# Adapting Particle Filter Algorithms to Many-Core Architectures

Mehdi Chitchian*, Alexander S. van Amesfoort*, Andrea Simonetto†, Tamás Keviczky† and Henk J. Sips*
E-mail: *mehdi.chitchian@gmail.com*, {*a.s.vanamesfoort, a.simonetto, t.keviczky, h.j.sips*}*@tudelft.nl*
*Parallel and Distributed Systems Group, Delft University of Technology, Delft, The Netherlands.
†Delft Center for Systems and Control, Delft University of Technology, Delft, The Netherlands.

***Abstract*—The particle filter is a Bayesian estimation technique based on Monte Carlo simulation. It is ideal for non-linear, non-Gaussian dynamical systems with applications in many areas, such as computer vision, robotics, and econometrics. Practical use has so far been limited, because of steep computational requirements.**

**In this study, we investigate how to design a particle filter framework for complex estimation problems using many-core architectures. We develop a robotic arm application as a highly flexible estimation problem to push estimation rates and accuracy to new levels. By varying filtering and model parameters, we evaluate our particle filter extensively and derive rules of thumb for good configurations.**

**Using our robotic arm application, we achieve a few hundred state estimations per second with one million particles. With our framework, we make a significant step towards a wider adoption of particle filters and enable studies into filtering setups for even larger estimation problems.**

***Index Terms*—Particle Filter; Bayesian Estimation; Many-Core; OpenCL; CUDA**

## I. Introduction

Estimating the state of a dynamical system through noisy measurements has many applications in science and engineering. The most common uses are to localize or track moving objects using computer vision [1], GPS [2], radar or sonar sensors. Other dynamical systems can be found in econometrics [3] and rare event simulation [4].

Often the system of interest is not perfectly understood, but can be modeled as a Markov process. Also, many of the quantities that constitute the state of a system cannot be observed directly, but need to be *inferred* from noisy measurements. Hence, there is uncertainty present in the system model and in the measurement model. For systems where the amount of non-linearity is limited, parametric filters such as the extended or the unscented Kalman filter can be used. Although limited to Gaussian measurement noise, these filters can produce accurate estimates at reasonable computational requirements. For state estimation in (highly) non-linear or non-Gaussian dynamical systems, particle filters provide the most accurate estimates given sufficiently large particle populations.

In short, a particle filter is a sequential Monte Carlo simulation that stochastically generates many potential system states, referred to as particles. Measurements taken from the dynamical system are used to assign weights to particles. The filter output is a single estimate derived from the weighted particles. Often, the particle set is then resampled before the procedure is repeated in the next time step. Unfortunately, the greater flexibility of particle filters comes with much higher computational requirements than Kalman filters, limiting practical use in (real-time) estimation problems.

This paper presents a comprehensive study on how to design and implement high-performance particle filters on many-core architectures (i.e. GPGPUs and multi-core CPUs). For our dynamical system, we use a realistic (industrial) robotic arm tracking a moving object along a known path to measure estimation speed and accuracy. We identify the following contributions for the work presented in this paper:

- We present a fully distributed particle filtering algorithm suitable for many-core architectures. All operations are performed locally on a limited amount of data.
- With our robotic arm tracking application, we present experimental results across a broad range of hardware platforms and number of particles. We achieve update rates of up to 200 Hz with one million particles on a medium sized estimation problem and scale the problem size further.
- We identify various filtering parameters and present the effects of varying them on the trade-off between estimation speed and accuracy. Using these results, we present guidelines for good filtering configurations.
- For the resampling step, we compare two algorithms: Roulette Wheel Selection (RWS) and Vose's alias method. With its lower complexity, Vose's is much faster than RWS for a large centralized filter, but turns out slower when resampling small, local filters.
- Our application separates generic particle filtering from model-specific routines. New dynamical system models can be easily added to further investigate particle filter configurations. The developed "Esthera" toolkit sources are available at https://github.com/alxames/esthera

Hereafter, Section II introduces Bayesian estimation and particle filtering. The design space for parallel particle filters and relevant literature are discussed in Section III, followed by our filtering algorithm in Section IV. Two different many-core architectures, GPGPUs and multi-core CPUs, are briefly introduced in Section V. After some implementation details in Section VI, we present our experiments in Section VII and an evaluation of the filter accuracy and performance in Section VIII. Section IX concludes our work.

## II. Background

### A. Bayesian Estimation

*Bayesian* estimation computes a *Probability Density Function* for the state of a dynamical system over the range of possible values. Suppose $\mathbf{x}$ is a quantity which we wish to infer from the measurement $\mathbf{z}$. The probability distribution $p(\mathbf{x})$ represents all our knowledge regarding this quantity *prior* to the actual measurement. This distribution is, therefore, called the *prior probability distribution*. The conditional probability $p(\mathbf{x} \mid \mathbf{z})$, called the *posterior probability distribution*, represents our knowledge of $\mathbf{x}$ having incorporated the measurement data. This distribution is usually unknown in advance as a result of the complex dynamics involved in most systems. The inverse probability $p(\mathbf{z} \mid \mathbf{x})$, however, directly relates to the measurement characteristics. *Bayes rule* allows us to calculate a conditional probability based on its inverse.

In order to discuss how the *Bayes filter* [5] calculates the state estimate, we first need to model the dynamics of the system. Let $\mathbf{x}_k$ denote the state at time $k$, and $\mathbf{z}_k$ denote the set of all measurements acquired at time $k$. Assuming the system exhibits Markov properties, the state $\mathbf{x}_k$ depends only on the previous state $\mathbf{x}_{k-1}$. Therefore, the evolution of the state is governed by the probability distribution $p(\mathbf{x}_k \mid \mathbf{x}_{k-1})$, which is referred to as the *state transition probability*. The measurements of the state follow the probability distribution $p(\mathbf{z}_k \mid \mathbf{x}_k)$ which is called the *measurement probability*.

The Bayes filter calculates the state estimate, from an initial state $p(\mathbf{x}_0)$, recursively in two steps:

Predict In this step, the state estimate from the previous step is used to predict the current state. This estimate is known as the *a priori* estimate, as it does not incorporate any measurements from the current time step.

$$p(\mathbf{x}_k) = \int p(\mathbf{x}_k \mid \mathbf{x}_{k-1})\, p(\mathbf{x}_{k-1} \mid \mathbf{z}_{k-1})\, d\mathbf{x}_{k-1}$$

Update The state estimate from the previous step is updated according to the actual measurements done on the system. Therefore, this estimate is referred to as the *a posteriori* estimate.

$$p(\mathbf{x}_k \mid \mathbf{z}_k) = \eta\, p(\mathbf{z}_k \mid \mathbf{x}_k)\, p(\mathbf{x}_k)$$

The Bayes filter, in its basic form, is inapplicable to any complex estimation problem. The main obstacle is that the prediction step requires an integration in closed form, which is only possible for basic estimation problems.

### B. Particle Filtering

Particle filtering [6] [7] is a recursive Bayesian filtering technique using Monte Carlo simulation. Particle filters represent the posterior by a finite set of random samples drawn from the posterior with associated weights. Because of their non-parametric nature, particle filters are not bound to a particular distribution form (e.g. Gaussian) and are compatible with arbitrary (i.e. non-linear) state transition functions.

As previously mentioned, particle filters represent the posterior by a set of particles. Each particle $\mathbf{x}_k^{[m]}$ can be considered as an instantiation of the state at time $t$. In the prediction step of the particle filter, each particle $\mathbf{x}_k^{[m]}$ is generated from the previous state $\mathbf{x}_{k-1}^{[m]}$ by sampling from the state transition probability $p(\mathbf{x}_k \mid \mathbf{x}_{k-1})$. In the update step, when measurement $\mathbf{z}_k$ is available, each particle is assigned a weight $\omega_t^{[m]}$ according to:

$$\omega_t^{[m]} = p(\mathbf{z}_k \mid \mathbf{x}_k^{[m]})$$

Given a large enough particle population, the weighted set of particles $\{\mathbf{x}_k^{[i]}, \omega_t^{[i]}, i = 0, \ldots, N\}$ becomes a discrete weighted approximation of the true posterior $p(\mathbf{x}_k \mid \mathbf{z}_k)$.

