Fall 12-18-2014

Parallel Computing of Particle Filtering Algorithms for Target Tracking Applications

Jiande Wu
University of New Orleans, jwu3@uno.edu

Parallel Computing of Particle Filtering Algorithms for Target Tracking Applications

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fullfllment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science

by

Jiande Wu

B.S. North China Electric Power University, 1998
M.S. North China Electric Power University, 2005
M.S. University of New Orleans, 2012

December 2014

Acknowledgements

I would like to express my sincere thanks to lots of people without whom this dissertation would not have been possible.

To my advisor, Dr. Vesselin Jilkov, who is always a source of knowledge and inspiration for me.

To Dr. X. Rong Li and Dr. Huimin Chen who also provide me invaluable and insightful thinking and guidance during my research.

To my committee members, Dr. Shengru Tu and Dr. George Ioup, for their great effort and help on my dissertation.

To my colleagues and friends, Dr. Zhansheng Duan, Dr. Yu Liu, Dr. Xiaomeng Bian, Dr. Meiqin Liu, Dr. Ji Zhang, Dr. Yongxin Gao, Ms. Jing Hou, Mr. Haozhan Meng and Mr. Gang Liu, without whom, my life during the PhD study in UNO would become much harder. Without these people, I can hardly make this journey. I dedicate this dissertation to them. Furthermore, I want to thank all the members and staff of the Department of Electrical Engineering at the University of New Orleans for their service and support.

Last but not least at all, to my parents, my wife, my sister and my children, who continuously support and encourage me throughout my study and research.

Further, my research was supported in part by Louisiana BoR through grant LEQSF(2009-12)-RD-A-25, ARO through grant W911NF-08-1-0409, ONR-DEPSCoR through grant N00014-09-1-1169, and NASA/LEQSF (2013-15) through grant NNX13A29A.

Contents

Abstract

Particle filtering has been a very popular method to solve nonlinear/non-Gaussian state estimation problems for more than twenty years. Particle filters (PFs) have found lots of applications in areas that include nonlinear filtering of noisy signals and data, especially in target tracking. However, implementation of high dimensional PFs in real-time for large-scale problems is a very challenging computational task.

Parallel & distributed (P&D) computing is a promising way to deal with the computational challenges of PF methods. The main goal of this dissertation is to develop, implement and evaluate computationally efficient PF algorithms for target tracking, and thereby bring them closer to practical applications. To reach this goal, a number of parallel PF algorithms is designed and implemented using different parallel hardware architectures such as Computer Cluster, Graphics Processing Unit (GPU), and Field-Programmable Gate Array (FPGA). Proposed is an improved PF implementation for computer cluster - the Particle Transfer Algorithm (PTA), which takes advantage of the cluster architecture and outperforms significantly existing algorithms. Also, a novel GPU PF algorithm implementation is designed which is highly efficient for GPU architectures. The proposed algorithm implementations on different parallel computing environments are applied and tested for target tracking problems, such as space object tracking, ground multitarget tracking using image sensor, UAV-multisensor tracking. Comprehensive performance evaluation and comparison of the algorithms for both tracking and computational capabilities is performed. It is demonstrated by the obtained simulation results that the proposed implementations help greatly overcome the computational issues of particle filtering for realistic practical problems.

Keywords: nonlinear filtering, particle filter, particle flow filter, parallel and distributed computing, GPU, computer cluster, FPGA, target tracking

List of Figures

List of Tables

Chapter 1

Introduction

Particle filtering (PF) was originally proposed in [35]. Due to its generality and simplicity, PF has become a topic of constantly growing interest, development and numerous target tracking applications, e.g. [75]. Particle filters are a group of posterior density estimation algorithms that estimate the posterior density of the state-space by directly implementing the Bayesian filtering approach. This is a recursive algorithm. It consists of two parts: prediction and update. If the variables are linear and normally distributed the Bayesian filter becomes equal to the Kalman filter [78]. The Bayesian filter is also used in computer science for calculating the probabilities of multiple beliefs.

Formally, let $\{x_k\}_{k=1,2,\dots}$ be a vector valued discrete-time Markov process with state transition *probability density function* (PDF) $p(x_k|x_{k-1})$, and let $\{z_k\}_{k=1,2,\dots}$ be another process, stochastically related to $\{x_k\}_{k=1,2,\dots}$ through the *likelihood* $p(z_k|x_k)$. x_k and z_k are the state and the measurement, respectively, and $p(x_k|x_{k-1})$ and $p(z_k|x_k)$ represent the state and measurement models. The exact *Bayesian recursive filter* (BRF) provides the posterior density

$p(x_k|z^k)$ via the following prediction – update scheme [58]:

$$p(x_{k-1}|z^{k-1}) \longrightarrow p(x_k|z^{k-1}) \longrightarrow p(x_k|z^k) \tag{1.1}$$

given the initial state PDF $p(x_0)$ and a sequence of measurements $z^k = \{z_1, ..., z_k\}$.

PF methods use a set of particles to represent the posterior density. These filtering methods make no restrictive assumption about the dynamics of the state-space or the density function. PF methods provide a well-established methodology for generating samples from the required distribution. The state-space model can be non-linear and the initial state and noise distributions can take any form.

However, PFs are not practical yet, due to their excessive computational and memory cost. A natural approach to overcome this prohibitive limitation is to employ *parallel & distributed* (P&D) computing. Development of P&D algorithms and architectures that utilize the spatial and temporal concurrency of PF computations is a possibility to make the PF-based multiple target tracking, and particle filtering in general, practical. This possibility has not been much studied so far, even though useful work has been done in this direction, e.g., [6, 84, 9, 8, 10, 56]. The most critical and computationally expensive in every implementation of particle filtering is the resampling. Resampling is in effect discarding of samples that have small probability and concentrating on samples that are more probable. In parallel implementations, resampling becomes a bottleneck due to its inherently sequential nature.

Another kind of PFs has been proposed recently - the particle flow filters (PFFs). They can solve the well known problem of particle degeneracy of the PFs. A flow of each particle from prior to posterior is governed by a flow equation which in general satisfies a linear partial differential equation (PDE) with constant coefficients. This PDE is central in this approach. Its numerical integration, however, is very challenging for real-time computation.

This dissertation addresses the computational challenges of PF and PFF. The main goal is to propose, develop and implement computationally efficient algorithms for particle and particle flow filters for target tracking, and thereby bring them closer to practical application. Parallel & distributed [3] computing is a natural way to overcome this challenge. Thus, to reach this goal, a number of parallel particle and particle flow filter algorithms are designed and implemented using different parallel hardware architectures, such as computer cluster, graphics processing unit (GPU), and field-programmable gate array (FPGA).

This dissertation is organized in six chapters. In these chapters, the motivation, contributions and results are described. Throughout the dissertation, performances of particle and particle flow filters are estimated and filter implementation prototypes are built for target tracking applications.

Chapter 1 introduces the background, motivation and goal of this research work.

Chapter 2 briefly surveys the related work, including nonlinear density estimation, and characterization of parallel algorithms and parallel architectures.

Chapter 3 proposes an improved parallel PF algorithm implementation that is more suitable for computer cluster architecture. Cluster implementation of target tracking problems for space object tracking, and for ground multitarget tracking using image sensor is conducted and the performance evaluated. Our algorithm has significant advantage in terms of computational performances as compared to other parallel PF algorithms. On the other hand, its tracking performance degradation is not that significant.

Chapter 4 proposes an improved PF resampling algorithm for GPU implementation. The new algorithm reduces the complexity of realization through addressing common issues such as decreasing the latency, memory size and memory access. It is tested using ground multitarget tracking, UAV-multisensor tracking, and a highly dimensional nonlinear density problem. Our proposed method is significantly superior to the generic filters implementations in both estimation accuracy and computational efficiency. The chapter also develops a GPU implementation of the exact PFF and shows that it has a superior computational performance as compared to the PF implementations.

Chapter 5 proposes a particle flow filter implementation on FPGA. A simple non-linear system, which is typically studied in the context of stochastic system is used as a simulation model. This chapter also analyzes PF & PFF algorithms performance in FPGA architecture in terms of computational complexity and potential throughput.

Chapter 6 draws conclusions and discusses future work.

Chapter 2

Background of Related Work

2.1 Particle Filtering

2.1.1 Models

Let $\{x_k\}_{k=1,2,\dots}$ be a vector valued discrete-time Markov process with state transition probability density function (PDF) $p(x_k|x_{k-1})$ as given by (2.1a), and let $\{z_k\}_{k=1,2,\dots}$ be another process, stochastically related to $\{x_k\}_{k=1,2,\dots}$ through the likelihood $p(z_k|x_k)$ as given by (2.1b). x_k and z_k are the state and the measurement, respectively, and $p(x_k|x_{k-1})$ and $p(z_k|x_k)$ are the state and measurement models in terms of probability distributions, i.e.,

$$x_k \sim p(x_x|x_{k-1}) \tag{2.1a}$$

$$z_k \sim p(z_k|x_k) \tag{2.1b}$$

The initial state distribution is $p(x_0)$, i.e., $x_0 \sim p(x_0)$.

Most often the state and measurement models are given in state-space representation

$$x_{k+1} = f(x_k, w_k) \tag{2.2a}$$

$$z_k = h(x_k, v_k) \tag{2.2b}$$

where f and h are the state and measurement function, and w_k and v_k denote the process and measurement noises, respectively. The relationship between (2.1) and (2.2) can be find in [58].

2.1.2 Bayesian Recursive Filter

The exact Bayesian recursive filter (BRF) provides the posterior density $p(x_k|z^k)$, given $p(x_0)$ and measurements $z^k = \{z_1, ..., z_k\}$ through the following recursion [58]:

$$\begin{tikzcd}
 & p(x_k|x_{k-1}) \arrow[d] & & z_k \to p(z_k|x_k) \arrow[d] & \\
p(x_{k-1}|z^{k-1}) & \longrightarrow & p(x_k|z^{k-1}) & \longrightarrow & p(x_k|z^k) \\
 & \text{(Prediction)} & & \text{(Update)} &
\end{tikzcd}$$

The prediction is given by

$$p(x_k|z^{k-1}) = \int_{\mathbb{R}^{n_x}} p(x_k|x_{k-1})p(x_{k-1}|z^{k-1})dx_{k-1}. \tag{2.3}$$

The update is given by

$$p(x_k|z^k) = \frac{p(z_k|x_k)p(x_k|z^{k-1})}{p(z_k|z^{k-1})}, \tag{2.4}$$

where $p(z_k|z^{k-1}) = \int_{\mathbb{R}^{n_x}} p(z_k|x_k)p(x_k|z^{k-1})dx_k$ is the normalization constant.

For output from a nonlinear BRF, usually the minimum mean square (MMS) estimate

$\hat{x}_k^{\text{MMS}}$ and its covariance $P_{k|k}^{\text{MMS}}$ are computed:

$$\hat{x}_{k|k}^{\text{MMS}} = \int x_k p(x_k|z^k) dx_k \tag{2.5a}$$

$$P_{k|k}^{\text{MMS}} = \int (x_k - \hat{x}_k^{\text{MMS}})(x_k - \hat{x}_k^{\text{MMS}})^T p(x_k|z^k) dx_k. \tag{2.5b}$$

2.1.3 Basic Particle Filter (PF)

PF methods use a set of particles to represent the posterior density. These filtering methods use an assumption about the dynamics of the state-space or the density function models (2.2) or (2.1), respectively. PF methods provide a well-established methodology for generating samples from the required posterior distribution. The state-space model can be non-linear and the initial state and noise distributions can take any form.

Generic SIS/R PF Algorithm

In particle filtering the probability density functions (PDFs) are represented approximately through a set of random samples (particles) and the BRF (1.1) is performed directly on these samples. Most PFs are based on two principal components [2]: sequential importance sampling (SIS) and resampling (R) as given in Table 2.1.

The importance distribution $\pi(\cdot)$ must contain the support of the posterior and is to be designed. One possibility is to choose $\pi = p(\ x_k|\ x_{k-1}^i)$ [35] which is often referred to as the sampling importance resampling (SIR) PF, or the bootstrap PF. Many other choices are possible [30]. Resampling is in effect discarding of samples that have small probability and concentrating on samples that are more probable. The resampling step is critical in every implementation of PF because without it the variance of the particles weights quickly increases leading to inference degradation.

Table 2.1: Generic SIS/R PF Algorithm [2]

```
• Importance Sampling (IS)
  – For $i = 1, \ldots, \bar{N}$
    Draw a sample (particle):    $\bar{x}_k^i \sim \pi\left(x_k | x_{k-1}^i, z^k\right)$
    Evaluate importance weights
      $\bar{w}_k^i = w_{k-1}^i \frac{p\left(z_k|x_k^i\right)p\left(x_k^i|x_{k-1}^i\right)}{\pi\left(x_k^i|x_{k-1}^i, z^k\right)}$
  – For $i = 1, \ldots, \bar{N}$
    Normalize importance weights: $w_k^i = \frac{\bar{w}_k^i}{\sum_{j=1}^{\bar{N}} \bar{w}_k^j}$
• Resampling (R)
  – Effective sample size estimation: $\hat{N}_{eff} = \frac{1}{\sum_{j=1}^{N}\left(w_k^j\right)^2}$
  – If $\hat{N}_{eff} < N_{th}$
    Sample from $\left\{\bar{x}_k^j, w_k^j\right\}_{j=1}^{N}$ to obtain
    a new sample set $\left\{x_k^i = \bar{x}_k^{j_i}, w_k^i = \frac{1}{N}\right\}_{i=1}^{N}$
```

2.1.4 Resampling Algorithms Review

The resampling step modifies the weighted approximate density to an unweighted density by deleting particles having low importance weights and by copying particles having high importance weights. In the particle filter literature [5, 34, 37, 40, 41, 54, 57, 64, 69, 70, 81, 87] four basic resampling algorithms can be identified [29], as given next.

Multinomial resampling

This approach to resampling is based on an idea of the bootstrap method [31]. It is known as multinomial resampling since the duplication counts $N_1, \ldots, N_m$ are by definition distributed according to the multinomial distribution $\mathrm{Mult}(n; w_1, \ldots, w_m)$.

