ALGORITHMIC MODIFICATION OF PARTICLE FILTERS FOR HARDWARE IMPLEMENTATION

Miodrag Bolić, Akshay Athalye, Petar M. Djurić, and Sangjin Hong

Department of Electrical and Computer Engineering
Stony Brook University
Stony Brook, NY, 11794-2350
mbolic, athalye, djuric, snjhong@ece.sunysb.edu

ABSTRACT

Particle filters are sequential Monte Carlo methods that have recently gained popularity in solving various problems in communications and signal processing. These filters have been shown to outperform traditional filters in important practical scenarios. However, they are computationally intensive and hence development of hardware for their real time implementation is an important and challenging research issue. In this paper we present some novel modifications applied to two particle filtering algorithms viz. Sampling Importance Resampling Filters (SIRFs) and Gaussian Particle Filters (GPFs) to make these filters suitable for implementation. We evaluate the proposed algorithms with respect to potential throughput and hardware resources. These modifications allow implementation of parallel architectures for these filters. Architectural parameters of proposed architectures for these filters are evaluated and compared.

1. INTRODUCTION

Particle filters (PFs) [1, 2] are used to perform filtering for problems that can be described using dynamic state space modeling [1]. In most practical scenarios, these models are non-linear and the densities involved are non-Gaussian. Traditional filters like the Extended Kalman Filter are known to perform poorly in such scenarios. The performance of PFs on the other hand, is not affected by these conditions. PFs are Bayesian in nature and their goal is to find an approximation to the posterior density of the states of interest (e.g. position of a moving object in tracking, or transmitted symbol in communications) based on observations corrupted by additive noise which are inputs to the filter. This is done using the principle of Importance Sampling (IS) whereby, samples (particles) are drawn from a known density (Importance Function (IF)) and assigned appropriate weights based on the received observations using IS rules [1]. This weighted set of samples represents the posterior density of the state and can be used to find all kinds of estimates of the state (like minimum mean square error MMSE or maximum a posteriori MAP). PF algorithms allow for recursive propagation of this density as the observations become available. However, performance of this scheme is affected by weight degeneracy [2]. Because of this, after several sampling periods there are only a few particles with significant weights while those of the rest become negligible. This problem is solved by introducing resampling which discards particles with negligible weights and replicates those with large weights while preserving constant number of particles. These operations form the traditional PF algorithm known as the Sampling Importance Resampling Filter (SIRF). Recently, a new type of PF known as Gaussian Particle Filter (GPF) has been introduced [3]. GPF works by approximating the desired densities as Gaussians, and calculating the estimates of the first two moments of these Gaussians using a particle based approach. These estimates are propagated in time. GPFs do not suffer from weight degeneracy and hence do not require standard resampling.

PFs have been the focus of wide research recently and immense literature can be found on their theory. Most of these works recognize the complexity and computational intensity of these filters, but there has not been much effort directed towards the implementation of these filters in hardware. In this paper, algorithmic transformations that lead to increased concurrency and saving resources for both SIRF and GPF are presented. Architectures for their implementation are proposed and evaluated based on their sampling period, memory requirements, data exchange patterns and complexity of sequential operations. In order to maximize throughput, only spatial design with independent hardware units for each operation is considered.

Often, in practical operations, a large number of particles need to be used for computing estimates of the desired state. As the number of particles increases, the input sampling period and hence the speed of the PF is seriously affected. The algorithms presented here were applied to the bearings only tracking problem presented in [4]. Using 2000 particles on a single state-of-the-art DSP, yielded speeds of up to 500Hz for SIRF. Clearly, this speed would seriously limit the range of applications for which the PF can be used for real time processing. Hence for meeting speed requirements of real time applications, it is necessary to have high throughput designs with ability to process a larger number of particles in a given time. Parallelizability is the key to high throughput design for PFs, as this enables simultaneous processing of particles. GPFs show great promise for parallel implementation. GPFs have lower sampling periods, simple data exchange patterns, and they do not require memories for storing particles. From a theoretical viewpoint, the prior densities are used as IFs for both the SIRFs and GPFs respectively since this allows for a fair comparison of these filters.

