Searches. When N is large, due to accuracy constraints, the workload on each core is significant. As we will see in Section V, Algorithm 3 hardly shows any speed-up vs S-R. In the next section, we develop a novel parallel redistribution that only requires one Binary Search per thread and takes advantage of the fast constant time of S-R.

Algorithm 3 Reference Parallel Redistribute

Input: N, $\mathbf{ncopies}$, $\mathbf{x}$, $\mathbf{cdf}$, u, T
Output: $\mathbf{x}_{new}$

```
#pragma omp parallel for num_threads(T)
for i ← 0; i < N; i ← i + 1 do
    j ←Binary Search(cdf, ncopies, u, i), search
        for the first j s.t. csum^j ≥ i, see (8)
    x^i_new ← x^j
end for
```

IV. Novel Parallel Redistribute

S-R cannot be parallelized directly as the workload could be unevenly distributed. Therefore, the approach we propose is to first balance the workload across the T cores, such that S-R could be then invoked with a fast and scalable time complexity equal to $O(N/T)$. To do that, one needs to fully exploit the information which is stored in $\mathbf{csum}$ and $\mathbf{ncopies}$.

Each thread (uniquely identified by an integer $0 \leq id \leq T-1$) has to generate N/T particle copies, given the instructions provided by $\mathbf{ncopies}$. However, since $\mathbf{ncopies}$ complies by definition with (5), it is always possible to identify T indexes, called pivots, between which the workload across the cores is equally divided. To also have a balanced data partitioning, each thread will have to place its N/T copies starting from index $i = id \times \frac{N}{T}$ in the output array. Therefore, each thread's pivot p is the first index such that $\mathbf{csum}^p \geq id \times \frac{N}{T}$.

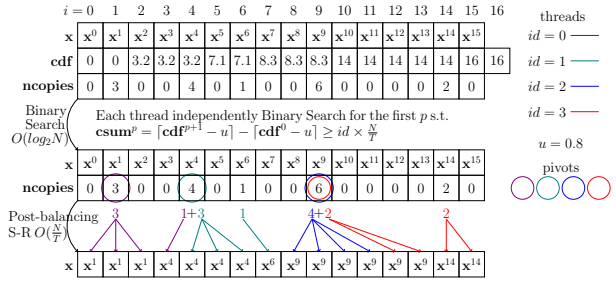


Fig. 1: Novel redistribute - Example for $N = 16$ and $T = 4$.

The threads can simultaneously find their pivot by calling Binary Search once. Since multiple threads may happen to share the same pivot, each thread must figure out how many copies of the particle $\mathbf{x}^p$ it has to create. This can be computed in constant time as the min value between $\mathbf{csum}^p - id \times \frac{N}{T}$ and $\frac{N}{T}$. That is because if two or more threads share the same pivot, only the thread with the highest id must create less than $\frac{N}{T}$ copies of $\mathbf{x}^p$. After that, the workload is balanced and the threads can independently produce their N/T particle copies by calling S-R. Since Binary Search and S-R are performed once, we can infer that the time complexity is $O(\frac{N}{T} + log_2 N)$.

Figure 1 illustrates a practical example for $N = 16$ and $T = 4$ and Algorithm 4 provides an OpenMP-like description of our novel parallel redistribute. For completeness, we denote that an OpenMP 4.5 algorithm can target GPU by adding `target teams distribute map` clauses to the existing `#pragma omp parallel for` directives. Further details, e.g. optimizations of the Random Number Generators used in (1) or the use of SIMD in all algorithms of this paper whenever possible are omitted for brevity due to page limit constraints.