Sampling each $x^{(k)}$ independently from the distribution associated with $\{\tilde{x}^{(i)}, \tilde{w}^{(i)}\}_{i=1}^{N}$ is equivalent to sampling numbers of replicates from a multinomial distribution: $M \sim$

$\mathcal{M}(\cdot; N, \{\tilde{w}^{(i)}\}_{i=1}^{N})$, which has the probability mass function:

$$\mathcal{M}(n_1, \ldots, n_N; N, \{\tilde{w}^{(i)}\}_{i=1}^{N}) = \begin{cases} \frac{N!}{\prod_{i=1}^{N} n_i!} \prod_{i=1}^{N} (\tilde{w}^{(i)})^{n_i} & \text{if } \sum_{i=1}^{N} n_i = N, \\ 0 & \text{otherwise.} \end{cases} \tag{2.6}$$

Stratified Resampling

Stratified resampling, [48] and [32, Section 5.3], is based on ideas used in survey sampling and consists in pre-partitioning the $(0, 1]$ interval into n disjoint sets,

$$(0, 1] = (0, 1/n] \cup \cdots \cup (\{n-1\}/n, 1]$$

Generate N ordered random numbers $\tilde{u}_k \sim \mathcal{U}[0, 1)$, then

$$u_k = \frac{(k-1) + \tilde{u}_k}{N} = \frac{k-1}{N} + \frac{1}{N}\tilde{u}_k \tag{2.7}$$

Systematic Resampling

It was introduced by [11]. Generate one random number $\tilde{u} \sim \mathcal{U}[0, 1)$, then

$$u_k = \frac{(k-1) + \tilde{u}}{N} = \frac{k-1}{N} + \frac{1}{N}\tilde{u} \tag{2.8}$$

and use u_k to select $x^{(k)}$ according to multinomial distribution.

Residual Resampling

Residual resampling is introduced by [85], [60]. The idea is to allocate $n_i = \lfloor N\tilde{w}^{(i)} \rfloor$ copies of particle $\tilde{x}^{(i)}$ to the new distribution. Then, resampling $m = N - \sum n_i$ particles from $\{\tilde{x}^{(i)}\}$ by selecting $\tilde{x}^{(i)}$ is proportional to $w'^{(i)} = N\tilde{w}^{(i)} - n_i$ using one of the resampling methods described before.

2.1.5 Parallel Resampling

In the generic PF as given in Table 2.1, the sample generation and weight update steps can be executed out independently for each particle and, thus, implemented in parallel.

Resampling, however, cannot start until all the particles are generated and the value of the cumulative sum is known. Therefore, the resampling step is not naturally parallelizable and, thus, is computationally expensive.

Standard resampling methods pose a significant challenge for parallel implementations as it can only begin when all the weights are computed at the weight computation stage, and the cumulative sum of the weights is available [77]. This means that any parallel implementation would start the resampling only after all the weights are computed. This increases the execution time of the entire PF implementation.

Partial Resampling

The idea of partial resampling [9] is to perform resampling only on particles with large weights and particles with small weights. Particles with moderate weights are not resampled. The main advantages of this method are that it is faster because it is done on a subset of particles and the communication is shorter since a smaller number of particles are replicated and replaced.

Resampling with Proportional Allocation (RPA), [10]

The algorithm is based on stratified sampling [30] with proportional allocation. The sample space is partitioned into p groups or strata and each stratum corresponds to a processing unit. Proportional allocation among strata is used, which means that more samples are drawn from the strata with larger weights. After the weights of the strata are known, the number of particles that each stratum replicates is calculated at central unit using residual systematic resampling, and this process is denoted as *inter-resampling* since it treats the processing units as single particles. Finally, resampling is performed inside the strata (at each processing unit, in parallel) which is referred to as *intra-resampling.* Therefore, the resampling algorithm is accelerated by using loop transformation, or specifically loop distribution [86], which allows for having an inner loop that can run in parallel on the processing units (intra-resampling) with small sequential centralized processing (inter-resampling) at central unit.

Resampling with Nonproportional Allocation (RNA), [10]

This algorithm is a modification of the distributed RPA. RPA requires a complicated scheme for particle routing and there is a need for an additional global preprocessing step (inter-resampling) at central unit which introduces an extra delay. These problems can be solved by using an RNA algorithm. In distributed RNA particle routing is deterministic and planned in advance by a designer. To achieve this, groups of one or more processing units are formed. In RPA the number of particles drawn is proportional to the weight of the stratum. On the other hand, in RNA the number of particles within a group after resampling is fixed and equal to the number of particles per group. Therefore, full independent resampling is performed in parallel by each group. More details can be found in [10].

While RPA and RNA were proposed an studied in the context of FPGA, we adopt and study them for our computer cluster and GPU implementations.

2.2 Particle Flow Filtering

Recently a new class of nonlinear filters has been proposed by Fred Daum & Jim Huang—the particle flow filters (PFFs) [15, 17, 16, 27, 28, 21, 20, 19, 25, 24, 22, 23, 26]. They can solve the well known problem of particle degeneracy of the PFs (and other, e.g., deterministic sampling methods for density-based nonlinear filtering [59] caused by the pointwise multiplication of the Bayes rule. The method for samples' update is deterministic and is conceptually based on a natural homotopy relating the prior and posterior filter densities which induces a flow of the samples from the prior density towards a set of samples from the posterior. A flow of each particle from prior to posterior is governed by a flow equation—an ordinary differential equation (DE)—which in general satisfies a linear partial DE (PDE) with constant coefficients. This PDE is central in this approach. If no other constraints on the flow are imposed, it is underdetermined (except for the scalar state) which can be exploited to look for solutions with some desirable properties (see e.g., [22]) such as: low computational complexity for real-time computing; stability of the flow dynamic system; sufficient accuracy; uniqueness of the solution, and possibly others. Daum & Huang have already proposed many possible methods and algorithms to solve the PDE and outlined promising new possibilities for future research in this direction, e.g., [27, 28, 18]. While a numerical integration of the PDE is challenging for real-time computation, for some special distributions, e.g. Gaussian and exponential families it is analytically tractable. The explicit solution for (unnormalized) Gaussian prior and likelihood, referred to as the *exact* PFF [28] is straightforward to implement and fast for computation. Simulation results have been reported by the authors of the PFF method, e.g., [21, 24, 23], showing that PFFs can outperform by far other nonlinear filters in computation and accuracy—"roughly seven orders of magnitude faster than standard particle filters for the same estimation accuracy" [24], and about two orders of magnitude more accurate than EKF and UKF for difficult nonlinear problems. Another property of the PFFs is that almost all computations are independent and can be conducted in a parallel/distrubuted manner as opposed to PFs where resampling is a bottleneck. This makes PFFs even more attractive and promising for P&D computing. It also motivated us to pursue parallel implementations of PFF and make a quantitative performance evaluation and comparison with parallel PF algorithms studied by us before [46, 45, 88].

2.2.1 Gaussian Exact Particle Flow Algorithm

In this dissertation we limit our consideration to the Gaussian Exact PFF [20]. We summarize one time-step, $k-1 \longrightarrow k$, of the algorithm in Table 2.2 [47].

First, note that the prior and posterior at each time-step k are also represented through samples: $\{\bar{x}_k^i\}_{i=1}^{\bar{N}}$ and $\{x_k^i\}_{i=1}^{N}$, respectively. In Table 2.2 we give the prediction step in terms of random sampling (as in the bootstap PF) which amounts to passing each particle from the posterior through the stochastic state dynamic model. However the prediction sampling need not be random in general—the prior can be approximated by deterministic samples as well, e.g., based on optimal Dirac mixture approximations [79]. In our implementation we use random sampling but deterministic sampling for prediction is of further interest for future implementation.

Second, the computation of the state prediction estimate $\bar{x}_k$, error covariance matrix $\bar{P}_k$, and linearized measurement model H_k is not explicitly given because it can be done in different ways, e.g., $\bar{x}_k$ and $\bar{P}_k$ can be computed as the sample mean an covariance of the particles and then H_k can be obtained via linearization of the measurement model about $\bar{x}_k$, or the computation can be done via EKF/UKF equations.

Third, the flow velocity parameters $A(\lambda)$, $b(\lambda)$ are common for all particles and, therefore,

Table 2.2: Gaussian Exact Particle Flow Algorithm [47]

```
• Particle Prediction
  – For $i = 1, \ldots, \bar{N}$
    Draw a predicted particle:   $\bar{x}_k^i \sim p\left(x_k | x_{k-1}^i\right)$
• Particle Update
  – Compute predicted estimate $\bar{x}_k$, its error covariance $P$,
    and linearized measurement model matrix $H$
  – Compute parameters of flow velocity (exact computation):
      $A(\lambda) = -\frac{1}{2} PH' \left(\lambda HPH' + R\right)^{-1} H$
      $b(\lambda) = (I + 2\lambda A)\left[(I + \lambda A) PH'R^{-1} z_k + A\bar{x}_k\right]$
  – For $i = 1, \ldots, N$
    Solve the particle flow ODE
        $\frac{dx}{d\lambda} = f(x, \lambda) = A(\lambda)x + b(\lambda)$
    for $\lambda \in [0\ 1]$ with initial condition $x(0) = \bar{x}_k^i$;
    Let $x_k^i := x(1)$.
```

their computation is given outside the `for`-loop for solving the ODE in Table 2.2.

2.3 Characterization of Parallel Algorithms

Parallel computer systems have been available for many years. Parallel computing is a form of computation in which many calculations are carried out simultaneously. Large problem could often be divided into smaller tasks across multiple processors. Investigating the performance of parallel computer systems is of great interest. Analyzing the performance of parallel computer systems means predicting its potential elapsed times for different input size, processor size and communication network. The results can be used in the design and implementation of practical applications.

For most parallel algorithms, execution time is of main concern. The other performance measurements in parallel algorithms are more complex. Unlike traditional algorithm complexity analysis, algorithm analysis requires two specifications [44]: characteristics of an algorithm and characteristics of the architecture. For each algorithm, performance analysis is to investigate the execution times while we alter algorithmic and computing environment specifications. Performance analysis of a parallel algorithm-architecture combination can be used to select the best algorithm-architecture combination for a problem. For a fixed problem size, it may be used to determine the optimal number of processors to be used and the maximum possible speedup that can be obtained. The performance analysis can also predict the impact of changing hardware technology on the performance and thus help design better parallel architectures for solving various problems.

2.3.1 Parallel Algorithms' Performance Measures

In sequential program scalability analysis, computing and input/output are the two major timing factors. For example, a typical fast sort program requires $O(n \log n)$ computing steps and $O(n)$ bytes for input/output when processing an input of size n. Unlike a sequential program, the key problem is that there are more performance sensitive parameters in parallel processing than that in sequential programming.

The importance of parallel algorithm analysis has been widely recognized [43, 55, 80]. Scalability metric [42] concerns a list of attributes that are considered important to the scalability of a parallel computer system. It can help guide parallel program development. However, it lacks a generic analysis tool. Scalability definition [72] was based on the ratio of the asymptotic speedup of an algorithm on a real machine to the asymptotic speedup on an ideal realization on an exclusive read exclusive write parallel random access machine. It does not include any resource specifications, such as processor and network capacities. The LogP model [14] tracks the communication overhead by analyzing detailed message passing patterns and latencies. It does not include processing time modeling. Next we describe the terminology that is used in the rest of the dissertation [53].

Execution Time T_p***:*** The time elapsed from the moment a multiprocessor computation starts to the moment the last processor finishes execution using p processors.

Speedup S***:*** The ratio of the serial execution time of the serial algorithm (T_1) to the parallel execution time of the chosen algorithm (T_p), i.e., $S = \frac{T_1}{T_p}$. The speedup characterizes the *scalability* of a parallel algorithm. Ideally, the best speedup is linear, i.e., $S = p$.

Efficiency E***:*** The ratio of speedup (S) to the number of processors (p). Thus, $E = \frac{S}{p} = \frac{T_s}{pT_p}$. The efficiency characterizes how well the processors are utilized. Ideally, it has values between 0 and 1. Algorithms with linear speedup have efficiency of 1 (the best).

Actually, if a parallel system is used to solve a problem instance of a fixed size, then the efficiency decreases as p increases. It is a well known that given a parallel architecture and a problem instance of a fixed size, the speedup of a parallel algorithm does not continue to increase with increasing number of processors. The speedup tends to saturate or peak at a certain value. In 1967, Amdahl [1] made the observation that if s is the serial fraction in an algorithm, then its speedup is bounded by $\frac{1}{s}$, no matter how many processors are used. For example, if an algorithm runs 10 second using one processor, and some part of the algorithm which cannot be parallelized takes 1 second to execute, while the remaining part which can be parallelized takes 9 seconds, then no matter how many processors are used to parallelized execution of this algorithm, the minimum execution time cannot be less than 1 second. Thus, the speedup is limited to at most 10. This statement, now popularly known as Amdahl's law, also known as Amdahl's argument, has been used to find the maximum improvement to a parallel system.

Actually, in addition to the serial fraction, the speedup obtained by a parallel system depends on a number of factors such as the degree of concurrency and overheads due to communication, synchronization, redundant work etc. For a fixed problem size, the speedup saturates either because the overheads grow with increasing number of processors or because the number of processors eventually exceeds the degree of concurrency inherent in the algorithm. A number of researchers have analyzed the optimal number of processors required to minimize parallel execution time for a variety of problems [33, 61, 71, 83].

In our research, we designed algorithms and analyzed their performance on various parallel systems for non-lineal filtering problems. The primary measures of parallel performance we used were *execution time*, *speedup*, and *efficiency*, as defined above.

Chapter 3

Parallel Filtering & Tracking Algorithms for Computer Cluster

Computer clusters emerged as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing. A computer cluster is a group of linked computers working together closely so that they perform like a single computer [4]. Commonly, the components of a cluster are connected to each other through fast local area networks (LAN). As far as the computers are tightly connected they can be viewed as a single system in many respects. Each node (computer) is used as a server running its own instance of an operating system.

Clusters are usually deployed to improve performance and availability over that of a single computer, while typically being much more cost-effective than single computers of comparable speed or availability. Computer clustering relies on a centralized management approach which makes the nodes available as orchestrated shared servers. It is distinct from other approaches such as peer to peer or grid computing which also use many nodes, but with a far more distributed nature.

3.1 Attributes of Computer Cluster

There are some general characteristics of computer cluster. First, it consists of many of the same or similar type of machines. Second, all machines are connected using fast network connections. Third, all machines have their own resources such as memory. Fourth, they must have software such as an Message Passing Interface (MPI) implementation installed to allow programs to be run across all nodes.

3.1.1 Message Passing and Communication

Two widely used approaches for communication between cluster nodes are the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). In our research we used MPI as a software environment.

MPICH

MPI [67] is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.