2. PARALLEL ARCHITECTURAL MODEL

The common architecture for both SIRFs and GPFs consists of multiple processing elements (PEs) simultaneously carrying out independent operations on particles and a central unit (CU) performing data dependent sequential operations. The operation of the CU is dependent on the result of the

This work has been supported by the NSF under Award CCR-0220011.

operation of the PEs and vice versa. Hence the PEs and the CU cannot operate simultaneously. Once all the particles are processed, the PEs send relevant data to the CU depending on which algorithm is being implemented. The CU starts operating after receiving all the data from the PEs while the PEs remain idle during this CU operation time. Once the CU completes its operation , the results are sent to the PEs and the next recursion is started. The parallel model of PFs is shown in Figure 1.

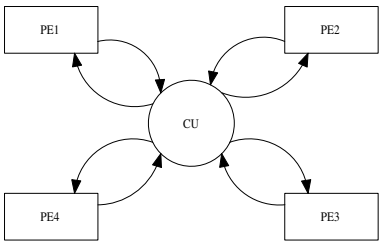


Figure 1: Architecture of the parallel PF with a CU and four PEs.

3. ALGORITHMIC MODIFICATIONS FOR SIRF

Detailed description of the traditional SIRF algorithm can be found in [1].

In one SIRF recursion at time instant n, the following operations are performed:

• Sampling step – Generation of new particles, in which M particles $x_n^{(m)}$ for $m = 1,...,M$ are drawn from an importance function $\pi(x_n)$ using the particles $x_{n-1}^{(m)}$ propagated from the previous time instant.

• Importance step – Computation of the particle weights $\tilde{w}_n^{(m)}$ for $m = 1,...,M$, and their normalization according to $w_n^{(m)} = \tilde{w}_n^{(m)}/W_n$, where W_n represents the sum of weights.

• Resampling step – drawing of M particles $\widetilde{x}_n^{(m)}$ from the set $x_n^{(m)}$ for $m = 1,...,M$ according to a resampling function $a_n^{(m)}$ whose support is defined by the particles $x_n^{(m)}$. Commonly $a_n^{(m)} = w_n^{(m)}$ for $m = 1,...,M$.

We modify the traditional SIRF algorithm to improve its parallel spatial implementation by avoiding normalization and applying distributed resampling. Traditional algorithms apply systematic resampling (SR) with normalized weights. Normalization can be eliminated by using sum of weights in the resampling step. A modified SR algorithm which works with non-normalized weights is shown by Pseudocode 1. Using this algorithm M divisions during normalization are replaced with one division as shown in step 2. Here, N_i is the input number of particles, N_o is the number of particles generated after resampling, and $\widetilde{w}_n$ is an array of non-normalized weights from the importance step. In non-parallel implementation $N_i = N_o = M$. The output i_n is an array of indexes, which shows the addresses of the particles that are replicated. The algorithm works by drawing a uniform random number U from the support $[0, \frac{W_n}{N_o}]$ and then updating it using step 10. At the same time, the sum of the first k particle weights S is calculated and compared with U. When $S < U$ the last particle is discarded and the weight of the particle $k+1$ is added to S. If $S > U$ the particle k is replicated and the number of replications is proportional to $E(\widetilde{w}_n^{(k)} N_o/W_n)$.

1. (i_n)=SR(N_i,N_o,W_n, $\widetilde{w}_n$)
2. $A_d = \frac{W_n}{N_o}$
3. Generate random number $U \sim \mathscr{U}[0, A_d]$
4. $S = 0$, $k = 0$
5. **for** $m = 1 : N_i$
6. **while** $(S < U)$
7. $k = k + 1$
8. $S = S + \widetilde{w}_n^{(k)}$
9. **end**
10. $U = U + A_d$
11. $i_n^{(m)} = k$
12. **end**

Pseudocode 1. Systematic resampling with non-normalized weights.

Propagation and weight calculation for different particles are independent and each require M iterations for one PF recursion. This allows for parallel implementation which is based on the concept of loop level parallelism [5]. Resampling, which is inherently sequential, has been modified in order to allow for parallel implementation [6]. Resampling is then performed concurrently in the PEs such that each PE, after resampling, produces a certain number of particles proportional to its sum of weights, $W_n^{(k)}$, of particles in that PE. Resampling consists of three operations:

1. CU resampling which is a sequential operation in which the CU first calculates the number of particles $N^{(k)}$ that each PE should produce after resampling based on its sum of weights $W_n^{(k)}$ for $k = 1,...,K$ where K is the number of PEs.
2. Simultaneous execution of resampling in the PEs once they get the numbers $N^{(k)}$. If resampling is performed based on Pseudocode 1, the input number of particles is equal for all the PEs $N_i = M/K$ and the output number of particles varies $N_o = N^{(k)}$.
3. Data exchange in which the particles among PEs are exchanged in a way that PEs with the surplus send the particles to the PEs with lack of particles. This step is necessary in order to assure that all the PEs have the same number of particles before the next sampling period.