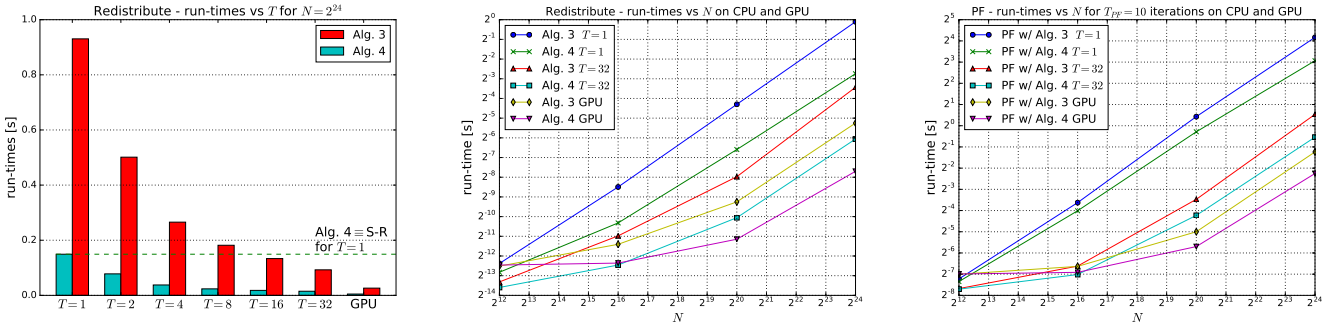
Algorithm 4 Novel Parallel Redistribute

Input: N, $\mathbf{ncopies}$, $\mathbf{x}$, $\mathbf{cdf}$, u, T
Output: $\mathbf{x}_{new}$

```
if T == 1 then
    x_new ← S-R(N, ncopies, x, 0), see Algorithm 2
else
    #pragma omp parallel num_threads(T){
        id ←omp_get_thread_num(), base ←id × N/T
        p ← Binary Search(cdf, ncopies, u, base),
            search for the first p s.t. csum^p ≥ base, see (8)
        n ← min(csum^p − base, N/T), see (8) for csum^p
        x^base_new, x^{base+1}_new, ..., x^{base+n−1}_new ← x^p_new
        x_new ← S-R(N/T − n, ncopies + p, x + p, p + 1)
    }
end if
```

V. Numerical Results

In this section, we first compare single iterations of Algorithms 3 and 4 and then two PFs working on the same model, both running $T_{PF} = 10$ iterations but differing for the constituent parallel redistribute. The two PFs are compared in



(a) Redistribute - best run-times vs T (b) Redistribute - best run-times vs N (c) PF - best run-times vs N

Fig. 2: Numerical results

the worst-case scenario which occurs when IS is relatively fast and resampling is invoked at every iteration for both PFs. We note that in other applications, where IS (which scales more quickly than redistribute) is slower than here, redistribution will always become the bottleneck for some bigger T.

All results are medians of 20 runs for up to $N = 2^{24}$ akin to [4] and different T. The CPU we use is a 2 Xeon Gold 6138 which provides up to $T = 32$ cores and for the GPU results we employ an NVIDIA Tesla V100 which has 5120 cores.

To generate the worst-case scenario, we choose the classic benchmark stochastic volatility model which estimates the evolution of the pound-to-dollar exchange rate between October 1, 1981 and June 28, 1985 [2]. The dynamic model follows:

$$\mathbf{X}_t = \phi \mathbf{X}_{t-1} + \sigma \mathbf{V}_t \tag{9a}$$

$$\mathbf{Y}_t = \beta \exp(\mathbf{X}_t/2)\mathbf{W}_t \tag{9b}$$

where the coefficients $\phi = 0.9731$, $\sigma = 0.1726$, $\beta = 0.6338$ and the noise terms are $\mathbf{V}_t \sim \mathcal{N}(0,1)$ and $\mathbf{W}_t \sim \mathcal{N}(0,1)$. The initial state is sampled as $\mathbf{X}_0 \sim \mathcal{N}(0, \frac{\sigma^2}{1-\phi^2})$. The particles are initially drawn from the prior distribution. (2) becomes $\mathbf{w}_t^i = \mathbf{w}_{t-1}^i p\left(\mathbf{Y}_t|\mathbf{x}_t^i\right)$ since the dynamics is used as the proposal.