MPICH [68] is a freely available, portable implementation of MPI, a standard for message-passing for distributed-memory applications used in parallel computing. The CH part of the name was derived from "Chameleon", which was a portable parallel programming library developed by William Gropp, one of the founders of MPICH. It is a high-performance and widely portable implementation of the MPI standard and runs on parallel systems of all sizes, from multicore nodes to clusters to large supercomputers. Many of the largest systems on the Top500 list run MPICH. It also provides a vehicle for MPI implementation research and for developing new and better parallel programming environments.

3.2 Generic Parallel PF Algorithms for Computer Cluster

3.2.1 Parallelization & Implementation for Computer Cluster

An apparent property of the SIR algorithm in Table 2.1, from computational point of view, is that a part of it is naturally parallel. That is, the computation of the algorithm for IS is independent for each particle, and thus can be conducted in parallel without any communication among the processors. Effective parallelization of the resampling part however is nontrivial because generation of a single resampled particle requires information from all particles of the sample set. Thus, in parallel and distributed implementations, resampling becomes a bottleneck due to its sequential nature and it imposes an increased complexity on the communications and data traffic between processors. Several parallel/distributed resampling schemes have been already proposed in the literature. In [6] three parallel PFs, referred to as a global distributed PF (GDPF), a local distributed PF (LDPF), and a compressed distributed PF (CDPF), respectively. PFs with distributed resampling schemes, referred to as RPA and RNA have been proposed in [10] for implementation on a field programmable gate array (FPGA). We implement and study the performance of parallel PFs with centralized resampling (CR), RPA, and RNA on a computer cluster for two target tracking applications: space object tracking and ground multitarget tracking using image sensor.

We consider a generic parallel computing environment (such as a Beowulf type computer cluster [82]) with p processors: a central processor – *head node* (HN), charged with directing the computations and communication control, and $p-1$ *processing nodes* (PNs), all interconnected, e.g, through fast local area networks. The HN can also operate as a PN.

Parallel PF with Centralized Resampling

This algorithm is straightforward. HN partition the set of all N particles into p subsets and sends each subset to a PN. All PNs (in parallel) perform prediction (draw predicted samples and calculate importance weights), and send the predicted particles (with weights) to HN. HN performs resampling.

Parallel PF with distributed RPA

The algorithm is based on stratified sampling with proportional allocation [10]. The sample space is partitioned into p groups or strata and each stratum corresponds to a PN. Proportional allocation among strata is used, which means that more samples are drawn from the strata with larger weights. After the weights of the strata are known, the number of particles that each stratum replicates is calculated at HN using residual systematic resampling, and this process is denoted as *inter-resampling* since it treats the PNs as single particles. Finally, resampling is performed inside the strata (at each PN, in parallel) which is referred to as *intra-resampling*.

Parallel PF with distributed RNA

This algorithm is a modification of the distributed RPA. RPA requires a complicated scheme for particle routing and there is a need for an additional global preprocessing step (inter-resampling) at HN which introduces an extra delay. These problems are solved by using distributed RNA where particle routing is deterministic and planned in advance by design. To achieve this, groups of one or more PNs are formed. In RPA the number of particles drawn is proportional to the weight of the stratum. On the other hand, in RNA the number of particles within a group after resampling is fixed and equal to the number of particles per group. Therefore, full independent resampling is performed in parallel by each group.

For more details on RPA, and RNA the reader is referred to [10].

3.2.2 Case Study I: Space Object Tracking

The Cluster Implementation

We used a Linux x86 5TF cluster which are Dell-based systems. Each cluster consists of 128 computer nodes, and is capable of 4.77 TFlops peak performance. Each node contains two Intel dual-core Xeon 64-bit processors operating at a frequency of 2.33 GHz. They run the Red Hat Enterprise Linux 4 operating system. The program code was written on MPI [36]. Both point-to-point and collective communication are supported.

The first MPI routine called in any MPI program is the initialization routine MPI_INIT. In our implementation the *head node* (HN) executes function `master()`, and a *processing node* (PN) executes function `slave()`. A high level pseudo-code of the implemented PPF with CR, RPA, and RNA, respectively is given below.

PPF with Centralized Resampling

```
int master() {
  generate();          // generate particles
  send();              // send particles to PNs
  while (iteration <= max_iteration)
    receive();         // receive particles & weights from PNs
    normalize_weight();
    resampling();
    send();            // send particles to PNs
    calculate_sample_mean();
    iteration = iteration + 1;
  end while }
int slave() {
  while (iteration <= max_iteration)
    receive();         // receive particles from HN
    prediction();      // sampling and weights calculation
    send();            // send particles & weights to HN
    iteration = iteration + 1;
  end while }
```

PPF with Proportional Allocation

```
int master() {
  generate();          // generate particles
  send();              // send particles to PNs
  while (iteration <= max_iteration)
    receive();         //receive weight from PNs;
    inter-resampling();
    determine_rout();  // determines a scheme for particle
                          exchange routing with other PNs
    send();            // send the scheme to PNs
    receive();         // receive sample mean from PNs
    calculate_sample_mean();
    iteration = iteration + 1;
  end while }
int slave() {
  receive();           // receive particles from HN
  while (iteration <= max_iteration)
    prediction();
    calculate_weight();
    send();            // send weight to HN
    receive();         // receive particle exchange routing
    intra-resampling;
    exchange_particle(); //exchange particles with other slaves
    calculate_weight_sum();
    send();            // send weight_sum to HN
    iteration = iteration + 1;
  end while }
```

PPF with Non-proportional Allocation

```
int master() {
  group();             // group the PNs
  generate();          // generate particles
  send();              // send particles to each group
  while (iteration <= max_iteration)
    receive();         // receive weight_sum from groups
    calculate_sample_mean();
    iteration = iteration + 1;
  end while }
```

```
// Resampling and particle routing are performed inside groups using RPA.
```

Space Object Tracking Problem

For a benchmark of the parallel implementations we choose a space object tracking problem introduced in [12]. In this problem the object (target) is a satellite in geostationary orbit (GEO) and the measurements are provided by sensors onboard of low Earth orbit (LEO) satellites. The geometry of a typical tracking scenario is illustrated in Fig. 3.1.

Target State Propagation Model

Denote by $\mathbf{x}(t)$ the continuous time target state given by

$$\mathbf{x}(t) \triangleq [\mathbf{r}(t)\ \dot{\mathbf{r}}(t)]' = [x(t)\ y(t)\ z(t)\ v_x(t)\ v_y(t)\ v_z(t)]',$$

where $\mathbf{r}(t)$ and $\dot{\mathbf{r}}(t)$ are the position and velocity vectors, respectively. The target dynamics model is

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x}(t)) + \mathbf{w}(t) \tag{3.1}$$

where $\mathbf{f}(t,\mathbf{x}) = [v_x\ v_y\ v_z\ -(\mu/r^3)x\ -(\mu/r^3)y\ -(\mu/r^3)z]'$, $r = \sqrt{x^2+y^2+z^2}$, μ is the Earth gravitational constant, and $\mathbf{w} = [0\ 0\ 0\ w_x\ w_y\ w_z]'$ is a random process noise which accounts for trajectory perturbations, model inaccuracies, and can serve as a simple model of some maneuvering.

Let t_k denote the kth sampling time, for $k = 0, 1 \ldots$, and $\mathbf{x}_k \triangleq \mathbf{x}(t_k)$. After discretization, the approximate model is

$$\mathbf{x}_k = \mathbf{f}_k(\mathbf{x}_{k-1}) + \mathbf{w}_k \tag{3.2}$$

where $\mathbf{f}_k(\mathbf{x}_{k-1}) \triangleq \mathbf{x}_{k-1} + \int_{t_{k-1}}^{t_k} \mathbf{f}\,(t, \mathbf{x}(t))\,dt$ and $\mathbf{w}_k$ is zero-mean white Gaussian noise (WGN) with covariance Q_k. Then, the corresponding state transition Markov model, needed for (2.1a), is given by

$$p\left(\mathbf{x}_k|\mathbf{x}_{k-1}\right) = p_{\mathbf{w}_k}\left(\mathbf{x}_k - \mathbf{f}_k(\mathbf{x_{k-1}})\right) = \mathcal{N}(\mathbf{x}_k - \mathbf{f}_k(\mathbf{x}_{k-1}); \mathbf{0}, Q_k) \tag{3.3}$$

where $\mathcal{N}$ denotes a multivariate Gaussian PDF.

Measurement Model

The most popular sensor onboard a space satellite is the space-based visible (SBV) sensor which uses visible band electro-optical camera to measure the azimuth and elevation of a target within the sensor's field of view. Here, for the purpose of the simulation study, it is assumed more generally that sensors onboard LEO satellites can provide the following type of measurements: range, azimuth, and elevation, defined below.

$$\text{Range:} \qquad h_r^{(i)} = \sqrt{\left(x - x^{(i)}\right)^2 + \left(y - y^{(i)}\right)^2 + \left(z - z^{(i)}\right)^2} \tag{3.4}$$

$$\text{Azimuth:} \qquad h_a^{(i)} = \tan^{-1}\left(\frac{y - y^{(i)}}{x - x^{(i)}}\right) \tag{3.5}$$

$$\text{Elevation:} \qquad h_e^{(i)} = \tan^{-1}\left(\frac{z - z^{(i)}}{\sqrt{\left(x - x^{(i)}\right)^2 + \left(y - y^{(i)}\right)^2}}\right) \tag{3.6}$$

where $\left(x^{(i)}, y^{(i)}, z^{(i)}\right)$ is the i-th observer location (known), and (x, y, z) is the target location.

The measurement model (with the observer's index omitted) is given by

$$\mathbf{z}_k = \mathbf{h}_k(\mathbf{x}_k) + \mathbf{v}_k \tag{3.7}$$

where $\mathbf{h}_k(\mathbf{x}_k) \triangleq [h_r(\mathbf{x}_k)\ h_a(\mathbf{x}_k)\ h_e(\mathbf{x}_k)]'$ and $\mathbf{v}_k$ is zero-mean WGN with covariance R_k which models the measurement errors. Then the corresponding likelihood function, needed for

(2.1b), is given by

$$p\left(\mathbf{z}_k|\mathbf{x}_k\right) = p_{\mathbf{v}_k}\left(\mathbf{z}_k - \mathbf{h}_k(\mathbf{x}_k)\right) = \mathcal{N}(\mathbf{z}_k - \mathbf{h}_k(\mathbf{x}_k); \mathbf{0}, R_k) \tag{3.8}$$

Experiment Setup

The simulated benchmark scenario includes a single LEO observer that tracks a GEO satellite in non-maneuvering mode of motion. The LEO orbit is nearly circular with a radius of 6600km and its position is assumed to be accurately calibrated by the GPS. The target being tracked is in a GEO with radius approximately 42164km. Range, azimuth and elevation measurements of the target are simulated without false alarm or missed detection except when the line of sight between the observer and the target is blocked by the earth. The standard deviations of measurement error (matrix R_k) are 0.1km, 2mrad/sec, 2mrad/sec, for range, azimuth, elevation, respectively. The sensor onboard the observer has a fixed sampling rate of 0.02Hz. For simplicity, both the observer and the target share the same orbit plane. The target has white process noise with the magnitude of random acceleration (matrix Q_k) at 0.01m/s^2. Fig. 3.1 illustrates the tracking scenario after 500 iterations. All three filters were initialized by the first measurement with the same initial covariance matrix. In order to obtain statistically significant evaluation of the performance metrics, 100 Monte Carlo runs were performed for each set of scenario parameters (i.e, different number of particles, different number of processors, etc.).

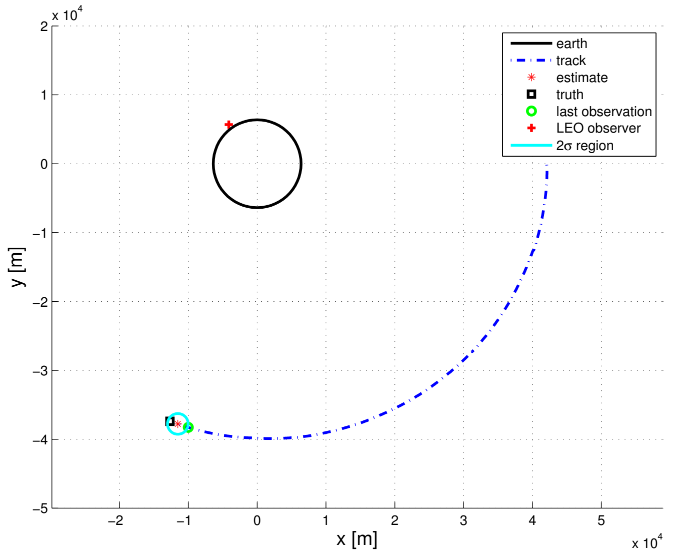


Figure 3.1: Target Trajectory Estimation; Single MC Run; 500 Iterations.

Performance Measures – Tracking performance

The accuracy of state estimation for all filters was measured in terms of *time averaged root mean square error* (TARMSE), defined (for both position and velocity), as follows

$$TARMSE_{m,n} = \frac{1}{n-m} \sum_{k=m+1}^{n} RMSE_k \tag{3.9}$$

where (for position) $RMSE_k = \left(\frac{1}{M} \sum_{i=1}^{M} [(x_k^{(i)} - \hat{x}_k^{(i)})^2 + (y_k^{(i)} - \hat{y}_k^{(i)})^2 + (z_k^{(i)} - \hat{z}_k^{(i)})^2] \right)^{1/2}$, $i = 1, \ldots, M$ denotes the ith MC run, and $[m+1, n]$ is the time interval of averaging. Likewise for velocity. TARMSE is a, commonly used in target tracking, single measure of a filter accuracy in an interval of “steady-state”, e.g., $[m+1, n]$.

Performance Measures – Computational Performance

The computational performance of the parallel PFs was measured in terms of the following measures, commonly used in parallel computing.

$$\textit{Execution time}: \quad T_p = \text{Execution time for one iteration using } p \text{ processors} \tag{3.10}$$

$$\textit{Speedup}: \quad S_p = \frac{T_1}{T_p} = \frac{\text{Execution time for one iteration using 1 processor}}{\text{Execution time for one iteration using } p \text{ processors}} \tag{3.11}$$

$$\textit{Efficiency}: \quad E_p = \frac{S_p}{p} = \frac{\text{Speedup}}{\text{Number of processors used}} \tag{3.12}$$

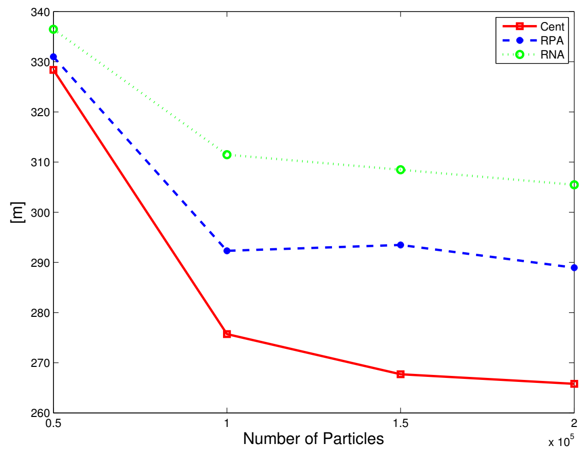
The speedup characterizes the *scalability* of a parallel algorithm. Ideally, the best speedup is linear, i.e., $S_p = p$. The efficiency characterizes how well the processors are utilized. Ideally, it has values between 0 and 1.