Using this modification, the time for resampling in parallel implementation is reduced K times in comparison with the implementation in which resampling is performed only by the CU. The average time for data exchange is reduced as well. In parallel implementations, sample generation, weight computation, and particle replication (item 2. above) are mapped into the PEs, while the sequential operations such as CU resampling and data exchange are mapped into the CU.

4. ALGORITHMIC MODIFICATIONS FOR GPF

The traditional GPF algorithm can be found in [3]. In this section, the GPF algorithm is modified to allow for overlapping of operations and implementation without storing particles into memories. If the IF is the prior, the GPF can be implemented using four steps: (a) drawing conditioning particles $\mathbf{x}_{n-1}$ from the approximated filtering density of the previous recursion, (b) generating new particles $\mathbf{x}_n$, (c) calculating weights and normalizing them, and (d) computing

the mean μ_n and covariance $\mathbf{\Sigma}_n$ of the filtering density. Steps (b) and (c) are same as those in SIR. These four steps can be executed using four loops each having M iterations. To fuse these loops and pipeline the operation of GPF, it is necessary to modify the algorithm by eliminating dependencies. These modifications are shown by Pseudocode 2.

The modified algorithm at time instant n combines into one loop the above mentioned step (a), (b) and parts of steps (c) and (d) (steps 1 to 4 in Pseudocode 2). The dependence that exists in step (c) and (d) (the calculation of mean and covariance requires normalized weights) is eliminated by calculating the mean and the variance elements with the non-normalized weights (step 4 in Pseudocode 2) and appropriately scaling the elements by the sum of weights at the end of M iterations (step 6). In order to calculate the covariance coefficients, the mean should be known in step (d). However, a simple transformation of this step is possible so that the mean can also be added at the end of M iterations (step 7). After calculating the covariance matrix, it is decomposed to $\mathbf{C}_{n-1}$ before starting the next recursion.

In these filters, only the mean and variance of the densities are propagated. Since the particles themselves are not propagated, they do not need to be stored in memories. Hence in Pseudocode 2, the superscript (m) is not shown for the particle weights and states as is done in traditional algorithms. In a parallel implementation with K PEs, where each PE processes M/K particles, steps 1 to 4 are mapped in the PEs for parallel implementation since there are no dependencies between these steps. Steps 5-8 are mapped to the CU. In the pseudocode, quantities with superscript (k) represent the partial results of operations of the $k-$th PE. Step 5 is necessary because the partial sum of the weights, the mean and covariance matrix have to be collected and updated from all the PEs.

Purpose: GPF iteration at time instance n where $n > 0$.
Input: The observation $\mathbf{y}_n$ and previous estimates μ_{n-1} and matrix $\mathbf{C}_n$, s.t. $\mathbf{\Sigma}_{n-1} = \mathbf{C}_{n-1}\mathbf{C}_{n-1}^T$
Setup: Mean μ_0 and covariance matrix $\mathbf{\Sigma}_0$ based on prior information, sum of the weights $W = 0$, initial mean $\mu_n^k = 0$ and covariance $\mathbf{\Sigma}_n^k = 0$
Method:

PE operation
for $m = 1$ **to** M
1. Draw a conditioning particle from $\mathcal{N}(\mathbf{x}_{n-1}; \mu_{n-1}, \mathbf{\Sigma}_{n-1})$ to obtain $\mathbf{x}_{n-1}$.
2. Draw a sample from $p(\mathbf{x}_n \mid \mathbf{x}_n = \mathbf{x}_{n-1})$ to obtain $\mathbf{x}_n$.
3. (a) Calculate a weight by $\tilde{w}_n = p(\mathbf{y}_n|\mathbf{x}_n)$.
   (b) Update the current sum of weights by $W = W + \tilde{w}_n$.
4. Update the current mean and covariance by
   (a) $\mu_n^k = \mu_n^k + \tilde{w}_n \mathbf{x}_n$
   (b) $\mathbf{\Sigma}_n^k = \mathbf{\Sigma}_n^k + \tilde{w}_n \mathbf{x}_n (\mathbf{x}_n)^T$.