The two parallel redistribution steps have been tested on the same inputs generated from a PF working on the model in (9a) and (9b). Figure 2a shows their scalability for $N = 2^{24}$ and increasing degree of parallelism, while Figure 2b shows the most significant run-times on CPU and GPU for increasing N. In Figure 2a, we can observe that Algorithm 3 for $T < 32$ provides little to no speed-up vs Algorithm 4 on a single core (i.e. S-R). However, Algorithm 4 achieves substantial speed-up for any $T > 1$. In Figure 2b, we can see that using a GPU in place of a CPU gets more effective as N increases, since the host-to-device transfer time is dominant over the computation for small N. Overall, both Figures 2a and 2b highlight that Algorithm 4 is up to six times faster than Algorithm 3 on CPU and GPU. Algorithm 4 on GPU also gives about a three-fold speed-up vs its CPU best run-time.

Figure 2c shows the results for the PFs using either Algorithm 3 or 4 and Figure 3 illustrates how much run-time is taken up by each task for $T_{PF} = 10$ iterations, for $T = 32$ cores and on GPU. We can see that, while Algorithm 3 still accounts for 63% of the whole run-time, Algorithm 4 is no longer the bottleneck. The overall performance of the PF has improved by up to a factor of 2.1. We denote that the theoretical maximum speed-up is $1/(1 - 0.63) = 2.7$ for a 63% bottleneck. On GPU, we can observe again over a three-fold speed-up vs the best run-time on CPU.

VI. Conclusions

In this paper, we have designed a novel parallel algorithm for redistribute, the bottleneck of PFs, on OpenMP 4.5. The baseline for comparisons is an advanced current implementation which achieves $O(\frac{N}{T} log_2 N)$ time complexity. Our approach has $O(\frac{N}{T} + log_2 N)$ time complexity and is up to six times faster on a 32-core CPU and on a modern NVIDIA Tesla V100 GPU. On an exemplary PF, for which the reference redistribution takes up to 63% of the total run-time, our redistribution, which is no longer the bottleneck, speeds up the overall performance by more than a factor of two. On GPU, the performance is over three times as fast as its equivalent on a 32-core CPU.

Future work should focus on applying the achieved improvements to enhance the performance of existing MPI or MPI+OpenMP implementations of PFs to fully exploit the computational power of modern supercomputers. Future work should also investigate implementation on other hardware (e.g. TPUs) and time-varying N as considered in [21] [24].

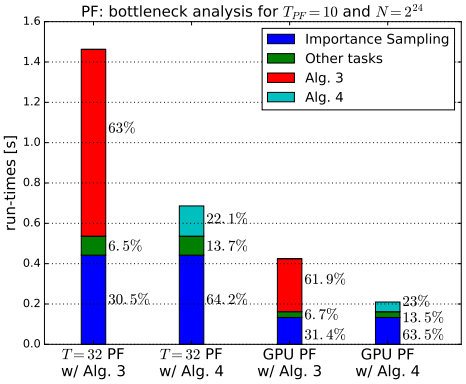


Fig. 3: Workload distribution for $N = 2^{24}$ on CPU and GPU.

Acknowledgements

This work was supported by the UK EPSRC Doctoral Training Award, the UK EPSRC Big Hypothesis award, Schlumberger and the STFC Hartree Centre's Innovation Return on Research programme, funded by the Department for Business, Energy & Industrial Strategy. The authors also acknowledge Dr. Jeyarajan Thiyagalingam, Dr. Luke Mason and Professor Vassil Alexandrov from STFC for the technical advices.

REFERENCES

[1] F. Gustafsson, "Particle Filter Theory and Practice with Positioning Applications," *IEEE Aerospace and Electronic Systems Magazine*, vol. 25, pp. 53–82, July 2010.

[2] A. Doucet, M. Briers, and S. Sncal, "Efficient Block Sampling Strategies for Sequential Monte Carlo Methods," *Journal of Computational and Graphical Statistics*, vol. 15, no. 3, pp. 693–711, 2006.

[3] Z. Wu and S. Li, "Reliability Evaluation and Sensitivity Analysis to AC/UHVDC Systems Based on Sequential Monte Carlo Simulation," *IEEE Transactions on Power Systems*, vol. 34, pp. 3156–3167, July 2019.