Results & Comparative Analysis

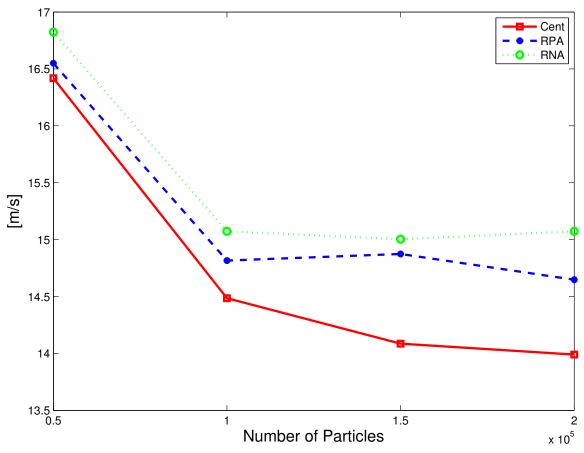
Due to space limitation only most representative results are reported in this dissertation.

Fig. 3.2 shows the TARMSE (position and velocity) and execution times of the three filters using $p = 64$ processors versus the number of particles used. Similar results (not presented here) were obtained for different number of processors, e.g., $p = 1, 2, 4, 16, 32, 64, 128$. They all indicated (as illustrated in Fig. 3.2, (a) and (b)) that for the RPA and RNA filters a significant improvement of accuracy is achieved by increasing the number of particles up to 100K, and for the CR filter – up to 150K. As it appears, for practical purposes, no more than 100K particles are needed for this tracking problem if RPA or RNA is used, and no more than 150K – if CR is used, regardless of the number of processors.

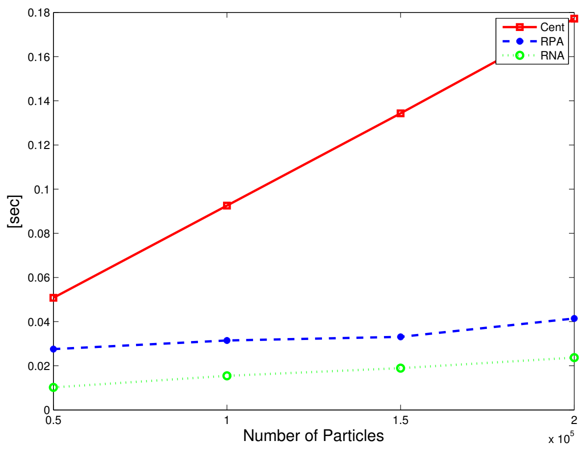
Fig. 3.2, (a) and (b), also illustrate that (as expected) the CR is the best in terms of accuracy (for the same number of particles), followed by RPA. This agrees with the



(a) Position TARMSE



(b) Velocity TARMSE



(c) Execution Time

Figure 3.2: Tracking Performance & Execution Time; 64 Processors; 100 MC Runs.

Table 3.1: Execution Times

	Number of Particles											
	50,000			100,000			150,000			200,000		
p	CR	RPA	RNA	CR	RPA	RNA	CR	RPA	RNA	CR	RPA	RNA
1	0.5554	0.4492	0.3659	1.1165	0.9056	0.7326	1.6794	1.3551	1.0983	2.2455	1.8067	1.4708
2	0.4166	0.3430	0.2815	0.8374	0.6915	0.5636	1.2596	1.0348	0.8449	1.6841	1.3797	1.1314
4	0.2777	0.2368	0.1759	0.5582	0.4775	0.3522	0.8397	0.7145	0.5280	1.1228	0.9526	0.7071
8	0.1157	0.1021	0.0880	0.2326	0.2058	0.1761	0.3499	0.3080	0.2640	0.4678	0.4106	0.3536
16	0.0716	0.0490	0.0416	0.1449	0.0987	0.0831	0.2123	0.1469	0.1243	0.2881	0.1937	0.1665
32	0.0542	0.0268	0.0206	0.1057	0.0515	0.0410	0.1588	0.0740	0.0612	0.2127	0.0979	0.0815
48	0.0462	0.0205	0.0148	0.0923	0.0401	0.0280	0.1395	0.0550	0.0417	0.1865	0.0684	0.0546
64	0.0458	0.0206	0.0119	0.0891	0.0325	0.0219	0.1305	0.0444	0.0318	0.1776	0.0545	0.0420
80	0.0476	0.0197	0.0100	0.0876	0.0299	0.0179	0.1295	0.0385	0.0261	0.1721	0.0474	0.0344
96	0.0473	0.0216	0.0091	0.0877	0.0283	0.0155	0.1313	0.0360	0.0222	0.1746	0.0439	0.0293
112	0.0494	0.0205	0.0085	0.0893	0.0294	0.0142	0.1310	0.0330	0.0197	0.1752	0.0420	0.0254
128	0.0508	0.0260	0.0102	0.0925	0.0311	0.0155	0.1343	0.0336	0.0189	0.1772	0.0404	0.0237
144	0.0528	0.0266	0.0099	0.0954	0.0312	0.0147	0.1351	0.0385	0.0184	0.1834	0.0419	0.0230
160	0.0543	0.0287	0.0095	0.0937	0.0326	0.0138	0.1353	0.0374	0.0178	0.1794	0.0435	0.0223
176	0.0562	0.0308	0.0087	0.0964	0.0338	0.0129	0.1403	0.0395	0.0164	0.1810	0.0406	0.0207

understanding that the CR has the best utilization of particles because it implements the resampling exactly , and the RPA is less approximate than RNA. The differences in accuracy are (somewhat surprisingly) considerable. Quantitatively (based on all simulations with 150K particles), the CR filter is about 10% more accurate than the RPA filter, and the RPA filter itself is about 6% more accurate than the RNA filter. On the other hand, Fig. 3.2, (c) shows the execution time – the "price" paid to achieve the accuracies shown in Fig. 3.2, (a) and (b). Now the order of performance is reversed with the CR filter being significantly slower than the two other filters. Quantitatively (based on all simulations with 150K particles), the CR filter is more than 3 times slower than the RPA filter, and the RPA filter is about 40% slower than the RNA.

The rest of the presented results concern the parallel computational performances. Table 3.1 provides the obtained execution times of each filter for a wide range of number of processors p and number of particles N. Fig. 3.3 shows the execution time, speedup, and efficiency for all algorithms with 150K particles versus the number of processors. Table 3.2 provides the computational performances of each filter with 150K particles. First, based on all results, it is observed that the CR PPF has a quite poor scalability and efficiency. Practically, its

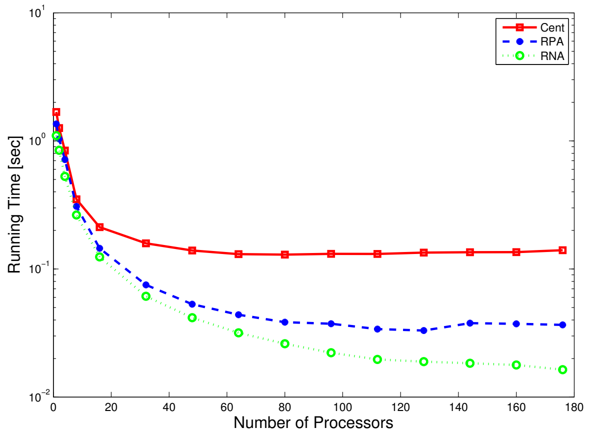
Table 3.2: Computational Performances; 100K Particles

$p =$		1	2	4	8	16	32	64	96	128	176
Cent	T_p	1.1165	0.8374	0.5582	0.2326	0.1449	0.1057	0.0891	0.0877	0.0925	0.0964
	S_P	1.0000	1.3333	2.0000	4.8000	7.7053	10.5650	12.5320	12.7365	12.0673	11.5843
	E_p	1.0000	0.6667	0.5000	0.6000	0.4816	0.3302	0.1958	0.1327	0.0943	0.0658
RPA	T_p	0.9030	0.6895	0.4761	0.2052	0.0977	0.0486	0.0302	0.0281	0.0315	0.0334
	S_P	1.0000	1.3095	1.8966	4.4000	9.2376	18.5841	29.8918	32.1571	28.6954	27.0579
	E_p	1.0000	0.6548	0.4741	0.5500	0.5774	0.5808	0.4671	0.3350	0.2242	0.1537
RNA	T_p	0.7326	0.5636	0.3522	0.1761	0.0831	0.0410	0.0219	0.0155	0.0155	0.0129
	S_P	1.0000	1.3000	2.0800	4.1600	8.8178	17.8831	33.3961	47.3319	47.3918	56.6979
	E_p	1.0000	0.6500	0.5200	0.5200	0.5511	0.5588	0.5218	0.4930	0.3702	0.3221

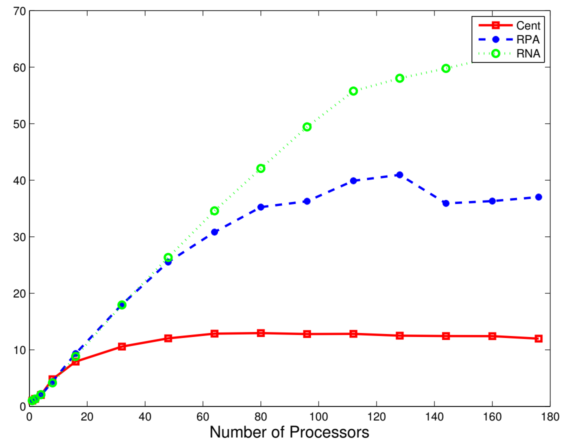
speedup remains constant for $p \geq 50$. Evidently, using CR PPF on more than 50 processors is of almost no use for this problem – most of the effort will be wasted on communication and synchronization. Second, the RPA PPF has significantly better parallel performance than CR. Practically, it provides and effective speedup (good scalability) up to the limit of about 110 processors with a reasonable (about 40%) efficiency within this limit. And, third the RNA PPF is considerably better than the RPA filter in terms of parallel performance. Quantitatively, its effective speedup limit (good scalability) is considerably higher – about 170 processors with about 45% efficiency within this limit.

3.2.3 Case Study II: Ground Multitarget Tracking using Image Sensor

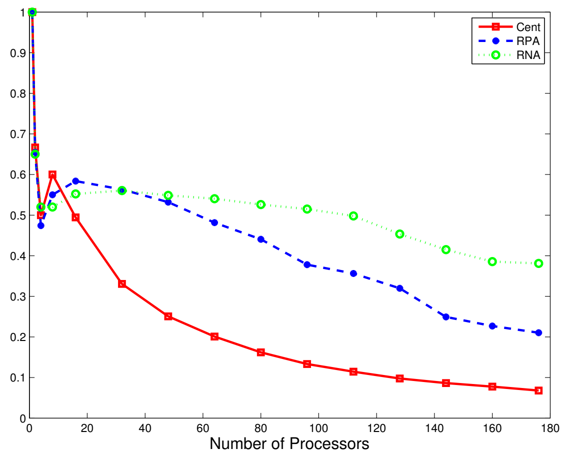
The main objective of this section is design, implementation, and performance evaluation of parallel multitarget tracking particle filters using a cluster of computers as a parallel computing environment. The parallel algorithms developed are based on the *Joint Multitarget Probability Density* (JMPD) algorithm for multiple target tracking [51, 52, 50, 65, 66, 49]. The resampling step is critical and computationally expensive in every PF implementation—It becomes a bottleneck due to its inherently sequential nature. To speed-up the computation,



(a) Execution Time



(b) Speedup



(c) Efficiency

Figure 3.3: Execution Time, Speedup & Efficiency; 150K Particles.

three techniques are implemented and experimented on a cluster for resampling – Centralized Resampling (CR), distributed Resampling with Proportional Allocation (RPA) and distributed Resampling with Nonproportional Allocation (RNA) [10], respectively. Another issue addressed is the inherent interdependence of the partitioning methods for JMPD tracking. Parallel versions of *Independent Partition* (IP), *Coupled Partition* (CP) and *Adaptive Partition* (AP) are developed and implemented that complete the design of several parallel PF algorithms for JMPD multitarget tracking.

Comprehensive experimentation by simulation of the algorithms over large-scale multitarget tracking scenarios (up to 50 targets) is conducted for performance evaluation. An analysis and comparison is made based on the obtained experimental data. The choice of the "best" among the six compared parallel particle filters for the considered multitarget tracking problem, taking into account the tracking and computational performances, imposes a complicated trade-off. In this regard, the obtained quantitative experimental results provide helpful information and guidelines in a practical design.

JMPD Particle Filter

In the JMPD approach to multiple target tracking [51, 52, 50, 65, 66, 49] the uncertainty about the number of targets present in a surveillance region as well as their individual states is represented by a single composite PDF. That is, the state of all targets is described by a meta-target state vector $X_k = \left(x_k^1, x_k^2, \ldots, x_k^T\right)$ where x_k^i is the state of target $i = 1, \ldots, T$. The number of targets at time k, $T_k \in [0, 1, 2, \ldots, \infty)$, is also assumed random. The posterior distribution of interest[1] is

$$p(X_k, T_k|Z^k) = p(X_k|T_k, Z^k)p(T_k|Z^k)$$

where $Z^k = \{Z_1, Z_2, \ldots, Z_k\}$ is the cumulative measurement set of the surveillance region up to time k, and $Z_l, l = 1, \ldots, k$ is the measurement set at time l. The model of target state and number evolution over time is given by $p(X_k, T_k|X_{k-1}, T_{k-1})$ and is referred to as the kinematic prior. It includes models of target motion, target birth and death, and any additional prior information on kinematics that may be available, e.g., terrain and road maps. The measurement model over the surveillance region is given by the likelihood $P(Z_k|X_k, T_k)$.

For the purpose of our implementation and performance study we adopt the kinematic prior and measurement models of [52].