*1) The Degeneracy Problem and Resampling:* A common problem with the basic particle filter algorithm mentioned previously is the *degeneracy* problem. It has been shown that the variance of the weights can only increase over time [8]. This results in a situation where, after only a few iterations, a single particle holds the majority of the weight with the rest having negligible weight. This results in wasted computational effort on particles which eventually contribute very little to the filter estimation.

*Resampling* is a statistical technique which can be used to combat the degeneracy problem. Resampling involves eliminating particles with small weights in favor of those with larger weights. This is achieved by creating a new set of particles by sampling with replacement from the original particle set according to particle weights. Particles with a higher weight will, therefore, have a higher chance of surviving the selection process. One of the implications of the resampling step is the loss of diversity among particles as the new particle set most likely contains many duplicates.

**Algorithm 1** Particle Filter with Resampling

```
Input: $X_{k-1}$, $\mathbf{z}_k$
Output: $X_k$
 1: $X'_k \leftarrow \emptyset$
 2: foreach $\mathbf{x}_k^{[i]} \in X_{k-1}$ do
 3:     sample $\mathbf{x}_k^{[i]} \sim p(\mathbf{x}_k \mid \mathbf{x}_{k-1}^{[i]})$
 4:     $\omega_k^{[i]} \leftarrow p(\mathbf{z}_k \mid \mathbf{x}_k^{[i]})$
 5:     $X'_k \leftarrow X'_k \cup \{(\mathbf{x}_k^{[i]}, \omega_k^{[i]})\}$
 6: end for
 7: $X_k \leftarrow \emptyset$
 8: for $i \leftarrow 1, |X'_k|$ do
 9:     draw $\mathbf{x}_k^{[r]}$ with probability $\propto \omega_k^{[r]}$
10:     $X_k \leftarrow X_k \cup \{\mathbf{x}_k^{[r]}\}$
11: end for
```

Algorithm 1 gives an overview of the particle filter algorithm with resampling. The first for loop (lines 2 through 6) generates, for each particle $i$, state $\mathbf{x}_k^i$ based on $\mathbf{x}_{k-1}^i$ (line 3) and assigns a weight according to the measurement $\mathbf{z}_k$ (line 4). The second for loop (lines 8 through 11) transforms the particle set $X'_k$ into a new set $X_k$ by resampling according to the weights. The resampling step resets the weights for the whole particle population.

## III. Related Work

In this section we discuss related work and put it in the context of the design space of parallel particle filter algorithms.

### A. Design Space

The qualities to evaluate a particle filter are estimation accuracy and execution speed. Speed can be plain throughput for offline simulations, while achievable update rate is more important for real-time systems. Unfortunately, there are no common system and measurement models (benchmarks) in use, so performance numbers reported in different studies are difficult to compare. A good parallelization strategy may use a modified particle filter algorithm as long as it can deliver the same estimation accuracy faster (or the same speed with better accuracy). This may or may not require more particles or other adaptations, but there is always a trade-off between accuracy and speed.

All parallel particle filtering algorithms partition the particle population over the processing cores. For the sampling and weight calculation steps this is easy as they work on each particle independently. The global estimate can be computed in two rounds: first locally on each partition and then globally by reducing the local estimates further (sequentially or in parallel). Most of the design choices are in the resampling. Some parallelized filters still resample sequentially or perform exactly the same resampling in parallel requiring communication and possibly load-balancing. Parallel resampling can also operate locally, either with or without exchanging particles. When exchanging particles, various exchange schemes can be envisioned, possibly adapted to the underlying hardware topology. An important question for local resampling is in what cases local resampling is acceptable. We identify two algorithms to perform the actual resampling: Roulette Wheel Selection and Vose's alias method. Many approaches do not name or describe how they perform the actual resampling, but it is often a time consuming operation for large particle populations. Extensive resampling can sometimes lead to a significant loss of variation and thus estimation accuracy. So an issue for any particle filter is whether to always resample or only when needed and how to determine that. A tutorial article on particle filters [7] suggests to evaluate a metric regarding the particle variation and to resample only if it falls below a predetermined threshold. Such a data-dependent conditional operation may also be a sound way to save work, but in its presented form it is undesirable for (resource constrained) real-time systems.

### B. Related Studies

Studies have looked into algorithmic, platform-specific and application-specific improvements to deal with the high computational requirements. The first publication to parallelize particle filtering [9] applied basic parallelization for each step in a master-worker scheme. The master computes the global estimate from the local estimates. Resampling is performed entirely local. Each worker computes the effective sample size and resamples independently if needed. Using two "academic non-linear filtering problems“, measurements are taken with 4K and 16K particles on five different parallel platforms. Speedup is linear up till 16 compute nodes and very platform dependent on 32 nodes. The authors find that local resampling can be as accurate as global resampling. Most subsequent work then reports on various resampling techniques or on filtering performance of new parallel and distributed platforms.

One such work [10] proposes three methods to implement distributed particle filters: (i) Global Distributed Particle Filter (GDPF), (ii) Local Distributed Particle Filter (LDPF), and (iii) Compressed Distributed Particle Filter (CDPF). With GDPF, only the sampling and weight calculation steps run in parallel, while resampling is performed centrally. LDPF is comparable to the basic parallel algorithm [9] where resampling is performed locally without communication. CDPF, similar to GDPF, does resampling centrally, but uses only a small representative set of particles for global resampling. The results are sent back to each individual node. The paper concludes from a number of simulations that LDPF provides both better estimation and performance.

Another study into resampling variants [11] compares two distributed resampling algorithms: (i) Resampling with Proportional Allocation (RPA), and (ii) Resampling with Non-proportional Allocation (RNA). RPA involves a two-stage resampling step (local + global), while RNA involves local resampling followed by a particle exchange step. These algorithms still involve a certain degree of centralized planning and information exchange. RPA provides a better estimation, while RNA has a simpler design. In later work [12], they compare a standard particle filter with a Gaussian particle filter[1] on an FPGA. The presented results indicate that the Gaussian particle filter is equally accurate for (near-)Gaussian problems, but is faster than a standard particle filter. Some of these algorithms (GDPF, RNA, RPA, Gaussian particle filter) are compared using a parallel implementation on a multi-core CPU [13]. The comparison goes until 10K particles. The Gaussian particle filter outperforms all other algorithms for Gaussian estimation problems. RNA achieves near linear speedup with respect to the number of cores, which is much better than the other non-Gaussian filters.

A recent study [14] geared towards distributed tracking by wireless sensor networks proposes a design by adapting the order of the filtering steps to take advantage of near neighbor overhearing. Their evaluation focuses on the lowered communication cost and the effect on tracking accuracy.

With the introduction of GPGPU, new speedup levels have come within reach. Various tracking applications in computer vision [1], [15]–[19] have benefited from using CUDA.

The CUDA-based particle filter implementation from [19] consists of parallel sampling and weight calculations and local resampling. The authors propose to use the finite-redraw importance-maximizing (FRIM) method for the sampling step. The FRIM method rejects drawn particles and redraws until a

[1]Gaussian particle filters are a variant of particle filters that approximate the posterior with a normal distribution and do not require a resampling step.

particle satisfies a minimum weight. This results in a reduction of the required total number of particles. However, the maximum number of redraws is bounded, which is critical for real-time systems. As pseudo-random numbers are generated on the host CPU, the implementation is somewhat limited. The experiments are run on a laptop GPU for up to 4K particles with best execution times around 200 ms.