[4] F. Lopez, L. Zhang, J. Beaman, and A. Mok, "Implementation of a Particle Filter on a GPU for Nonlinear Estimation in a Manufacturing Remelting Process," in *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 340–345, July 2014.

[5] C. A. Naesseth, F. Lindsten, and T. B. Schön, "High-Dimensional Filtering Using Nested Sequential Monte Carlo," *IEEE Transactions on Signal Processing*, vol. 67, pp. 4177–4188, Aug 2019.

[6] J. Zhang and H. Ji, "Distributed Multi-Sensor Particle Filter for Bearings-only Tracking," *International Journal of Electronics - INT J ELECTRON*, vol. 99, pp. 239–254, 02 2012.

[7] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes, "On the Utility of Graphics Cards to Perform Massively Parallel Simulation of Advanced Monte Carlo Methods," *Journal of Computational and Graphical Statistics*, vol. 19, no. 4, pp. 769–789, 2010.

[8] M. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002.

[9] A. Doucet and A. Johansen, "A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later," *Handbook of Nonlinear Filtering*, vol. 12, 01 2009.

[10] O. Demirel, I. Smal, W. Niessen, E. Meijering, and I. Sbalzarini, "PPF - A Parallel Particle Filtering Library," *IET Conference Publications*, vol. 2014, 10 2013.

[11] P. Djuric, T. Lu, and M. Bugallo, "Multiple Particle Filtering," vol. 3, pp. III–1181, 05 2007.

[12] S. Maskell, B. Alun-Jones, and M. Macleod, "A Single Instruction Multiple Data Particle Filter," in *IEEE Nonlinear Statistical Signal Processing Workshop*, pp. 51–54, 2006.

[13] J. Thiyagalingam, L. Kekempanos, and S. Maskell, "Mapreduce Particle Filtering with Exact Resampling and Deterministic Runtime," *EURASIP Journal on Advances in Signal Processing*, vol. 2017, p. 71, Oct 2017.

[14] A. Varsi, L. Kekempanos, J. Thiyagalingam, and S. Maskell, "Parallelising Particle Filters with Deterministic Runtime on Distributed Memory Systems," *IET Conference Proceedings*, pp. 11–18, 2017.

[15] A. Varsi, L. Kekempanos, J. Thiyagalingam, and S. Maskell, "A Single SMC Sampler on MPI that Outperforms a Single MCMC Sampler," *ArXiv*, May 2019.

[16] F. Lopez, L. Zhang, A. Mok, and J. Beaman, "Particle Filtering on GPU Architectures for Manufacturing Applications," *Computers in Industry*, vol. 71, pp. 116 – 127, 2015.

[17] L. M. Murray, A. Lee, and P. E. Jacob, "Parallel Resampling in the Particle Filter," *Journal of Computational and Graphical Statistics*, vol. 25, no. 3, pp. 789–805, 2016.

[18] R. van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*. The MIT Press, 1st ed., 2017.

[19] G. Özen, S. Atzeni, M. Wolfe, A. Southwell, and G. Klimowicz, "OpenMP GPU Offload in Flang and LLVM," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pp. 1–9, Nov 2018.

[20] J. D. Hol, T. B. Schon, and F. Gustafsson, "On Resampling Algorithms for Particle Filters," in *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pp. 79–82, Sept 2006.

[21] T. Li, M. Bolic, and P. M. Djuric, "Resampling Methods for Particle Filtering: Classification, Implementation, and Strategies," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 70–86, 2015.

[22] T. Li, G. Villarrubia, S.-d. Sun, J. Corchado Rodrguez, and J. Bajo, "Resampling Methods for Particle Filtering: Identical Distribution, a New Method, and Comparable Study," *Frontiers of Information Technology & Electronic Engineering*, vol. 16, pp. 969–984, 11 2015.

[23] Y. Mitani, F. Ino, and K. Hagihara, "Parallelizing Exact and Approximate String Matching via Inclusive Scan on a GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1989–2002, July 2017.

[24] D. Fox, "KLD-Sampling: Adaptive Particle Filters," in *Advances in neural information processing systems*, pp. 713–720, 2002.