Each target $i = 1, \ldots, T$ is assumed to follow a nearly constant velocity motion model

$$x_k^i = Fx_{k-1}^i + w_k^i \tag{3.13}$$

where $x = (\mathsf{x}, \dot{\mathsf{x}}, \mathsf{y}, \dot{\mathsf{y}})'$ is the state vector, $w \sim \mathcal{N}(0, Q)$ is white process noise with given covariance, and

$$F = diag\left\{\begin{bmatrix} 1 & \Delta \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & \Delta \\ 0 & 1 \end{bmatrix}\right\}$$

where Δ is the sampling interval. To account for maneuvers a mode variable can be also added [66]. The number of target is considered constant and known. Unknown number of targets using the transitional model of [66] will be included in future work.

It is assumed that a pixelized sensor provides raw (unthresholded) measurements data

[1] Note that in this formulation the so-called "mixed labeling" [7] is not addressed. It is assumed that no track extraction is needed and, consequently, the ordering of x^i within X is irrelevant as far as only the density is of interest. In [52] this assumption is referred to as a symmetry under permutation.

from the surveillance region according to the following *association-free model* [52] used often for track-before-detect (TBD) problems. A sensor scan at time k consists of the outputs of M pixels (cells of the region), i.e., $Z = \{z[1], \ldots, z[M]\}^2$ where $z[i]$ is the output of pixel i. The likelihood $P(Z|X,T)$ is given by

$$P(Z|X,T) \propto \prod_{i \in i_X} \frac{p_{n_{[i]}(X)}(z[i])}{p_0\,(z[i])} \tag{3.14}$$

where i_X is the set of all pixels that couple to X, $n_{[i]}(X)$ is the occupation number of pixel i (number of targets from X that lie in i). The output z of each pixel is assumed to follow the Rayleigh model

$$p_n(z) = \frac{z}{1+n\lambda} \exp\left(-\frac{z^2}{2\,(1+n\lambda)}\right),\ n = 0, 1, 2, \ldots \tag{3.15}$$

where λ denotes the signal-to-noise ratio (SNR).

With the definition of the transitional density $p(X_k, T_k|X_{k-1}, T_{k-1})$ and likelihood $P(Z_k|X_k, T_k)$ the solution to the multitarget tracking problem formally boils down to the BRF (1.1) and the standard SIS/R PF given above can be applied. However, with large number of target the computation requirements become prohibitive. The first step to improve the efficiency of the multitarget PF is to choose an appropriate importance (proposal) distribution for sampling that takes into account the specifics of the multitarget problem. Along with the SIR algorithm's *Kinematic Prior* (KP) proposal $\pi = p(X_k, T_k|X^i_{k-1}, T^i_{k-1})$, where $i-$ particle index, [52] suggested three more sophisticated schemes for choosing the importance distribution for multitarget PF, referred to as *Independent Partition* (IP), *Coupled Partition* (CP), and *Adaptive Partition* (AP) [52]. The second step is to parallelize as much as possible the resulting multitarget algorithms. In the next section we develop parallel implementation of

[2] Time index k is omitted here to lighten notation.

these schemes and incorporate them in the parallel structures of the corresponding overall multitarget parallel algorithms.

JMPD Parallel PF

In the JMPD SIR PF the proposal is just the kinematic prior and the IS step is completely decoupled with respect to particles $\{X^i\}$. Consequently, all three versions of the above generic parallel PF work without any modification. The significantly more efficient proposal schemes IP, CP, and AP of JMPD PF have intrinsic coupling among particles introduced by the dependence of the proposal on the current measurement data. By more careful inspection, however, IP and CP can be parallelized as given next.

Each particle i for T_i targets is $X^i = \left(x^{i,1}, x^{i,2}, \ldots, x^{i,T_k^i}\right)$ and $x^{i,j}$ is referred to as a partition j of particle i.

The IS step of the JMPD with IP can be done as follows:

A) For each partition $j = 1, \ldots, T_k^i$ (in parallel)

$$x_k^{ij}, w_k^{ij} = \quad \mathsf{IP}\ \left[\{x_{k-1}^{ij}, w_{k-1}^{ij}\}_{i=1}^{N}, Z_k\right]$$

B) For each particle $1 = 1, \ldots, N$

Importance weights $\widetilde{w}_k^i = w_{k-1}^i \frac{p\left(Z_k|X_k^i\right)}{\Pi_{\theta=1}^{T} w_k^{ij}}$

where IP denotes the IP subroutine of [52] which practically implements the SIR algorithm for each partition. The IS step of the JMPD with CP can be done similarly, except for IP being replaced by a subroutine CP which practically implements the known auxiliary SIR particle filter [30] for each partition but only outputs one resampled partition.

The local importance weights $w_k^{i,j}$ are data dependent and their inclusion in the calculation of importance weights $\widetilde{w}_k^i$ amounts to improving the proposal π – bias the proposal towards the optimal importance density $\pi(\cdot, Z_k)$.

Part A) can be integrated easily with any of above resampling schemes, CR, RPA, RNA. In the CR version part B) is naturally computed in HN where all particles are resampled. In RPA and RNA versions part B) can also be computed at HN at the expense of an extra communication between HN and all PNs, or it can be computed locally at each PN but this incurs extra pairwise (node-to-node) communication between all PNs. The latter option is parallel but not necessarily faster due to the communication overhead. In our cluster implementation we use the former option – compute B) at HN.

The IP method relies on exact labeling and is only appropriate for well separated (uncoupled) targets. the CP is appropriate for closely spaced (coupled) targets but is an expensive overkill if targets are not close. An *adaptive partition* (AP) can be achieved by spatial clustering of the individual partitions and applying IP and CP for the groups of independent and clustered partitions, respectively. In [52] this is achieved by using the K-means clustering algorithm, however, for many targets its sequential implementation (at the head node) is computationally prohibitive.

Implementation on Computer Cluster

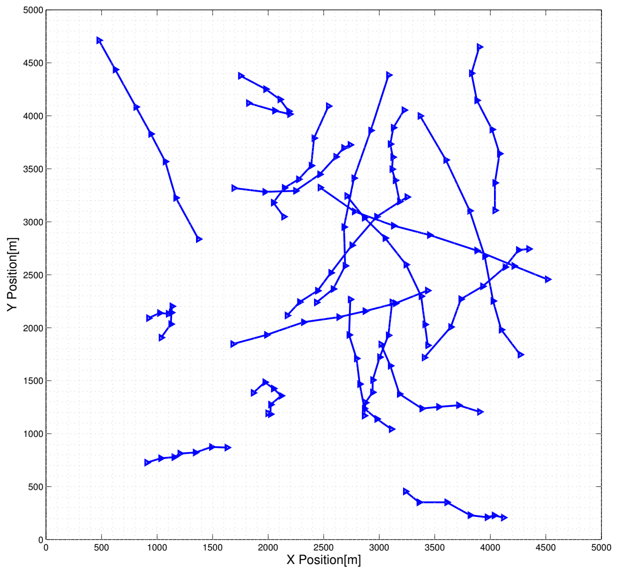
We used the cluster computer system *Poseidon*, housed at the University of New Orleans. *Poseidon* is a 128-node, 2 dual-core processor Red Hat Enterprise Linux (RHEL) v4 cluster from Dell with 2.33 GHz Intel Xeon 64bit processors and 4 GB RAM per node. It has 4.772 TFlops peak performance. It is part of the High Performance Computing (HPC) Louisiana Optical Network Initiative (LONI). More technical and performance details of this cluster can be found at URL: `http://www.hpc.lsu.edu/systems/`.

The program code was written on Message Passing Interface (MPI) [36]. MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported.

The first MPI routine called in any MPI program is the initialization routine MPI_INIT. In our implementation the *head node* (HN) executes function `master()`, and a *processing node* (PN) executes function `slave()`. A high-level pseudo-code of the implemented PPF with CR, RPA, and RNA, respectively is the same as in Case Study I.

Experiment Setup

The simulated multiple target tracking examples are based on the ground target tracking example of [52]. The targets move in a 5000m$\times$5000m surveillance area. They have a nearly constant velocity motion, according to (3.13) with $Q = diag\{20, 0.2, 20, 0.2\}$. The initial position of each target, for each Cartesian coordinate x and y, is generated randomly from the uniform distribution $\mathcal{U}(0, 5000)$, and the initial velocity of each target is also generated randomly from the uniform distribution $\mathcal{U}(-10, 10)$. The sensor scans a fixed rectangular region of 50×50 pixels, where each pixel represents a 100m$\times$100m area on the ground plane. The sensor returns Rayleigh-distributed (given by (3.15)) measurements in each pixel, depending on the number of targets that occupy the pixel according to the measurement model (3.14). The sensor sampling interval $\Delta = 1s$. The presented results are for SNR $\lambda = 15$. Scenarios with different number of targets were simulated, i.e., $T = 3, 10, 20, 30, 50$. The scenarios with $T = 20$ and $T = 50$ are illustrated in Fig. 3.4. The design parameter $R = 200$ in the CP algorithm. In order to obtain statistically significant evaluation of the performance metrics, 100 Monte Carlo runs were performed for each set of scenario and algorithms parameters (i.e, different number of particles, different number of processors, different number targets etc.) as described below.



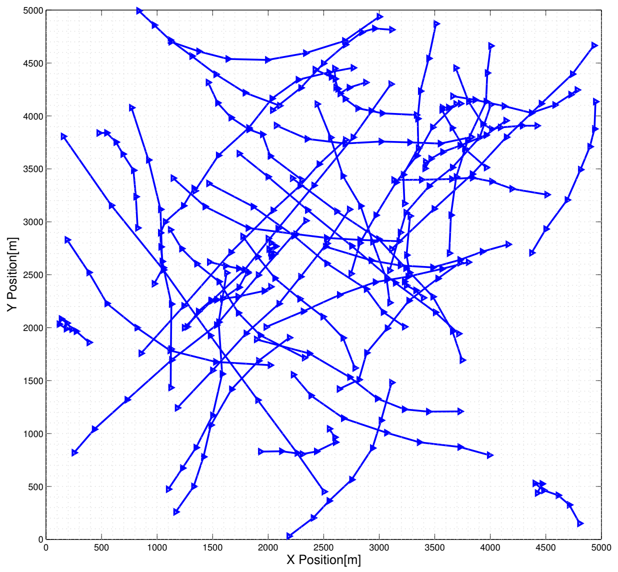


Figure 3.4: Scenarios with 20 Targets (top) and 50 Targets (bottom).

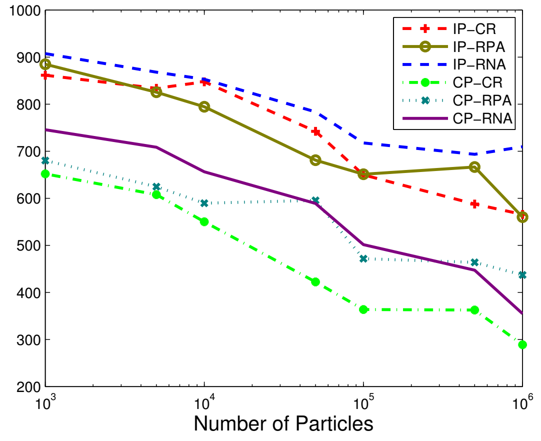


Figure 3.5: Position TARMSE; 64 Processors; 50 Targets; 100 MC Runs.

Performance Measures – Tracking performance

The accuracy of state estimation for all filters was measured in terms of *time averaged root mean square error* (TARMSE) [46], defined (for both position and velocity) by (3.9), where (for position)

$$RMSE_k = \left(\frac{1}{M} \sum_{i=1}^{M} [(x_k^{(i)} - \hat{x}_k^{(i)})^2 + (y_k^{(i)} - \hat{y}_k^{(i)})^2] \right)^{1/2} \tag{3.16}$$

where $i = 1, \ldots, M$ denotes the ith MC run. Likewise for velocity.

The state estimates for each target needed in (3.16) were obtained through K-means clustering of all posterior samples, as suggested by [52], and then computing each estimate as the sample mean of the particles within each cluster.

Performance Measures – Computational Performance

The algorithms' computational performance was measured in terms of *execution time*, *speedup*, and *efficiency*, given by (3.10), (3.11), and (3.12), respectively.

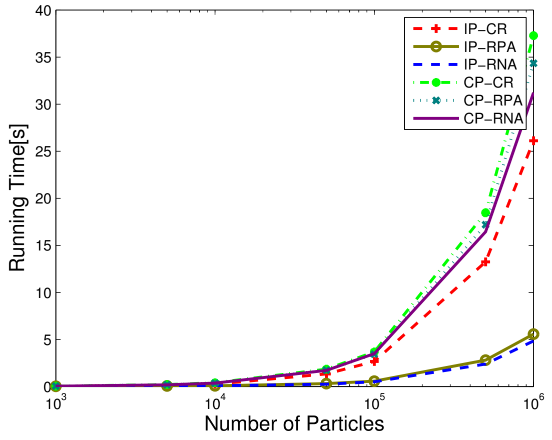


Figure 3.6: Average Execution Time; 64 Processors; 50 Targets; 100 MC Runs.

Results & Comparative Analysis

Due to space limitation, only the most representative results are reported in this dissertation. Included are the following six parallel multitarget tracking algorithms:

IP-CR	IP-RPA	IP-RNA
CP-CR	CP-RPA	CP-RNA

where IP and CP denote independent and coupled partition, respectively, and CR, RPA and RNA denote centralized resampling, resampling with proportional allocation and resampling with nonproportional allocation, respectively. More results are presented for the most difficult 50-target scenario Note that for $T = 50$ the dimension of the state vector is 200.

Fig. 3.5 shows the position TARMSE and Fig. 3.6 shows the execution times of the six filters using $p = 64$ processors versus the number of particles used. Similar results (not presented here) were obtained for different number of processors, e.g., $p = 1, 2, 4, 8, 16, 32, 64, 128, 256$. They all indicated (as illustrated in Fig. 3.5) that for all filters using slightly more than 100K particles is a reasonable choice for practical purposes, regardless of the number of processors used.