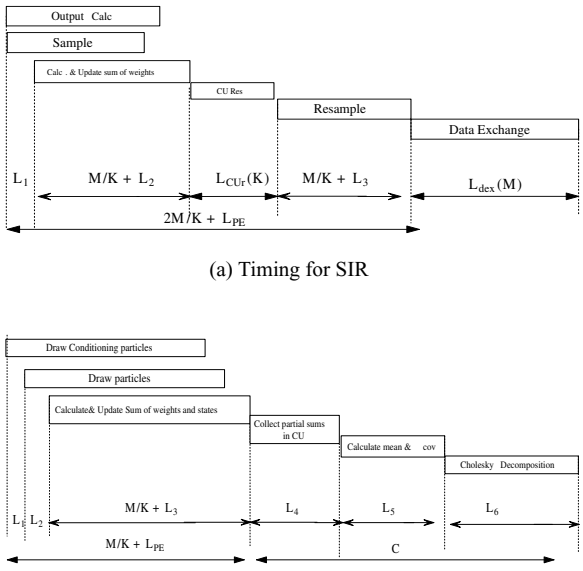
end
CU operation
for $k = 1$ **to** K
5. Collect and update sum of weights, mean and covariance
   (a) $W_n = W_n + W_n^k$.
   (b) $\mu_n = \mu_n + \mu_n^k$
   (c) $\mathbf{\Sigma}_n = \mathbf{\Sigma}_n + \mathbf{\Sigma}_n^k$

end
6. Scale mean and covariance
   (a) $\mu_n = \mu_n / W_n$
   (b) $\mathbf{\Sigma}_n = \mathbf{\Sigma}_n / W_n$
7. Update the covariance estimate $\mathbf{\Sigma}_n = \mathbf{\Sigma}_n - \mu_n(\mu_n)^T$
8. The Cholesky decomposition of the matrix $\mathbf{\Sigma}_n$ in order to obtain $\mathbf{C}_n$.

Pseudocode 2. GPF algorithm after loop fusion is applied.

5. ARCHITECTURAL PARAMETERS

5.1 Sampling period



(a) Timing for SIR

(b) Timing for GPF.

Figure 2: Latency and timing of various steps of particle filter with K PEs (a) SIR (b) GPF

The timing diagrams in Figure 2 show the latencies of various operations in the SIR and GPF filters. In the SIR filter with resamapling distributed to the PEs, the latency of processing one input observation will be $\frac{2M}{K} + L_{PE} + L_{dex}(M)$, where $\frac{2M}{K} + L_{PE}$ is the latency of processing in the PEs. This is because each PE processes M/K particles, and it generates new particles and computes their weights. After the sum of weights of each PE is obtained, the CU takes these sums and computes the number of particles that each PE must produce after resampling [6]. The latency for this CU resampling is a function of the number of PEs represented by $L_{CUr}(K)$. Once this processing is done, another M/K cycles are needed to perform resampling in each PE. The term $L_{PE} = \sum_{i=1}^{3} L_i + L_{CUr}(K)$ accounts for the start up latencies of each block and is inherent to any pipelined feed forward data path. The $L_{dex}(M)$ represents the latency of data exchange after particle allocation, which is done to redistribute particles after resampling. Since data exchange is done through the CU in SIR, increasing the number of particles will reduce the processing speed because the maximum number of particles that is transferred is a function of M.

The net latency of processing is $\frac{M}{K} + L_{PE} + C$ for the GPF. The latency $L_{PE} = \sum_{i=1}^{3} L_i$ accounts for the start up latencies

of the various blocks inside the PEs. The processing time for the CU in the GPF is constant $C = \sum_{i=4}^{6} L_i$ and it accounts for the latency of both the CU and the data exchange between the PEs and the CU.

In Figure 3 the sampling periods of GPFs and SIRs for varying number of particles and PEs are plotted. It can be clearly seen that for a large number of particles and PEs, the GPFs are faster than the SIRFs. Also, the latency L_{dex} for SIRFs is not taken into account. With increasing the number of particles, the latency of the PEs in GPFs becomes dominant while the latency of the CU and that of data exchange remains the same. On the contrary, in SIRFs, as particles increase, the latency of the PEs increases and that of the CU also increases. The results are based on component delays on a Field Programmable Gate Array (FPGA) platform.