A more recent study [2] investigates a particle filter for localization and map matching for vehicle applications on a CPU using OpenMP and on a GPU using CUDA. The state dimension is only four and does not benefit from more than 32K particles, but the application is an interesting and well explained case for a particle filter. Experimentation shows that with 128K particles, a CPU is 4.7x faster on six cores and that a GPU is another 16x faster. While they do partition all particles over all cores, resampling is performed on the CPU, but only if particle variance is below a threshold. This means that they need to compute the global variance and perform global resampling. It also means that they can only keep the amount of host-device transfers minimal for rounds where no resampling takes place. When resampling is needed, however, particle weights are transferred to the host and data describing the surviving particles must be transferred back. This strategy is fast only if resampling is not needed very often.

While related studies have compared resampling techniques, or run (parts of) the filter on accelerator platforms, these studies do not scale the number of particles very far. Their dynamic systems do not benefit from more particles, or it has been too computationally intensive to consider. Some designs still contain centralized operations, or data-dependent operations in the context of real-time systems. In the next section, we introduce our design tailored to modern many-core architectures. By carefully designing and optimizing our filter for CUDA/OpenCL devices and by measuring on recent architectures, we can scale at least two orders of magnitude beyond earlier studies to gain insight into how high performance particle filters can best be designed.

## IV. Parallel Particle Filter Design

The particle filter algorithm, as described in Algorithm 1, consists of three steps: (i) Sampling (prediction), (ii) Importance weight calculation (update), and (iii) Resampling. The first two steps apply independent functions to each particle. Delivering a single estimate and resampling require (in principle) coordination between different (groups of) particles.

Our design is based on the fully distributed scheme proposed earlier [20]. The idea was tested with a small experiment (16 groups of 5–80 particles each, connected by a 2D mesh) comparing the estimation accuracy of a distributed computation filter to a centralized filter with 500 particles. In this paper, we work out the idea much further, by using a larger model, by introducing alternative particle exchange schemes, and by exploring the impact of filtering parameters beyond the number of particles on the accuracy/speed trade-off using high-performance algorithms and implementations. The idea is to construct a network of smaller particle filters ("sub-filters"). Each sub-filter computes independently, and only exchanges a few particles each round with its neighbors, which is inherently scalable. Instead of scaling the *size* of each sub-filter, we can scale the *number* of sub-filters in the network, which matches the hardware trend of increasing core counts. Each sub-filter runs the algorithm as described in Algorithm 2. We distribute the particle sub-filters over available CPU cores or GPGPU SMs/CUs. To exchange efficiently, the exchange scheme can be matched to the underlying physical network topology. However, we will later show that All-to-All on globally shared memory is not necessarily a good choice. Figure 1 depicts the exchange schemes considered. We also vary the number of particles exchanged per neighbor pair.

**Algorithm 2** Distributed Particle Filter

```
Input: $X_{k-1}$, $\mathbf{z}_k$, $n\_particles$, $n\_filters$, $n\_transfers$
Output: $X_k$
 1: for $n \leftarrow 1, n\_filters$ do
 2:     $X'_k \leftarrow \emptyset$
 3:     for $i \leftarrow 1, n\_particles$ do
 4:         sample $\mathbf{x}_k^{[n][i]} \sim p(\mathbf{x}_k \mid \mathbf{x}_{k-1}^{[n][i]})$
 5:         $\omega_k^{[n][i]} \leftarrow p(\mathbf{z}_k \mid \mathbf{x}_k^{[n][i]})$
 6:         $X'_k \leftarrow X'_k \cup \{(\mathbf{x}_k^{[n][i]}, \omega_k^{[n][i]})\}$
 7:     end for
 8:     sort $X'_k$ according to weight
 9:     calculate local/global estimate
10:     foreach neighboring filter $q$ do
11:         for $t \leftarrow 1, n\_transfers$ do
12:             send $(\mathbf{x}_k^{[n][t]}, \omega_k^{[n][t]})$ to neighbor $q$
13:         end for
14:     end for
15:     $X_t \leftarrow \emptyset$
16:     for $i \leftarrow 1, n\_particles$ do
17:         draw $\mathbf{x}_k^{[r]}$ with probability $\propto \omega_k^{[r]}$
18:         $X_k \leftarrow X_k \cup \{\mathbf{x}_k^{[r]}\}$
19:     end for
20: end for
```

Another design issue is how to determine the global estimate. We know that the worst local estimate in such a network may still be close to the best local estimate [20]. So depending on the communication costs, it may or may not pay off to take all local estimates into account. What is a good function to select and/or aggregate depends on the application and model; we select the particle with the highest global weight.

The design of the resampling stage is one of the most

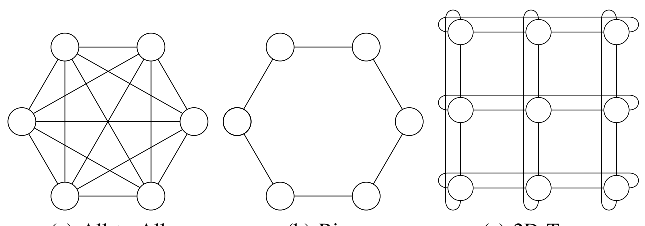


Fig. 1. Considered exchange schemes

important parts of a (distributed) particle filter implementation. As a result of our distributed particle filter design strategy, resampling can be performed locally on small sub-filters as opposed to the global particle population. This circumvents many difficulties arising when resampling a large number of particles for traditional centralized particle filters. There are still a number of choices to be made regarding the resampling algorithm and how often it should be performed.

We have implemented two algorithms to sample (with replacement) from a discrete probability density function: the Roulette Wheel Selection (RWS) algorithm and Vose's alias method [21]. An excellent explanation on both and related algorithms can be found in an article online [22]. Comparing their computational complexities, RWS has $\Theta(n)$ (initialization) and $\Theta(log\ n)$ (generation), while Vose's has $\Theta(n)$ (initialization) and $\Theta(1)$ (generation). Vose's appears favorable, but its initialization every round is harder to parallelize and our sub-filter size is limited.

As for the choice whether to resample or not, we have experimented with the suggested metric to compute the effective sample size as well as a simpler resampling frequency parameter (each sub-filter randomly decides to resample at a fixed ratio of the time). From these experiments we have concluded that although it might be beneficial for low particle settings, frequent resampling generally yields better results. Other applications might still benefit from conditional resampling, but we do not examine this further in our experiments.

While traditional centralized particle filters are characterized only by the total number of particles, the behavior of our distributed particle filter depends on more parameters, listed in Table I.

To evaluate the model with enough particles, deliver a global estimate and maintain good particle variance, we need to select hardware platforms with enough processing capacity.

TABLE I
DISTRIBUTED PARTICLE FILTER PARAMETERS

| **Parameter** | **Symbol** |
|---|---|
| Number of particles per sub-filter | $m$ |
| Number of sub-filters | $N$ |
| Exchange scheme | $X$ |
| Number of particles per exchange | $t$ |

## V. MANY-CORE HARDWARE PLATFORMS

We briefly describe many-core hardware platforms, consisting of GPGPUs and multi-core CPUs.

### *A. General-Purpose GPUs (GPGPUs)*

The CUDA (NVIDIA, Feb 2007) [23] and OpenCL (Khronous Group, Nov 2008) [24] hardware/software platforms present the same virtual platform with a host and device side. Their APIs offer equivalent functionality for NVIDIA GPUs and for many-cores with some minor differences in vendor-specific extensions. The host runs on the CPU to manage device memory, transfer data to/from devices and launch kernels. Each kernel launch specifies the number and size of *independent* work groups, each consisting of 32–1024 threads operating cooperatively in parallel. The device consists of 8–30 "streaming multi-processors" (SM) or "compute units" (CU) to execute kernels and is connected to off-chip, device-wide shared, "global" memory. Each SM/CU contains a data parallelism oriented set of processing units that runs up to several work groups concurrently. It also features a large register file and scratch pad memory, both statically partitioned over work groups at kernel launch time, and caches. A GPGPU SM/CU executes vector instructions in SIMT fashion (single instruction, multiple threads). SIMT supports scatter/gather and a thread runs in each vector lane that can independently diverge on branches (though diverging often costs performance). For GPGPUs, threads in a work group must access global memory with high spatial locality, such that the hardware can combine accesses to deliver high bandwidth. Data in local memory is stored interleaved over multiple banks, so efficient accesses avoid serializing "bank conflicts".