Table 3.3: Execution Time(s); 50 Targets

	Number of Particles											
	10, 000						100, 000					
p	IP-CR	IP-RPA	IP-RNA	CP-CR	CP-RPA	CP-RNA	IP-CR	IP-RPA	IP-RNA	CP-CR	CP-RPA	CP-RNA
1	0.9921	0.9883	0.9999	20.2018	20.3858	20.2411	9.9831	9.8122	9.9038	203.4868	204.7674	211.2306
2	0.9393	1.1786	1.2232	10.3009	10.3325	10.6168	9.2455	11.4716	11.7845	103.5883	103.2430	108.2115
4	0.5910	0.5894	0.6277	5.1990	5.2394	5.3241	5.9815	5.8952	6.0107	51.8891	51.6296	56.4860
8	0.4273	0.2885	0.2440	2.6219	2.6005	2.6810	4.0835	3.0311	2.5772	26.4441	25.8102	28.4187
16	0.3349	0.1759	0.1575	1.3376	1.3371	1.3784	3.3469	1.6788	1.5255	13.3824	13.1720	14.2625
32	0.2930	0.1057	0.0962	0.6964	0.6754	0.6767	2.8235	0.8976	0.8959	6.9516	6.6127	6.6826
64	0.2691	0.0797	0.0784	0.4093	0.3717	0.3695	2.6662	0.5734	0.5097	3.6776	3.4416	3.4927
128	0.2644	0.0977	0.0929	0.2414	0.2084	0.2021	2.5957	0.4191	0.3565	2.0892	1.8729	1.7946
256	0.3273	0.1313	0.1081	0.1727	0.2102	0.1997	2.5185	0.3913	0.3416	1.3248	1.1068	0.9871

Fig. 3.5 also illustrates that CP-CR is the best in terms of accuracy (for the same number of particles), followed by CP-RPA and CP-RNA. This agrees with 1) the understanding that CP is the right partition method to use because the targets are closely spaced and IP is inadequate, and 2) that CR has better utilization of particles than RPA and RNA because it implements the resampling exactly as opposed to RPA and RNA which are approximate. The differences in accuracy are considerable. Quantitatively (based on all simulations with 100K particles), CP-RPA and CP-RNA are about 20% less accurate than CP-CR, and all IP methods are more than 40% less accurate than CP-CR.

On the other hand, Fig. 3.6 shows the execution time – the "price" paid to achieve the accuracies shown in Fig. 3.5. Now the order of performance is reversed with IP-RNA and IP-RPA being significantly faster than all other filters. Quantitatively (based on all simulations with 100K particles - see Table 3.3), IP-RNA and IP-RPA (which are close in computation) are about 4.5 times faster than IP-CR and more than 6 times faster than CP-CR, CP-RPA, CP-RNA which are relatively close in computation time.

The rest of the presented results concern the parallel computational performances. Table 3.3 provides the obtained execution times of each filter for a wide range of number of processors p and number of particles N. Fig. 3.7 shows the execution time, speedup, and efficiency for all algorithms with 100K particles versus the number of processors. Table 3.4

Table 3.4: Computational Performances; 100K Particles, 50 Targets

$p =$		1	2	4	8	16	32	64	128	256
IP-CR	T_p	9.9831	9.2455	5.9815	4.0835	3.3469	2.8235	2.6662	2.5957	2.5185
	S_p	1.0000	1.0798	1.6690	2.4447	2.9828	3.5357	3.7443	3.8460	3.9638
	E_p	1.0000	0.5399	0.4172	0.3056	0.1864	0.1105	0.0585	0.0300	0.0155
IP-RPA	T_p	9.8122	11.4716	5.8952	3.0311	1.6788	0.8976	0.5734	0.4191	0.3913
	S_p	1.0000	0.8553	1.6644	3.2372	5.8448	10.9318	17.1135	23.4112	25.0727
	E_p	1.0000	0.4277	0.4161	0.4046	0.3653	0.3416	0.2674	0.1829	0.0979
IP-RNA	T_p	9.9038	11.7845	6.0107	2.5772	1.5255	0.8959	0.5097	0.3565	0.3416
	S_p	1.0000	0.8404	1.6477	3.8428	6.4922	11.0545	19.4294	27.7810	28.9935
	E_p	1.0000	0.4202	0.4119	0.4804	0.4058	0.3455	0.3036	0.2170	0.1133
CP-CR	T_p	203.4868	103.5883	51.8891	26.4441	13.3824	6.9516	3.6776	2.0892	1.3248
	S_p	1.0000	1.9644	3.9216	7.6950	15.2055	29.2720	55.3321	97.3990	153.5968
	E_p	1.0000	0.9822	0.9804	0.9619	0.9503	0.9148	0.8646	0.7609	0.6000
CP-RPA	T_p	204.7674	103.2430	51.6296	25.8102	13.1720	6.6127	3.4416	1.8729	1.1068
	S_p	1.0000	1.9834	3.9661	7.9336	15.5456	30.9657	59.4984	109.3300	185.0159
	E_p	1.0000	0.9917	0.9915	0.9917	0.9716	0.9677	0.9297	0.8541	0.7227
CP-RNA	T_p	211.2306	108.2115	56.4860	28.4187	14.2625	6.6826	3.4927	1.7946	0.9871
	S_p	1.0000	1.9520	3.7395	7.4328	14.8102	31.6091	60.4778	117.7012	213.9824
	E_p	1.0000	0.9760	0.9349	0.9291	0.9256	0.9878	0.9450	0.9195	0.8359

provides the computational performances of each filter with 100K particles.

First, based on all results, it is observed that IP-CR has a quite poor scalability and efficiency. Practically, its execution time remains constant for $p \geq 30$. Evidently, IP-CR PPF on more than 32 processors is of almost no use for this problem – most of the effort will be wasted on communication and synchronization. Second, IP-RNA and IP-RPA are better than IP-CR but much worse than all CP algorithms in terms of speedup and efficiency. For IP-RNA and IP-RPA there is no real use of increasing number of the processors after $p \geq 64$. Third, all CP algorithms have very good speedup and efficiency with a considerable advantage of CP-RNA (best) and CP-RPA (second best). This advantage increases significantly with increasing the number of processors, e.g., CP-RPA becomes more than 15% more efficient than CP-CR, and CP-RNA becomes more than 25% more efficient than CP-CR. It becomes clear from the results that within the CP algorithms RNA and RPA can help reduce considerably the computation of CP-CR if the number of processors $p \geq 128$.

Finally, Fig. 3.8 provides a comparison of the execution times of the tracking filters as a function of the number of targets.

Summary

In this case study, six parallel particle filters for multitarget tracking have been designed and implemented on a cluster of computers. Comprehensive experimentation by simulation of the algorithms over large-scale multitarget tracking scenarios has been conducted for performance evaluation, and comparison has been made based on the experimental data.

Overall, the CP-CR algorithm is the best in terms of accuracy at a given number of particles and its computation time is close to that of CP-RPA and CP-RNA. CP-RNA and CP-RPA have shown the best computational performance in terms of speedup and efficiency of the parallelization but they do not reduce the computation time of the CP-CR algorithm very significantly. This is because most of the computation time in CP-CR is spent on CP and very little on CR. So, speeding up CR by RPA or RNA with the same number of processors has a little effect on the overall algorithm.

All IP filters are significantly betters in terms of execution time but they have poorer scalability and tracking accuracy than the CP algorithms. RPA and RNA help greatly in reducing the execution time of IP-CR. This is because the resampling is a significant part of the computation with IP and thus RPA or RNA lead to a dramatic reduction of the computation time of the overall algorithm.

The choice of the "best" among the six compared parallel particle filters for the considered multitarget tracking problem, taking into account the tracking and computational performances, imposes a complicated trade-off. In this regard, the obtained quantitative experimental results provide helpful information and guidelines in a practical design.

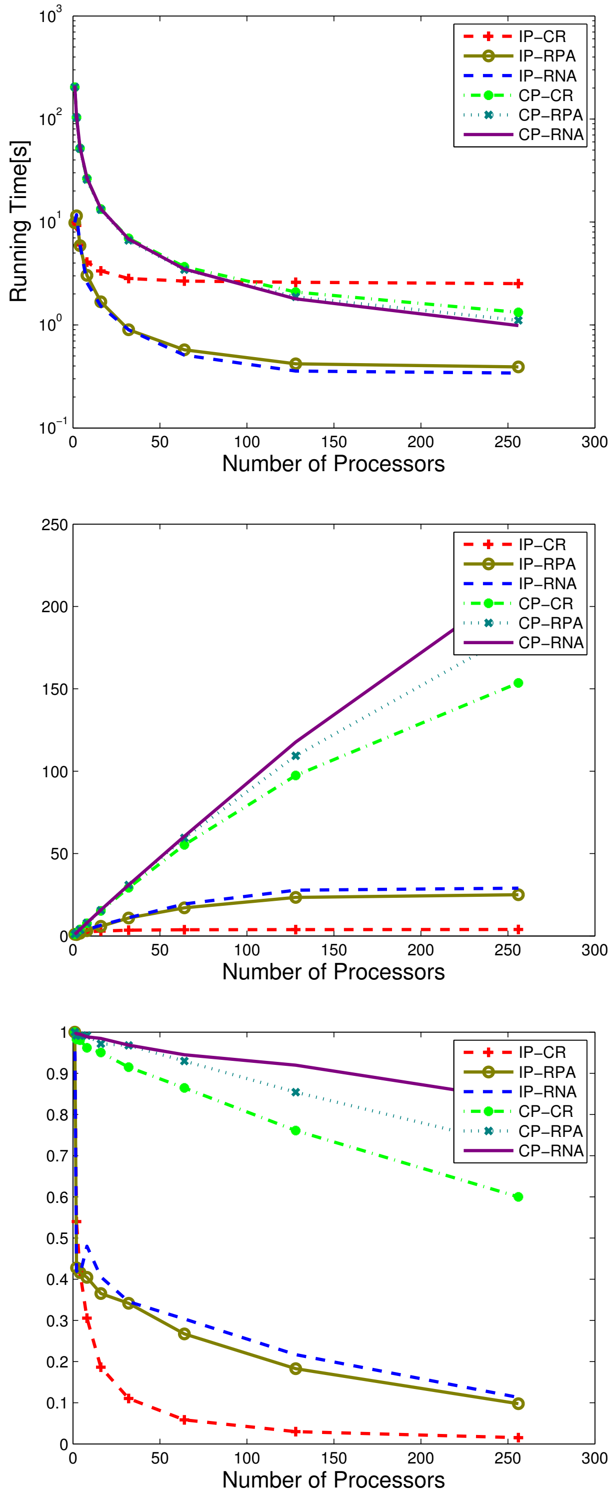


Figure 3.7: Execution Time (top) in Log-scale, Speedup (middle), Efficiency (bottom); 50 Targets; 100K Particles.

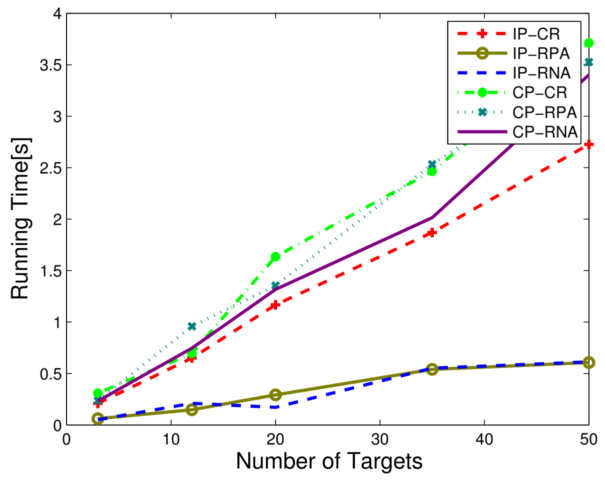


Figure 3.8: Execution Time; 64 Processors; 100K Particles.

3.3 Improved PF Algorithm for Computer Cluster

3.3.1 Particle Transfer Algorithm (PTA) for Load Balancing

Since the nodes of a computer cluster could communicate with each other easily, we propose to implement an improved algorithm that takes advantage of this capability — the PTA.

The main idea of PTA is to regroup computer nodes based on the weight of each node in order to achieve a better load balancing. It is similar to the RNA described in section 2.1.5 in that the processing nodes (PNs) are grouped, but the groups are adaptively changing in contrast to the RNA wherein the groups are fixed. The weight of each PN is calculated as a sum of the weights of the particles inside the PN, i.e., $W = \sum_{i=1}^{N} w^i$. After PN-resampling at each PN (Fig. 3.9), particles are transmitted from some PNs to other PNs using particle transfer algorithm to balance the loads of all PNs. PTA runs on each nodes (PNs and HN) and maintains a list of nodes information which include number of particles each node has. The list is generated by HN through running inter-resampling algorithm then broadcast the list to all other nodes. After each node gets the list, then each node executes the PTA algorithm. PTA ensures that all nodes in the cluster have the same list.

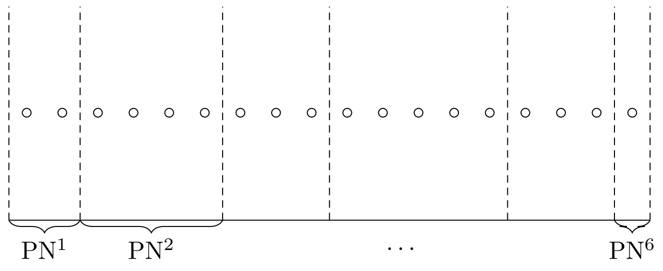


Figure 3.9: Unbalanced Stratified Resampling

An example of particle transfer is shown in Fig. 3.10. The computer cluster with 8 PNs is considered, where each PN processes $N = 100$ particles. After PN-resampling, the number of particles that each PN will produce is determined and it is $180, 57, 156, 88, 89, 16, 90$ and 124 respectively. The PNs are divided in two groups. One group has ordered PNs which number of particles is greater than 100, the other group has ordered PNs which number of particles is less than 100. Therefore, PNs $1, 3$ and 8 have surpluses of particles. In this example, PN1 send 80 particles to PN6, PN3 send 43 particles to PN2 and PN8 send 12 particles to PN4. After it, PNs $1, 2$ and 4 have 100 particles. Continue to do the processing with the remained PNs until all PNs have 100 particles.