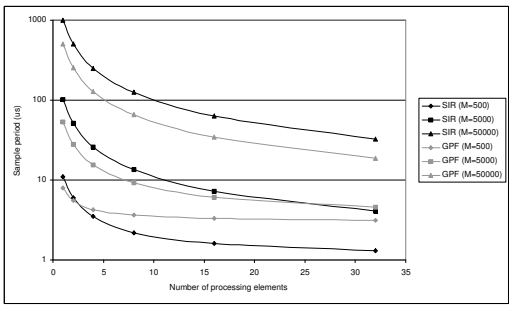


Figure 3: Sampling period vs. number of PEs for SIR and GPF for different number of particles.

5.2 Memory requirements

As the model dimension N_s increases, so do the number of memories used for storing particles and the number of particles necessary for proper functioning of the PFs as shown in [7]. SIRFs require memories for storing particles, their weights and indexes after resampling, so that, the total number of required memory words is $(N_s + 2) \cdot M$. GPFs can be implemented without storing samples and their weights between successive recursion which is one of their main advantages. This greatly reduces the memory requirement of the design. Due to the need to store particles in memory, the number of particles that can be used in SIRFs is limited by the size of the memory. In GPFs, however, any number of particles can be used.

5.3 Data exchange patterns

Data exchange patterns are evaluated based on the number of data transferred and the type and direction of data exchange. In GPFs, from Pseudocode 2, we see that the CU acquires $K \cdot (N_s + 1) \cdot (N_s/2 + 1)$ coefficients of μ_n^k and $\mathbf{\Sigma}_n^k$ for $k = 1, ..., K$ and sends back to PEs $(N_s + 1) \cdot N_s/2$ coefficients of μ_n and $\mathbf{C}_n$. The number of data exchanged for these filters is fixed, the direction is known before run-time and the type is static.

For SIRFs, we consider that data exchange after resampling is performed through the CU. PEs with surplus of particles after resampling $(N^k > M/K)$ donate their exceess particles to the PEs with lack of particles $(N^k < M/K)$. The direction of the data exchange and the amount of particles transferred between the CU and each PE is not known before run-time. It also changes after each sampling period because the distribution of the weights changes. In the worst case, the PE with lack of particles receives all the M/K new particles and the maximum number of particles exchanged through the interconnections is $(K-1) \cdot M/K$. So, resampling introduces non-deterministic and complicated data exchange patterns between the PEs and the CU.

6. CONCLUSION

In this paper, algorithmic modifications of SIRFs and GPFs suitable for parallel hardware implementation are presented. High speed PFs can be realized in hardware using these modified algorithms. This will enable use of PFs in real time practical scenarios where they currently cannot be used. We also analyzed how these modifications affect the architectural parameters of PFs, mainly their speed and resources. To summarize, the GPF has some attractive properties which are the result of algorithmic transformations:

1. The data exchange pattern in GPFs with multiple PEs are simple and deterministic. On the other hand, resampling in SIRFs introduces non-deterministic and complicated data exchange patterns between the PEs and the CU.
2. GPFs can be implemented without storing particles between successive recursions. This greatly reduces the memory requirement in their design as opposed to that of traditional PFs.
3. The GPFs also have CUs whose operations are sequential with data dependencies. The time for the CU operations, however, is fixed and independent of the number of used particles.

REFERENCES

[1] A. Doucet, S. J. Godsill, and C. Andrieu, "On sequential Monte Carlo methods for Bayesian filtering," *Statistics and Computing*, pp. 197–208, 2000.

[2] A. Doucet, N. de Feritas, and N. Gordon, Eds., *Sequential Monte Carlo Mehods in Practice*, Springer Verlag, New York, 2001.

[3] J. H. Kotecha and P. M. Djurić, "Gaussian particle filtering," *IEEE Transactions on Signal Processing*, vol. 51, no. 10, pp. 2592–2601, Oct 2003.

[4] N. J. Gordon, D.J. Salmond, and A. F. M. Smith, "Novel approach to non linear/non Gaussian Bayesian state estimation," *IEEE Proceedings-F*, vol. 140, no. 2, pp. 107–113, April 1993.

[5] J. Henessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, third edition, 2003.

[6] M. Bolić, P. M. Djurić, and S. Hong, "Resampling algorithms and architectures for distributed particle filters," Submitted to *IEEE Transactions on Signal Processing*.

[7] F. Daum and J. Huang, "Curse of dimensionality and particle filters," in *Fifth ONR/GTRI Workshop on Target Tracking and Sensor Fusion*, Newport, RI, June 2002.