In contrast to CPUs, GPUs are designed to reach high throughput for massively parallel workloads. They feature more computing resources and global memory bandwidth than CPUs, but require much more (esp. spatial) locality of reference and less control flow to maintain pace.

### *B. Multi-Core CPUs*

Multi-core CPUs have 2–8 cores. Some can execute two or four hardware threads per core. An extensive cache hierarchy must keep all cores fed with typically around 2x 32 kB L1 and 256 kB L2 per core and many MBs L3 cache. Multiple channels of main memory are connected directly to each CPU (NUMA). If cores are idle, shared resources are fully used by active cores, and given enough thermal headroom, clock frequencies are increased dynamically.

CPUs can better tolerate unfavorable accesses, bank conflicts and branch divergence. While OpenCL compilers for CPUs can auto-vectorize some constructs, a surer way is to program with CUDA/OpenCL vector types (some GPGPUs may also profit). The maximum power drain of a high-end CPU is around 95 Watt, so one GPU tends to use about the same power as two CPUs.

## VI. IMPLEMENTATION DETAILS

We have implemented a distributed particle filter algorithm using CUDA as well as OpenCL. We have also implemented a sequential, centralized particle filter in C as a reference. All floating point operations in the parallel implementations are in single precision. We compared delivered estimates with those from our double precision reference and found that it does not improve our estimation accuracy by a meaningful amount. Our model as well as the model independent calculations are not so precision sensitive.

The I/O channel between host and device memory is often a bottleneck for GPGPU programs. Unlike many filtering studies, *all* our distributed particle filter operations are executed on the CUDA/OpenCL device. Reducing data transfers to only measurement data and estimates is essential to run experiments with millions of particles.

We have decided to have each GPGPU thread process one particle and each work group one sub-filter. This way, most communication stays within a work group and we can utilize fast, local memory and synchronization to efficiently work cooperatively in parallel. This limits the sub-filter size on GPGPUs to 512 or 1024 particles. This limit could be stretched by processing multiple particles per thread, which may also ease auto-vectorization for CPUs, but increases register pressure and complicates programming.

To maximize attained memory bandwidth on GPGPUs, we need to ensure that as many global memory transfers as possible access memory closely together. If particle data in global memory is more than 16 bytes, transferring in *Structure of Arrays* (SoA) format will not result in efficient transfers, so we store it in the *Array of Structures* (AoS) format. In a few kernels where we cannot have very efficient reads, it can be beneficial to pack individual elements of the particle data into larger, aligned structures. This reduces the number of inefficient memory operations. For more information on optimizing CUDA/OpenCL programs, see the vendor's optimization manuals [25], [26].

Based on the details of Algorithm 2 we implemented the following computational kernels.

1) Pseudo-Random Number Generation
2) Sampling and Importance weight calculation
3) Local Sorting
4) Global Estimate
5) Particle Exchange
6) Resampling

## A. Pseudo-Random Number Generation

Particle filters rely heavily on pseudo-random number generators (PRNGs). Mersenne Twister [27] is a widely used PRNG, characterized by a large period, good test results and an inspiring name. But a PRNG running on many-cores must be able to generate many uncorrelated sequences. To provide this, MTGP [28] was developed as an MT variant optimized for CUDA. We ported MTGP to OpenCL, added a Box-Muller transformation to get a normal distribution, and used it for all our experiments as a separate kernel. MT/MTGP does need substantial computations and state. For GPUs this matters; ideally, MTGP would be directly usable from the sampling and resampling kernels, but then their static resource size increases, allowing fewer concurrent threads per SM.

## B. Sampling and Importance weight calculation

The sampling step involves generating new particles $\mathbf{x}_k^{[m]}$ from the previous particles $\mathbf{x}_{k-1}^{[m]}$. Pseudo-random numbers produced by the previous kernel are used to generate samples from the state transition distribution. The importance weight calculation assigns weights to each particle using measurement data. We can combine sampling and importance weight calculation in one kernel, as measurement data is available at the start of each round.

## C. Local Sorting

Each sub-filter needs to sort its particles according to their weights for the next steps. We use a bitonic sort, which executes a fixed sequence of parallel comparisons and has a complexity of $O(log^2(n))$. In general, the particle data is too large to fit in local memory. Therefore, we sort weights and keep track of the permutation using an index array that we both store locally. To sort the data in global memory, we apply the index array and prefer non-contiguous reads over the more expensive non-contiguous writes. By giving up the flexibility to exchange an arbitrary number of particles, local sorting can in principle be replaced by a cheaper operation such as a local maximum.

## D. Global Estimate

To output a global estimate, we perform a parallel reduction on all particles. The reduction operator can compute the particle with the highest weight, a weighted average, or any other associative operator suitable for the application. We select the particle with the highest weight and since we just sorted locally, we only perform the last reduction round(s).

## E. Particle Exchange

The exchange scheme ($X$) determines which sub-filters exchange particles. It is important to realize that on our platforms, all exchanges go through cached, globally shared memory. We have implemented the All-to-All, Ring, and 2D Torus exchange schemes. For All-to-All, the parameter $t$ indicates the number of particles that each sub-filter supplies. Then, all sub-filters read back the same $t$ "best" particles from the supplied set. The Ring and 2D Torus are more distributed in that exchanged particles are unique to neighbor pairs. In those cases, each sub-filter exchanges $t$ particles with its neighbor.

## F. Resampling

To resample, we implemented the Roulette Wheel Selection (RWS) algorithm and Vose's alias method. Here, we describe our implementations and refer to an article [22] explaining the algorithms. The generation of $m$ samples is independent, but the initializations require more care to parallelize with CUDA/OpenCL. After all surviving particles are known, state vectors are reordered, again preferring non-contiguous reads over non-contiguous writes.

For RWS, initialization can be done using a parallel prefix sum to compute an array of cumulative sums. We use a bank-conflict avoiding implementation [29]. Then to generate, each thread draws one random number, multiplies it with the sum of the local weights, and performs a binary search to find the highest index with a weight not larger than the drawn number.

For Vose's alias method (and in case of weight-dependent conditional resampling), weights must first be normalized. To generate samples, each thread needs *two* uniform random numbers per sample, one to select a particle and the other as a biased coin using the selected particle's weight to select either the particle itself or its "alias". Thus an alias table has to be produced first, which is guaranteed to exist. This is implemented as follows: Particles with a weight below $1/m$ ("small") are segregated from those with a higher weight ("large"). To fill the alias table, weights are "transferred" from a large to a small particle until all weights end up at $1/m$. A particle that receives weight, registers the supplier as its alias and ends up at $1/m$. However, a particle can have at most one alias, so a large particle may become small requiring an alias itself. To save local memory, we operate in-place after filling a single array forwards with small elements and backwards with large elements using atomic operations. One can assign an alias to small elements one at a time, but we operate on $min(\#large, \#small)$ particle pairs at a time. However, concurrency usually drops steeply towards one and some synchronization points are needed. For resampling in our experiments, we only employ Vose's alias method in the sequential, centralized filter, because as we will see, its parallel version is not faster working on sub-filters.

## VII. Experiments

In this section, we explain our robotic arm application and our test setup, followed by experiments to measure filtering frequency and the effects of varying filtering parameters.