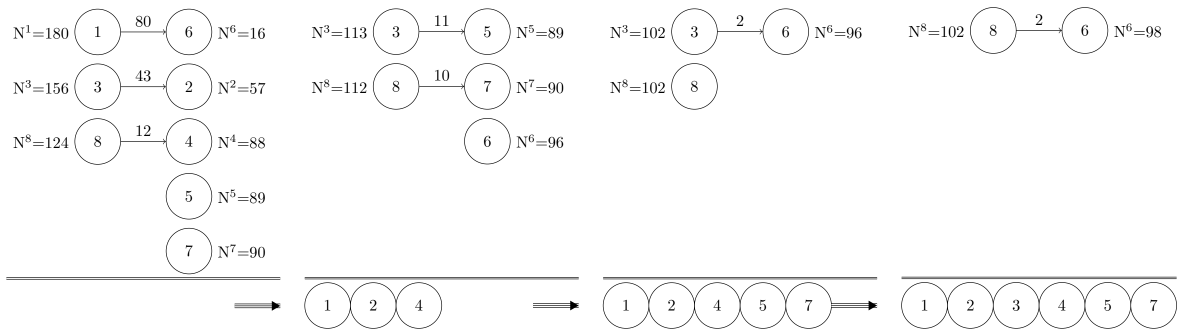


Figure 3.10: Particle Transfer Algorithm (PTA)

The main advantage of PTA lies in balancing the PN loads in a distributed resampling when the resampling is executed concurrently in the PNs instead in the HN. The time for

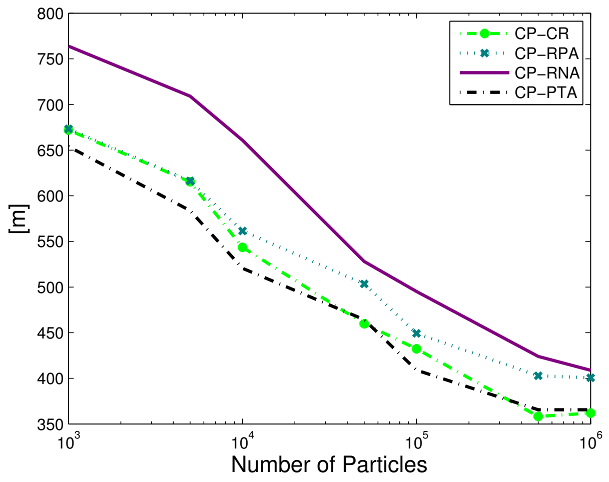


Figure 3.11: Position TARMSE; 64 Processors; 50 Targets; 100 MC Runs.

the resampling procedure in PTA is reduced $N/(N/K + K)$ times, where N is total number of particles, N/K denote the PN-resampling time, and K is the time for HN-resampling. It is important to note that HN-resampling requires global communication among the PNs, whereas PN-resampling is completely local within the PNs. PTA also requires communication between two PNs. The advantage is that the communication between each pair of PNs is done concurrently across pairs which helps save on communication time.

3.3.2 PTA Implementation & Simulation – Case Study II

Using again Case Study II (section 3.2.3), Fig. 3.11 shows the position TARMSE and Fig. 3.12 shows the execution times of the four filters using $p = 64$ processors versus the number of particles used. Similar results (not presented here) were obtained for different number of processors, e.g., $p = 1, 2, 4, 8, 16, 32, 64, 128, 256$.

It is seen from Fig. 3.11 that CP-CR and CP-PTA are the best in terms of accuracy (for the same number of particles), followed by CP-RPA and CP-RNA. Clearly, this is due the fact that PTA has better utilization of particles than RPA and RNA which are too

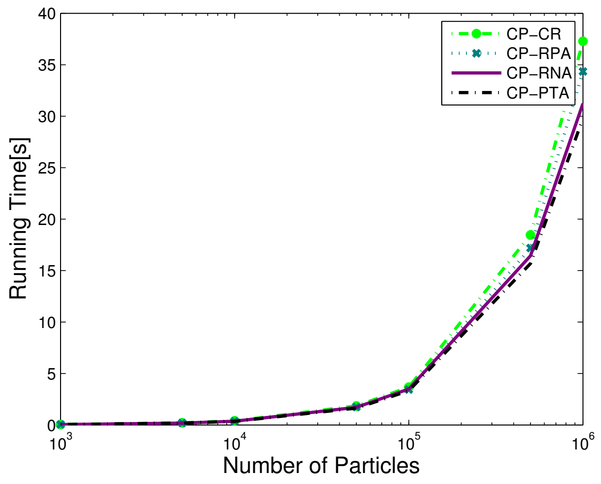


Figure 3.12: Execution Time; 64 Processors.

approximate.

On the other hand, Fig. 3.12 shows that CP-PTA is significantly faster than all other filters.

The rest of the presented results concern the parallel computational performances. Fig. 3.13 shows the execution time, speedup, and efficiency for all algorithms with 100K particles versus the number of processors.

Based on all results, it is observed that CP-CR has a quite poor scalability and efficiency. CP-PTA is the best of all algorithms in terms of speedup and efficiency. It becomes clear from the results that in comparison with the CP algorithms, the PTA can help reduce considerably the computation time and keep the best accuracy.

3.4 Summary

The computation of particle filter algorithm for importance sampling part is independent for each particle, and thus can be conducted in parallel without any communication among the

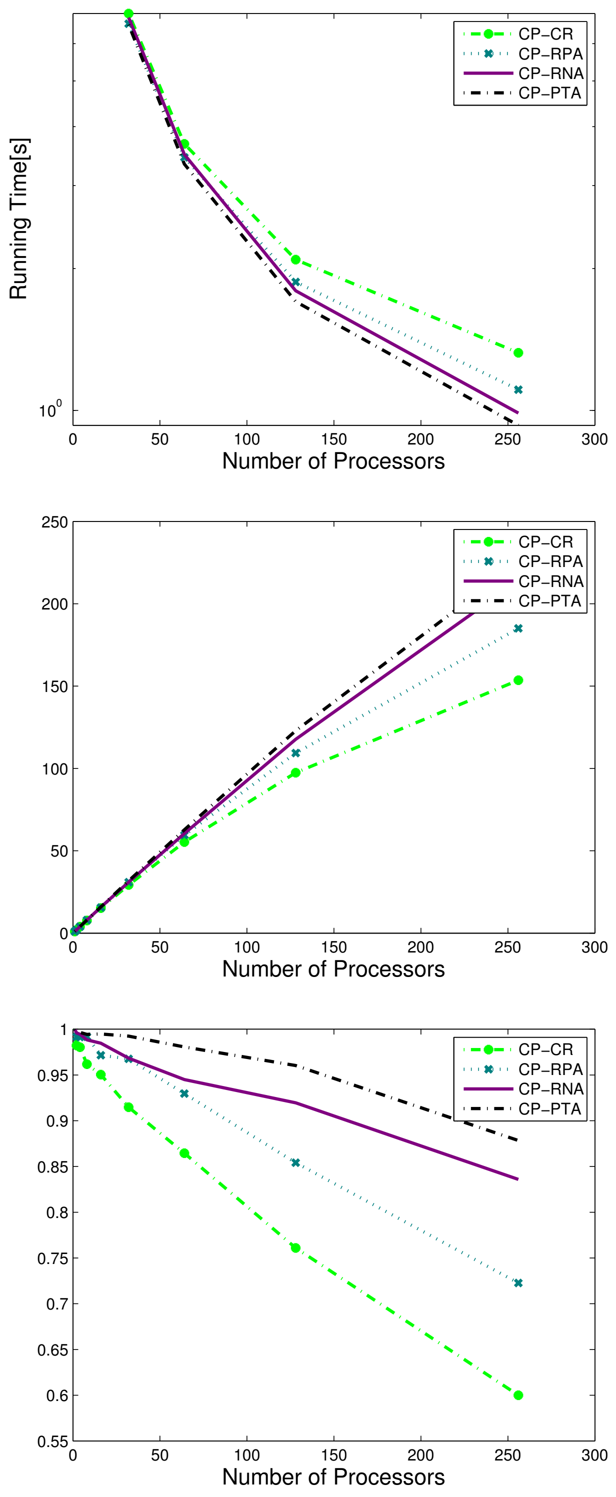


Figure 3.13: Execution Time (top) in Log-scale, Speedup (middle), Efficiency (bottom); 50 Targets; 100K Particles.

processors. Effective parallelization of the resampling part is nontrivial because generation of a single resampled particle requires information from all particles of the sample set. Thus, in parallel and distributed implementations, resampling becomes a bottleneck and it imposes an increased complexity on the communications and data traffic between processors. First, we implemented group resampling methods such as distributed RPA and RNA. Compared with central resampling method, the distributed resampling methods have the following advantages:

- Distributed group resampling is faster because it is done concurrently on subsets of particles of smaller size.
- Communication time is shorter since less particles are replicated and replaced.

However, group resampling is approximate, so its accuracy is less than that of a central resampling.

In order to further enhance accuracy and optimize the communication time, we proposed to use an improved algorithm for load balancing — the PTA. Based on the simulation, the PTA has shown quite significant advantage in terms of accuracy and computational performances as compared to other parallel PF algorithms. In the best cases, RPA has shown to be more than three times faster than CR, RNA – more than nine times faster than CR, and PTA – more than fifteen times faster than CR. On the other hand, their tracking performance degradation shown, was not that significant. In the worst cases the estimation error of RPA has increased up to about 10%, compared to that of CR; and that of RNA – up to about 17% , compared to that of CR. RNA has shown to be (in the best cases) up to 2.6 times faster than RPA, with an accuracy degradation (in the worst cases) – up to 7% as compared to RPA. The estimation error of PTA has almost the same with that of CR. So our improved implementation shows the best performance in terms of accuracy and computation.

Chapter 4

Parallel Filtering & Tracking Algorithms for Graphic Processing Unit

4.1 GPU & CUDA Computing

A graphics processing unit (GPU) is a single instruction multiple data (SIMD) parallel processor intended originally to meet the demands of computationally intensive tasks for real-time high-resolution 3D-graphics. Nowadays, GPUs are highly parallel multicore systems that can process very efficiently large blocks of data in parallel [63]. For highly parallelizable algorithms GPUs are becoming more efficient than the sequential central processing unit (CPU) [38]. On the other hand, GPUs are easily accessible and inexpensive – most new personal computers have a GPU card. Hence, GPUs offer an attractive opportunity for speeding up PFs and PFFs for real-time applications.

A typical GPU is a collection of multiprocessors (MPs) where each of them has several scalar processors (SPs), also referred to as cores [13]. At any given clock cycle, each SP executes the same program (one or a set of instructions) on different data by following the single instruction/program multiple data (SIMD/SPMD) models. Each SP has access to different memory levels. Fig. 4.1 gives a simplified illustration of the architecture of Nvidia GPU. This type of architecture is ideally suited to data-parallel computation since large quantities of data can be loaded into shared memory for the cores to process in parallel.

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia as a computing engine in GPUs. It allows access to the virtual instruction set and memory of a GPU's parallel computational elements. By using CUDA the GPUs can be used for computation like CPUs.

A detailed description of CUDA is given in [73]. The MP model used in CUDA is called single-instruction multiple-thread (SIMT). In SIMT, the MP maps each thread to one SP core, and each thread executes independently with its own instruction address and register state. The MP creates manages, and executes concurrent threads in hardware with no scheduling overhead. The threads are logically organized in blocks and grids. A block is a set of threads, while a grid is a set of blocks. The sizes of the blocks and grids can be programmatically controlled. To optimize an execution configuration the first parameters to choose are the grid size (number of blocks per grid) and block size (number of threads per block). As a general guideline, provided by [73], the primary concern is keeping the entire GPU busy—the number of blocks in a grid should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute, and there should be multiple active blocks per multiprocessor so that blocks which are not waiting for thread synchronization can keep the hardware busy.

CUDA devices use several memory spaces, which have different characteristics that reflect their distinct usages in CUDA applications. These memory spaces include global, local,

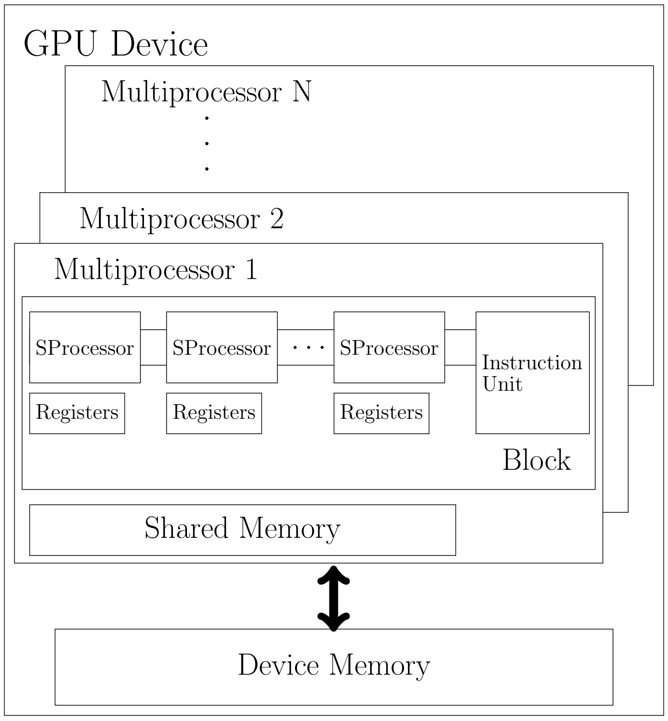


Figure 4.1: Nvidia GPU Architecture

shared, texture, constant and registers. For example, there is a 16 KB per thread limit on local memory, a total of 64 KB of constant memory, and a limit of 16 KB of shared memory, and either 8,192 or 16,384 32-bit registers per multiprocessor. Global, local, and texture memory have the greatest access latency, followed by constant memory, registers, and shared memory. Memory optimizations are key for performance. The best way to maximize bandwidth is to use as much fast-access memory and as little slow-access memory as possible.

There are many important factors for GPU performance such as memory optimizations, thread and block size. Memory reading from or writing to shared, local, or global memory takes different time cycles. When accessing local or global memory, there are 400 to 600 clock cycles more than accessing shared memory. Sometimes much of this global memory accessing time cycles can be hidden by the thread scheduler. Accessing time cycles hiding depends on the number of active threads per MP, which is implicitly determined by the resource of register and shared memory. When choosing the parameter – the number of threads per block – the primary concern is keeping the GPU busy. Thus, there should be multiple active blocks per MP so that blocks which are not waiting for a synchronize signal can keep the GPU busy.

4.2 PF & PFF GPU Implementation

As discussed in Chapter 2, standard PFs cannot be run efficiently on parallel processors due to the bottleneck caused by resampling of particles. That is, as seen in Chapter 2, using M processors to implement a PF would not result in roughly a factor of M speed-up in computation time because resampling of particles requires all of them be used jointly, thereby destroying the easy parallelization. Another example is in [45], where using hundreds of processors results in less than an order of magnitude speedup, even after the resampling method was modified specifically to address this issue. In contrast, particle flow filter does not resample particles, and hence it does not suffer from this bottleneck to parallelization, even though there is a small part of it that is still coupled.

The general idea of implementing both particle algorithms (PF and PFF) on GPU is to map (dedicate) a thread to a particle and organize the algorithms in terms of groups of particles, mapped correspondingly to GPU blocks, in order to take advantage of the GPU architecture. The key is to use the shared memory for fast particle data communication within each block and avoid/reduce communications between different blocks, as much as possible.