### A. *Robotic Arm Application*

To test and benchmark our particle filter, we use the realistic industrial application of a robotic arm. The main reason for this choice is that its measurement equations are highly non-linear and challenging for standard estimation techniques. Another reason is its parametric form. By adjusting the number of joints, we can increase the state dimensions.

In our experiments, the robotic arm has $N$ joints that can be controlled independently. It has one degree of freedom per joint plus the rotation of the base. Each joint has a sensor to measure the angle. The camera at the end is used for tracking an object that is moving on a fixed $x-y$ 2D plane. Figure 2 gives an illustration of this robotic arm.

Let $\theta_{i,k}$ be the angles of the joint $i$ at the discrete time $k$ ($i = 0$ represents the rotational degree of freedom of the base). Let $(x_k, y_k) \in \mathbb{R}^2$ be the position of the object to be tracked at the discrete time step $k$ in the fixed reference system of the robotic arm, as indicated in Figure 2, and let $(v_{x,k}, v_{y,k}) \in \mathbb{R}^2$ be its velocity. Denote by $\mathbf{x}_k = (\theta_0, \ldots, \theta_N, x_k, y_k, v_{x,k}, v_{y,k})^\top$ the state of the robotic arm and object dynamics. We model the angle dynamics as single integrators and the object dynamics as double integrators as:

$$\begin{aligned} \theta_{i,k} &= \theta_{i,k-1} + h_s u_{i,k-1} + w_{\theta_i,k-1} \quad \forall i \in \{0,\ldots,N\} \\ x_k &= x_{k-1} + v_{x,k-1} h_s + w_{x,k-1} \\ y_k &= y_{k-1} + v_{y,k-1} h_s + w_{y,k-1} \\ v_{x,k} &= v_{x,k-1} + w_{v_x,k-1} \\ v_{y,k} &= v_{y,k-1} + w_{v_y,k-1} \end{aligned}$$

where the terms $w$ model the process noise, $u_i$ is the control action applied to the joints, and $h_s$ is the sampling time. This system of dynamical equations represents our a priori $p(\mathbf{x}_k|\mathbf{x}_{k-1})$.

The camera detects the object in its own frame of reference. Let $(x_{C,k}, y_{C,k})$ be the position of the object in the camera moving frame, which can be related back to the state $\mathbf{x}_k$ via rotations and translations. Let $\hat{\theta}_{i,k}$ be the measured value for the angle of each joint and the base. Let $\mathbf{z}_k = (x_{C,k}, y_{C,k}, \hat{\theta}_{0,k}, \ldots, \hat{\theta}_{N,k})^\top$ be the measurement vector. The measurement equations read as:

$$\begin{bmatrix} x_{C,k} \\ y_{C,k} \end{bmatrix} = \mathbf{h}(\mathbf{x}_k) + w_{C,k} \qquad (1)$$

and

$$\hat{\theta}_{i,k} = \theta_{i,k} + w_{\hat{\theta}_i,k}$$

TABLE II
Default filter and model parameters with noise terms

| | |
|---|---|
| Number of particles per sub-filter (GPGPU) | 256 |
| Number of particles per sub-filter (CPU) | 128 |
| Number of sub-filters | 2048 |
| Exchange scheme | Ring |
| Number of particles per exchange | 1 |
| Number of joints | 5 |
| State dimension (#joints + 4) | 9 |
| Arm length (meter) | 0.5 |
| $w_{\theta_i,k}$ | $\mathcal{N}(0, 0.1)$ rad/s |
| $w_{\hat{\theta}_i,k}$ | $\mathcal{N}(0, 0.1)$ rad |
| $w_{C,k}, w_{x,k}, w_{y,k}$ | $\mathcal{N}(0, 0.1)$ m |
| $w_{v_x,k}, w_{v_y,k}$ | $\mathcal{N}(0, 0.1)$ m/s |

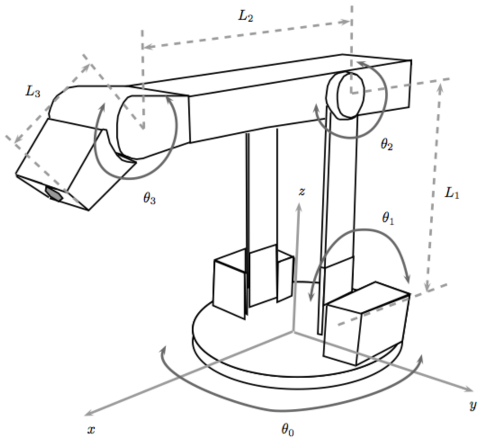


Fig. 2. A 3-joint robotic arm with a camera at the end.

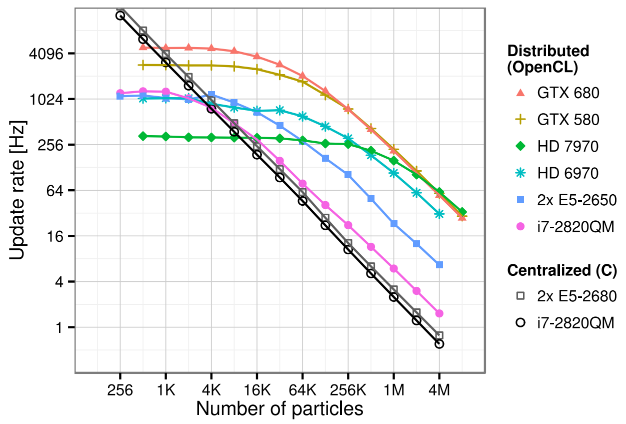


Fig. 3. Achieved particle filter update rate

TABLE III
HARDWARE PLATFORMS

| | Intel Core i7-2820QM | Intel Xeon E5-2680 | NVIDIA GeForce GTX 580 | NVIDIA GeForce GTX 680 | AMD Radeon HD 6970 | AMD Radeon HD 7970 |
|---|---|---|---|---|---|---|
| Platform Type | CPU | dual CPU | GPGPU | GPGPU | GPGPU | GPGPU |
| Nr. Cores/SMs/CUs | 4 | 2x 8 | 16 | 8 | 24 | 8 |
| Core Clock (GHz) | 2.3 | 2.7 | 1.5 | 1.0 | 0.88 | 0.93 |
| Main Mem. (GB) | 8 | 2x 16 | 1.5 | 2 | 2 | 3 |
| On-chip Mem. (kB) | 4x 256, 8192 | 8x 256, 20480 | 16x 64, 768 | 8x 64, 512 | 24x 8, 512 | 8x 16, 768 |
| Comp. SP (GFlop/s) | 74 | 2x 173 | 1581 | 3090 | 2703 | 3789 |
| Memory Bw. (GB/s) | 51 | 2x 51 | 192 | 192 | 176 | 264 |
| TDP (Watt) | 45 | 2x 130 | 244 | 195 | 250 | 230 |
| Released | Jan 2011 | Mar 2012 | Nov 2010 | Mar 2012 | Dec 2010 | Jan 2012 |
| Runtime Software | Intel OpenCL SDK 2012 | | NVIDIA CUDA 4.1 | | AMD APP SDK 2.6 | |

for all $i = 0, \ldots, N$. Here $\mathbf{h}(\mathbf{x}_k)$ represents the highly non-linear rotation-translation function, and the terms $w$ are the measurement noise terms. From these measurement equations we derive our a posteriori $p(\mathbf{z}_k|\mathbf{x}_k)$.

The default filter and model parameters with noise terms (the $w$'s) are listed in Table II. We have adapted the number of particles per sub-filter to the platform, trying to keep them close together without incurring a significant performance penalty. We use these parameters for all tests, unless indicated otherwise.

### *B. Test Setup*

Table III lists the platforms we use. We use a *dual* Xeon CPU (Sandy Bridge-EP) configuration, so we can compare a GPGPU against a CPU platform with similar power usage (TDP). The i7-2820QM is a mobile CPU (Sandy Bridge). Both CPU types can scale their frequency beyond 3 GHz.

All experiments were run under Linux. The centralized implementation was compiled with GCC 4.6.3 -O3 and uses SIMD vectorization for the PRNG and Box-Muller transformation.

### *C. Filtering Runtime Performance*

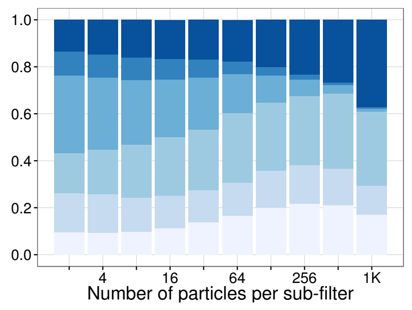
Figure 3 presents the achieved filtering frequency on all platforms. On GPGPUs, we can reach an update frequency of a few hundred Hz with 1 million particles. The dual CPU platform is up to 6.5x faster than running a centralized filter sequentially, but up to 14x slower than a high end GPGPU. The Radeon HD GPGPUs stay behind even more for very small filters, but outperform CPUs for medium filters, while the HD 7970 outperforms any other at 4M and 8M particles. (We cannot make larger GPU memory allocations to see its trend beyond.) We also have timed our OpenCL code on the GTX 580; it is at most 5% slower than with CUDA.

We now explore the relative performance impact of our filter kernels when scaling (i) the number of particles per sub-filter, (ii) the number of sub-filters, and (iii) the state dimensions. For each experiment, all other parameters stay the same. (The total number of particles does increase for the first two experiments). The plotted breakdowns in Figure 4 have been run on a GTX 580 running CUDA. The trends from other (esp. NVIDIA) GPGPUs are similar. The biggest difference between our dual CPU and GPGPU performance is that the CPU spends much more time on random numbers (40% at 64 particles/sub-filter). This can be attributed to the PRNG kernel used; MTGP is highly optimized for (NVIDIA) GPGPUs and apparently performs poorly on CPUs. Checking this out, we found that our OpenCL MTGP port runs about 50% slower on the dual E5-2680 than "SFMT", the optimized single core CPU implementation of MT.

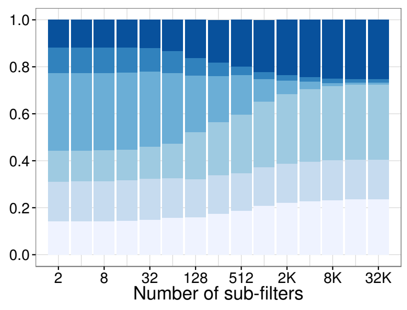
*1) Scaling the Number of Particles per Sub-Filter:* From Figure 4a, we see that when the number of particles increases, the compute-heavy sorting and resampling stages dominate the runtime at the cost of non-local stages. This is good news, because it means that the filter can be tuned to the computation to communication ratio using this parameter. Our experiments on the CPU platform confirm this with non-local stages being much cheaper.

*2) Scaling the Number of Sub-Filters:* From Figure 4b, we see that when the number of sub-filters increases towards 32K (16M particles), local operations dominate, but unlike the last experiment, changes appear to be settling down when reaching 32K. Local sort takes most of the time, so we would do well to take a second look at that in future optimizations. When the kernels that have enough computation to overlap long stalls dominate, execution time rises linearly with more sub-filters. With a small setup, the HD 7970 spends more time on the global estimate in favor of sampling, while for a large setup, resampling is fractionally cheaper at the cost of the RNG. The CPU trades in relative time in non-local operations for more time to sort, but the other kernels (except RNG) stay the same with resampling at about 18%.

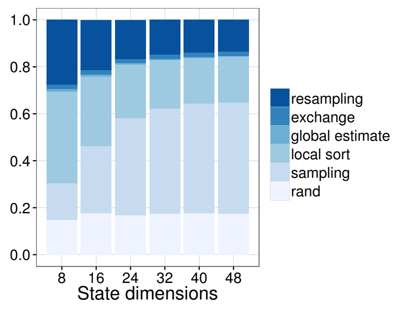
*3) Scaling the State Dimensions:* In this experiment, we scale a model-specific aspect of filtering by testing state dimensions from 8–48 (4–44 joints) variables of 4 bytes. From Figure 4c, we see that when the state dimensions increase to 48, the sampling (with weight calculation) fraction increases to 45% of the runtime at the cost of local sorting and resampling. As the problem becomes more complex, the "filtering" aspect loses relevance and the model implementation more and more determines the runtime. Depending on the model, sampling may be easier to implement efficiently, and certainly to parallelize.



(a) Number of particles per sub-filter



(b) Number of sub-filters



(c) State dimensions

Fig. 4. Performance breakdown when scaling various parameters

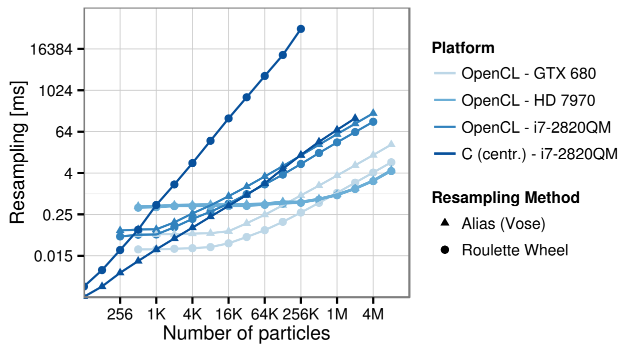


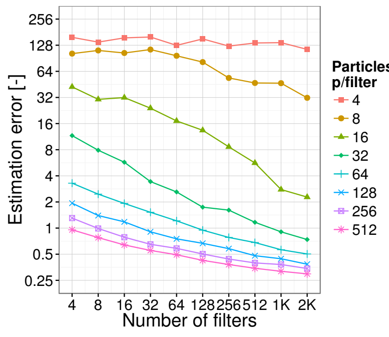
Fig. 5. Runtime of Roulette Wheel Selection vs. Vose's alias method

*4) Resampling Algorithms:* In Figure 5, we compare the runtime of the resampling kernel running the Roulette Wheel Selection algorithm and Vose's alias method. For a sequential, centralized filter, Vose's is much faster as expected. (Vose's need for more random numbers is not included, but in no way changes this.) But for all platforms running OpenCL code, resampling with Vose's is never faster. A sub-filter size of 256 is too small to earn back the overhead from synchronization and reduced concurrency during table initialization. We have some ideas how to improve concurrency, but only using more atomic operations on local memory. We do see that on the HD 7970 GPGPU both algorithms are equally fast (plot lines are superimposed).

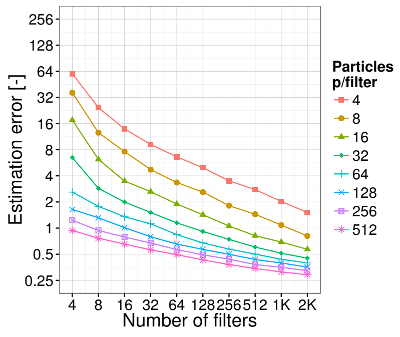
## D. Estimation Accuracy

After runtime performance, we inspect the impact to accuracy from exchange scheme and number of exchanged particles. We scale the number of sub-filters, since that is our principal scaling direction. The presented estimation errors in Figure 6 are averages from 100 runs over 100 time steps for each configuration.

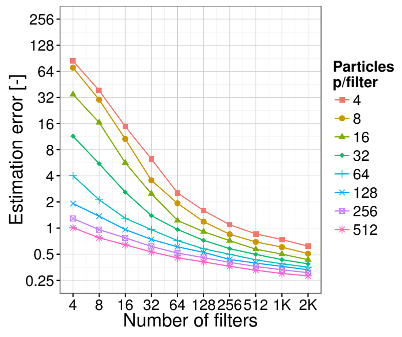
*1) Filter Network Size and Exchange Scheme:* From Figure 6, we can clearly see that All-to-All delivers the worst



(a) All-to-All

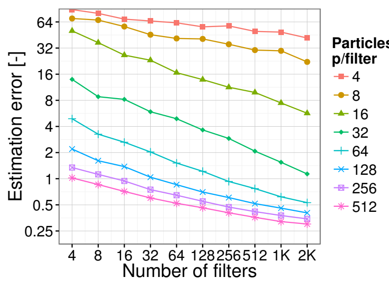


(b) Ring

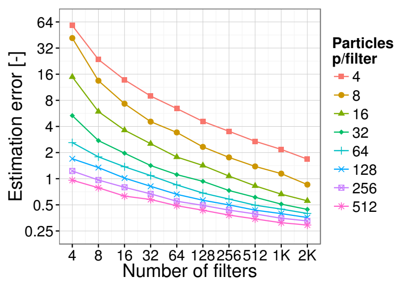


(c) 2D Torus

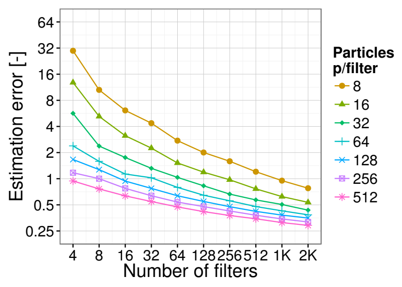
Fig. 6. Estimation error with varying particle exchange schemes



(a) $t = 0$



(b) $t = 1$



(c) $t = 2$

Fig. 7. Estimation error with varying number of exchanged particles

estimates. This happens as a consequence of a loss of diversity among the whole particle population as the same particles are fed into all sub-filters. The most interesting observation from both the ring and 2D Torus is that in all cases, a low number of particles can be compensated by adding more sub-filters. This confirms our strategy of dividing a large particle filter into a network of smaller sub-filters. We also observe that with a low number of sub-filters, the Ring outperforms the 2D Torus, while with a large number of sub-filters, the 2D Torus outperforms the Ring. The extra connectivity from the 2D Torus results in a loss of diversity in small networks, while it enables faster propagation of more likely particles in large networks.

*2) Particle Exchange:* From Figure 7, we see that the benefit of particle exchange is evident. Exchanging more than one particle offers a minor improvement, but exchanging a single particle seems sufficient for the likely particles to spread. We ran up to $t = 16$ to verify the trend.

### *E. Discussion*

Using our robotic arm application, we attained high filtering rates with large particle sets using GPGPUs. A comparable CPU platform performs slower, but we believe the gap can be reduced by using CPU specialized kernels.

We analyzed the scaling behavior in three directions and found that for larger particle populations, local operations increasingly take more runtime. However, a large particle population is only needed for complex models with large state dimensions, where ultimately, model-specific sampling and weight calculations become the dominant factor in the total runtime. As for particle exchanges, even minimal communication among the individual filters is sufficient to spread likely particles throughout the network. An article [30] from the perspective of control systems provides additional experimental (and theoretical) results using an actual robotic arm in a closed-loop setting.

A single, optimal configuration does not exist. The rules of thumb we deduce are that in small filtering setups, limited communication and a low connectivity network provide the best results. High particle settings tend to perform better with a more connected network and increased communication. Both resampling algorithms we tried have their uses, but when the model state is large, this becomes less relevant.

## VIII. Evaluation

In this section, we explain how we validated that our particle filter produces proper estimates. We also assess if a distributed scheme requires the same number of particles as a centralized scheme to deliver the same estimation accuracy.

### *A. Correctness Validation*

We used two techniques to validate our particle filter implementations: (i) using a ground truth, and (ii) using reference implementations.

Initially, without a reference implementation for our model, a particle filter can be checked to see if it converges to a known correct state under various noise levels and filter configurations. In Figure 8, the dark line shows our ground truth as a lemniscate path that starts by heading up from the right side, as would be observed from the camera on our robotic arm. Two particle filters start in the center on the right *off* the ground truth. The high particle estimation uses $512 \times 512$ particles and converges. But the low particle estimation uses only $16 \times 16$ particles, which is not enough.

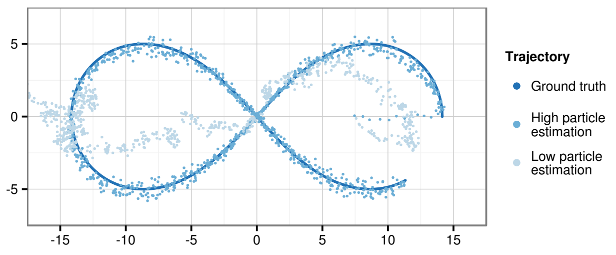


Fig. 8. Lemniscate ground truth with two filter traces

We also developed a centralized and a distributed particle filter as sequential reference implementations. These are much easier to implement as intended than optimized CUDA/OpenCL versions. Once tested against ground truths, output from optimized versions can be compared to reference output in more detail.

### *B. Distributed Filtering Overhead*

To quantify the overhead of our distributed scheme, we compare the estimation error of our distributed filter with that of a centralized filter having the same total number of particles. To this end, we developed our sequential, centralized filter in C as a fast reference for estimation accuracy. As the sub-filter size significantly alters the distributed filter behavior, we made multiple comparisons using different sub-filter sizes. The results are depicted in Figure 9.

The presented results confirm that, although many distributed particle filter configurations perform poorly, for all filter sizes, distributed configurations exist which perform similarly to (or even outperform) their centralized counterparts. For small filtering setups of hundreds of particles, the default sub-filter sizes of 128 and 256 are large enough; we do not measure a negative impact to estimation accuracy (which we

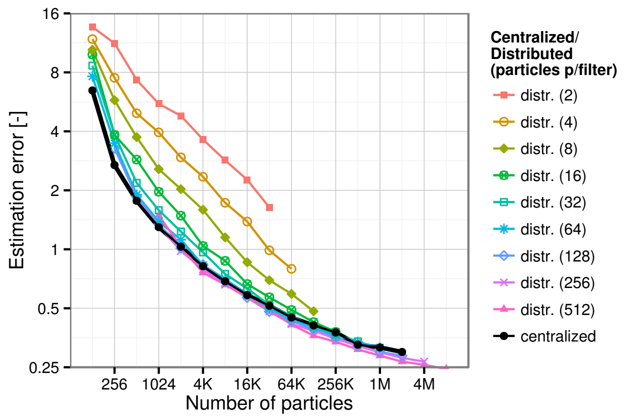


Fig. 9. Estimation error of distributed vs. centralized filtering

would then have to compensate for with extra particles for a fair performance comparison). Only if we use very small filtering or sub-filter sizes (the filter may not converge), accuracy is impacted, possibly severely. Although in general a distributed setup is not "for free", our measurements show that the effect is manageable, even with a few million particles. Given a proper filtering configuration with exchanges, our network of sub-filters can replace a centralized particle filter without requiring additional particles that decrease performance.

## IX. Conclusion and Future Work

The particle filter is a powerful, but computationally demanding Monte Carlo based Bayesian estimation method. With the introduction of massively parallel many-core processors, particle filtering has become viable, but the design intricacies of the algorithm remained elusive.

For this paper, we thoroughly investigated a new particle filter algorithm and implemented a realistic robotic application on various hardware platforms using CUDA and OpenCL. We identified the parameters that strongly affect the behavior of our filter and experimentally quantified their scaling effects on estimation accuracy and execution speed. For the particle exchange and resampling functions, we implemented several alternatives, and we demonstrated that there is no single configuration that is optimal for all settings.

Our experiments indicate that real-time particle filtering for complex estimation problems is feasible on current generation GPGPUs and, for smaller systems, on multi-core CPUs. For small estimation problems with up to four state variables, we can reach kHz estimation rates. For our robotic arm application with nine state variables utilizing over one million particles, we pushed our GPGPUs to attain estimation rates of 100–200 Hz. Given many-core processor trends, it is important to use a design that effectively combines more (and not larger) sub-filters. The All-to-All particle exchange scheme may seem an intuitive choice for a shared memory system, but it decreases particle diversity too much. We discovered that exchange schemes with a lower connectivity are preferable *for accuracy reasons*. The ring performs better with a small number of sub-filters, while the added connectivity of the 2D Torus compares favorably with a large number of sub-filters. Another rule of thumb is that accuracy can improve a lot by exchanging even one particle per pair of neighboring sub-filters.

As for multi-core CPUs with similar power usage (dual), CUDA/OpenCL particle filter performance is up to a factor 14 slower than a high-end GPGPU. We believe that this gap can be narrowed, but that requires writing platform optimized kernels, such as with the Mersenne Twister variants that we ran. Note that the filtering performance of a particular estimation problem also depends on model-specific characteristics.

We see two interesting directions for future work. The first direction is a matter of scale: down to real-time applications on embedded systems (with GPGPU cores), or up to take advantage of clusters. Each platform scale direction will present new challenges to performance portability.

Another direction is to focus on applications with different types of estimation problems. We expect to gain an even better understanding of the particle filter design with data available from different estimation problems.

## Acknowledgment

This research was partially funded by NWO/STARE under project AstroStream. The authors would like to thank Intel Corporation for providing a dual Xeon E5-2680.

## References

[1] O. M. Lozano and K. Otsuka, "Real-time visual tracker by stream processing," *Journal of Signal Processing Systems*, vol. 57, no. 2, pp. 285–295, 2009.

[2] K. Par and O. Tosun, "Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures," in *Proc. of the IEEE 2011 Intelligent Vehicles Symposium.* IEEE, June 2011, pp. 820–826.

[3] T. Flury and N. Shephard, "Bayesian inference based only on simulated likelihood: particle filter analysis of dynamic economic models," *Econometric Theory*, vol. 27, no. 05, pp. 933–956, May 2011.

[4] F. Cérou, P. D. Moral, T. Furon, and A. Guyader, "Sequential monte carlo for rare event estimation," *Statistics and Computing*, vol. 22, no. 3, pp. 795–808, May 2012.

[5] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, ser. Intelligent robotics and autonomous agents. The MIT Press, Sep 2005. [Online]. Available: http://www.worldcat.org/isbn/0262201623

[6] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *Radar and Signal Processing*, vol. 140, no. 2, pp. 107–113, Apr 1993.

[7] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian tracking," *IEEE Trans. on Signal Processing*, vol. 50, no. 2, pp. 174–188, Feb 2002. [Online]. Available: http://dx.doi.org/10.1109/78.978374

[8] A. Doucet, S. Godsill, and C. Andrieu, "On sequential Monte Carlo sampling methods for Bayesian filtering," *Statistics and Computing*, vol. 10, pp. 197–208, Jul. 2000. [Online]. Available: http://portal.acm.org/citation.cfm?id=599374

[9] O. Brun, V. Teuliere, and J. M. Garcia, "Parallel Particle Filtering," *Journal of Parallel and Distributed Computing*, vol. 62, no. 7, pp. 1186–1202, July 2002. [Online]. Available: http://dx.doi.org/10.1006/jpdc.2002.1843

[10] A. S. Bashi, V. P. Jilkov, X. R. Li, and H. Chen, "Distributed Implementations of Particle Filters," in *Proc. of the 6th Int'l Conf. of Information Fusion*, Jul. 2003, pp. 1164–1171.

[11] M. Bolić, P. M. Djurić, and S. Hong, "Resampling Algorithms and Architectures for Distributed Particle Filters," *IEEE Trans. on Signal Processing*, vol. 53, no. 7, pp. 2442–2450, Jul 2005. [Online]. Available: http://dx.doi.org/10.1109/TSP.2005.849185

[12] M. Bolić, A. Athalye, S. Hong, and P. Djurić, "Study of Algorithmic and Architectural Characteristics of Gaussian Particle Filters," *Journal of Signal Processing Systems*, vol. 61, no. 2, pp. 205–218, Nov 2010. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0434-4

[13] O. Rosén, A. Medvedev, and M. Ekman, "Speedup and Tracking Accuracy Evaluation of Parallel Particle Filter Algorithms Implemented on a Multicore Architecture," in *2010 IEEE Int'l Conf. on Control Applications*, Sep 2010, pp. 440–445. [Online]. Available: http://dx.doi.org/10.1109/CCA.2010.5611217

[14] B. Jiang and B. Ravindran, "Completely distributed particle filters for target tracking in sensor networks," in *Proc. of the 2011 IEEE International Parallel & Distributed Processing Symposium.* IEEE, May 2011, pp. 334–344.

[15] K. Otsuka and J. Yamato, "Fast and robust face tracking for analyzing multiparty face-to-face meetings," in *Proc. of the 5th Int'l Workshop on Machine Learning for Multimodal Interaction*, vol. LNCS 5237. Springer, Sep 2008, pp. 14–25.

[16] R. M. Friborg, S. Hauberg, and K. Erleben, "GPU accelerated likelihoods for stereo-based articulated tracking," in *the ECCV Workshop on Computer Vision on GPUs*, Sep 2010.

[17] J. F. Ferreira, J. Lobo, , and J. Dias, "Bayesian real-time perception algorithms on GPU," *Journal of Real-Time Image Processing (Special Issue)*, vol. 6, no. 3, pp. 171–186, 2010.

[18] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron, "Parallelization of particle filter algorithms," in *Proc. of 3rd Workshop on Emerging Applications and Many-core Architecture*, June 2010.

[19] M.-A. Chao, C.-Y. Chu, C.-H. Chao, and A.-Y. Wu, "Efficient parallelized particle filter design on CUDA," in *Proc. of the 2010 IEEE Workshop on Signal Processing Systems*, San Francisco, USA, Oct 2010, pp. 299–304.

[20] A. Simonetto and T. Keviczky, "Recent Developments in Distributed Particle Filtering: Towards Fast and Accurate Algorithms," in *1st IFAC Workshop on Estimation and Control of Networked Systems*, Sep 2009. [Online]. Available: http://www.ifac-papersonline.net/Detailed/40541.html

[21] M. D. Vose, "A linear algorithm for generating random numbers with a given distribution," *IEEE Trans. on Software Engineering*, vol. 17, no. 9, pp. 972–975, Sep 1991. [Online]. Available: http://web.eecs.utk.edu/~vose/Publications/random.pdf

[22] K. Schwarz, "Darts, dice, and coins: Sampling from a discrete distribution," Dec 2011. [Online]. Available: http://www.keithschwarz.com/darts-dice-coins/

[23] NVIDIA, *NVIDIA CUDA Programming Guide*, NVIDIA Corp., Santa Clara, CA, USA, Nov 2011.

[24] Khronos OpenCL Working Group, *The OpenCL Specification*, Nov 2011.

[25] NVIDIA, *CUDA C Best Practices Guide*, NVIDIA Corp., Santa Clara, CA, USA, Jan 2012.

[26] Intel, *OpenCL* Optimization Guide*, Intel Corp., Apr 2012.

[27] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Trans. on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, Jan 1998. [Online]. Available: http://dx.doi.org/10.1145/272991.272995

[28] M. Saito, "A Variant of Mersenne Twister Suitable for Graphic Processors," Jun. 2010. [Online]. Available: http://arxiv.org/abs/1005.4973

[29] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, Aug 2007, ch. 39. [Online]. Available: http://www.worldcat.org/isbn/0321515269

[30] M. Chitchian, A. Simonetto, A. S. van Amesfoort, and T. Keviczky, "Distributed computation particle filters on gpu-architectures for real-time control applications," *IEEE Transactions on Control Systems Technology*, vol. 21, 2013, to appear.