4.2.1 Parallel PF

We implement distributed resampling with nonproportional allocation (RNA) [10]. The idea is similar to the cluster implementation described in section 3.2.1. Some specific details are as follows, and in our papers [46, 45, 88].

Distributed RNA is a modification of the distributed resampling with proportional allocation (RPA) which is essentially based on stratified sampling [30]. The sample space is partitioned into several strata (groups) and each stratum corresponds to a processing node (PN)—mapped to a GPU block of threads in our GPU implementation. Proportional allocation among strata is used, which means that more samples are drawn from the strata with larger weights. After the weights of the strata are known, the number of particles that each stratum replicates is calculated at a head processing node (HN)—a dedicated block of threads in a GPU implementation—using residual systematic resampling, and this process is referred to as inter-resampling since it treats the PNs as single particles. Finally, resampling is performed inside the strata (at each PN, in parallel) which is referred to as intra-resampling. RPA requires a complicated scheme for particle routing. In the parallel RNA particle routing is deterministic and planned in advance by the design. The number of particles within a group after resampling is fixed and equal to the number of particles per group as opposed to the RPA where this number is random (proportional to the weight of the stratum). This introduces an extra approximation in the resampling (in statistical sense) but allows to execute resampling in parallel by each group (GPU block).

In our implementation each group of particles, as defined in RNA, is naturally mapped into a GPU block as each particle of the group corresponds to a thread of the block. Thus, in terms of the GPU architecture, the importance sampling is thread-parallel while the resampling part is only block-parallel, as shown in Table 4.1.

Table 4.1: Parallel PF Algorithm for GPU

- *Importance Sampling* (IS)
  - For $i = 1, \dots, N$ (Thread Parallel)

    Draw a sample (particle)

    $x_k^i \sim \pi \left(x_k | \, x_{k-1}^i, \, z^k \right)$

    Evaluate importance weights

    $w_k^i = w_{k-1}^i \frac{p\left(z_k | \, x_k^i \right) p\left(x_k^i | \, x_{k-1}^i \right)}{\pi\left(x_k^i | \, x_{k-1}^i, \, z^k \right)}$
  - For $i = 1, \dots, N$ (Centralized)

    Normalize importance weights:

    $w_k^i = \frac{w_k^i}{\sum_{j=1}^{N} w_k^j}$
- *Resampling* (R)

  Sample from $\left\{ x_k^j, \; w_k^j \right\}_{j=1}^{N}$ to obtain a new sample set $\left\{ x_k^i = x_k^{j_i}, \; w_k^i = \frac{1}{N} \right\}_{i=1}^{N}$
  - *Inter Resampling* (Centralized)
  - *Intra Resampling* (Block Parallel)

Other important issues arising in the GPU implementation are the cumulative summation (needed for normalization) and random number generation (needed for IS). The cumulative sum can in principle be implemented using a multipass scheme similar to that of [38], but we propose a better way in section 4.4. The CURAND library [74] provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers. It is used to create several generators at the same time. Each generator has a separate state and is independent of all other generators.

4.2.2 Parallel PFF

GPU parallelization of PFF is much easier than that of the PF. It is apparent from Table 2.2 that the prediction part and solving the ODE for each particle is independent across particles and can be executed completely in parallel by mapping a particle to a GPU thread.

For computing $\bar{x}_k$, P_k, H_k, and the flow velocity parameters $A(\lambda)$, $b(\lambda)$ there are different options because they are common for all particles. One way is to have them computed by the CPU ("externally" to the GPU) by a point estimator like EKF/UKF and sent to each particle. Another way is to collect all predicted particles in the CPU, compute sample mean $\bar{x}_k$, and based on it, compute P_k, H_k, $A(\lambda)$, $b(\lambda)$ and send the results to each particle. Involvement of the CPU imposes a communication time overhead (due to the need to access the slow memory), and does not take advantage of the GPU block architecture. That is why, as in the GPU-PF, we divide all particles in groups and map each group to a GPU block wherein each particle corresponds to a thread. A predicted estimate $\bar{x}_{k,b}$ is computed for each block b, as the sample mean of its particles, and based on it, flow velocity parameters for this block are computed. Thus, even though some extra approximation is incurred, any communication with the CPU and among different block is avoided. A high level outline is given in Table 4.2.

In accordance with the limitation of shared memory, the number of particles we include in each block is specific for each application and is given later in the description of each case study, respectively. The same block sizes we also use for the GPU-PF implementation.

Note that, except for the small block sample mean part (which is block parallel), this GPU-PFF implementation is completely (thread) parallel as opposed to the GPU-PF implementation wherein a significant part (the resampling) is only partially (block) parallel. This clearly gives an explanation to the fact that the implemented GPU-PFF is much faster than the GPU-PF for the same number of particles, as seen in the Case Study III simulation results presented in the next section.

Table 4.2: Parallel PFF Algorithm for GPU

- *Particle Prediction* (Thread Parallel)
- *Particle Update*
  - Compute $\bar{x}_{k,b}$, $P_{k,b}$, $H_{k,b}$ (Block Parallel)
  - Compute particle flow ODE (Thread Parallel)

For computing a block $\bar{x}_{k,b}$ we use the sample mean.

$P_{k,b}$, $H_{k,b}$ are computed using the EKF equations.

For numerical computation of the ith particle's ODE we use the finite difference forward Euler method:

$$\begin{aligned} x_{[0]} &= \bar{x}_k^i \\ x_{[l+1]} &= x_{[l]} + \Delta\lambda f\left(x_{[l]}, l\Delta\lambda\right) \\ x_k^i &= x_{[L]} \end{aligned} \tag{4.1}$$

where $l = 0, 1, \ldots, L-1, L > 1$ is an integer defining the discretization step $\Delta\lambda = 1/L$ of the interval [0 1].

4.3 Target Tracking Applications

4.3.1 Case Study II: Ground Multitarget Tracking using GPU

We implement and study the performance of PFs with CPU-CR (Centralized Resampling), GPU-CR, and GPU-DR (Distributed Resampling) on a computer with GPU for the Case Study II tracking application, described in section 3.2.3.

CPU-CR

CPU performs prediction (draw predicted samples and calculate importance weights) and resampling using residual systematic resampling method [30]. This algorithm does not use GPU. It is implemented just for comparison.

GPU-CR

GPU (in parallel) performs prediction and CPU performs resampling.

GPU-DR

Both IS & R are performed in parallel on the GPU. The resampling is as described in section 4.2.1. After the weights of the block are known, the number of particles that each block replicates is calculated at CPU using residual systematic resampling (inter resampling). Finally, resampling is performed inside the block in parallel. Since the current generation of GPUs has a maximal texture size the number of particles that can be resampled as a single unit is limited.

Experiment Setup

The parameters of Case Study II (section 3.2.3) that are used in this GPU implementation are as follows. The targets move in a $5000m \times 5000m$ surveillance area. They have a nearly constant velocity motion, according to the state model (3.13) with $Q = diag\{20, 0.2, 20, 0.2\}$. The initial position and velocity of each target, for each Cartesian coordinate x and y are generated randomly from the uniform distributions $\mathcal{U}(0, 5000)$ and $\mathcal{U}(-10, 10)$, respectively. The sensor scans a fixed rectangular region of 50×50 pixels, where each pixel represents a $100m \times 100m$ area on the ground plane. The sensor returns Rayleigh-distributed measurements in each pixel, depending on the number of targets that occupy the pixel according to the measurement model (3.14). The sensor sampling interval $\Delta = 1s$ and SNR $\lambda = 15$.

Scenarios with different number of targets were simulated, i.e., $T = 3$ and 20 (Fig. 4.2). The three PFs (CPU-CR, GPU-CR, GPU-DR) were implemented with different number of particles for each scenario. In order to obtain statistically significant evaluation of the performance metrics, 100 Monte Carlo runs were performed for each set of scenario and

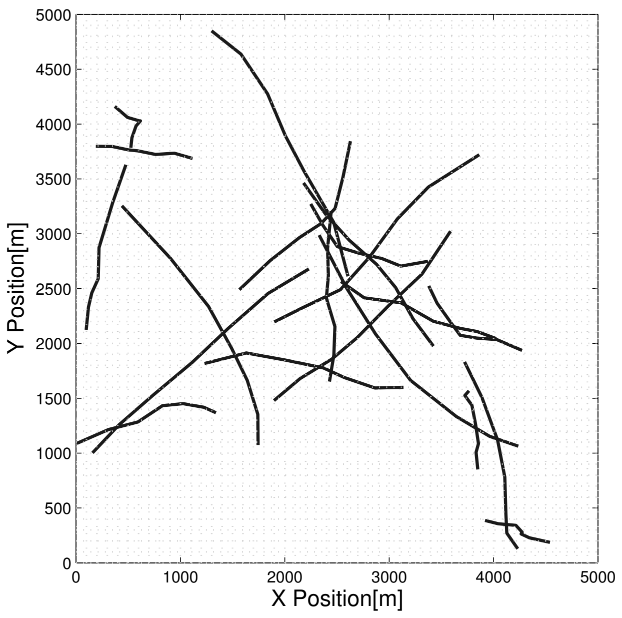


Figure 4.2: Scenario with 20 Targets.

algorithm parameters.

The hardware used in our experiments is presented in Table 4.3. Note that there are 8 parallel pipelines and the number of particles $\gg$ number of pipelines. The code for all implementations is written in C++ and compiled using Visual C++ 2008.

Results & Comparison

Due to space limitation, only the most representative results are reported in this dissertation.

First, Fig. 4.3 shows the computation times spent on different parts of the PF algorithm in the GPU-DR implementation. The resampling step incurs the highest computational cost. Quantitatively, resampling is from two to four times more costly (depending on the number of particles) than importance sampling, and about 2.5 times than generating the estimates.

Second, Fig. 4.4 shows a comparison between the times for resampling only (CR and DR) with different number of particles. Even though DR is in parallel, the improvement over

Table 4.3: Hardware Used for the Simulation

CPU	Model:	Intel Core(TM)2 Duo
	Clock Rate:	1.40GHz
	Memory:	2.0G
	Operating System:	Windows 7
GPU	Model:	NVIDIA GeForce 8400M GS
	CUDA Driver:	3.20
	Clock Rate:	0.80 GHz
	Cores:	2 (MP) x 8 (Cores/MP) = 16 (Cores)
	Global Memory:	115M bytes
	Constant Memory:	64K bytes
	Shared Memory:	16K bytes/block

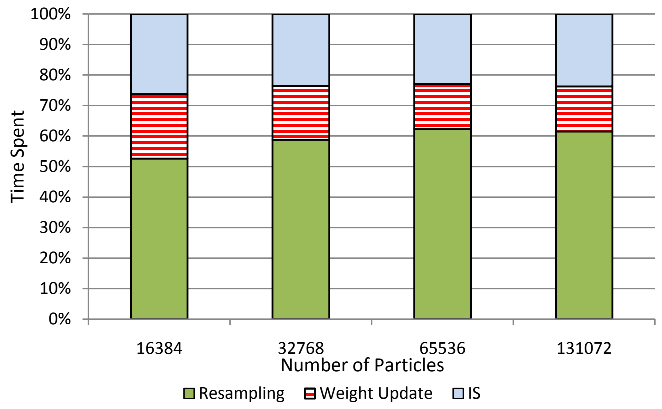


Figure 4.3: Relative Times Spent in the Different Steps Using GPU-DR.

CR is not significant because the clock rate of GPU is much lower (almost two times) than that of the CPU. Also, as the number of particles increases the efficiency of DR decreases due to the limited pipelines of GPU.

Next, Fig. 4.5 and Fig. 4.6 show the position *time-average root-mean square errors* (TARMSE) and execution times of the three PF algorithms for 3 targets, respectively. Fig. 4.7 and Fig. 4.8 are for 20 targets. Fig. 4.5 indicates that, for 3 targets, using slightly more than 60K particles (in all filters) is a reasonable choice for practical purposes. For 20 targets, Fig. 4.7 indicates that more than 130k particles are needed. These figures also illustrate that CPU-CR and GPU-CR are better than GPU-DR in terms of accuracy (for

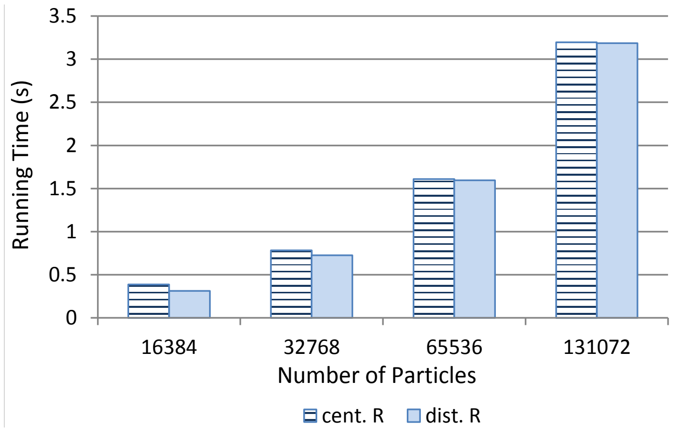


Figure 4.4: Computation Times of CR & DR.

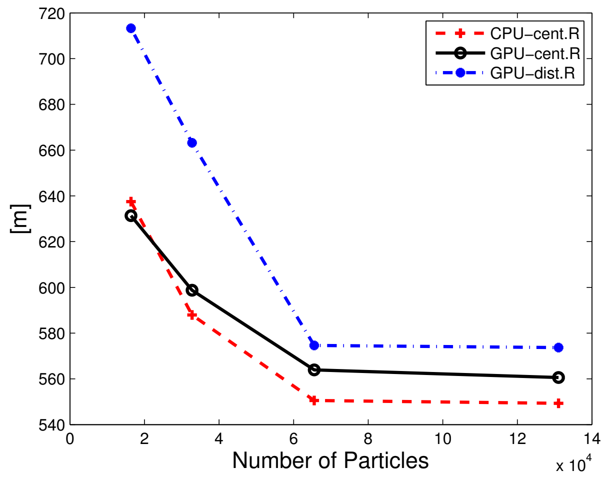


Figure 4.5: Position TARMSE; 3 Targets; 100 MC Runs.

the same number of particles). This is clear because CR has better utilization of particles than DR—the former implements the resampling exactly as opposed to the latter which is approximate.

On the other hand, Fig. 4.6 and Fig. 4.8 show the execution times for one computational cycle of the tracking filter. Now the order of performance is reversed with CPU-CR being considerably slower than both GPU-CR and GPU-DR (which are close in computation times). Quantitatively (based on all simulations with 130K particles), CPU-CR is about 30% slower than GPU-DR.

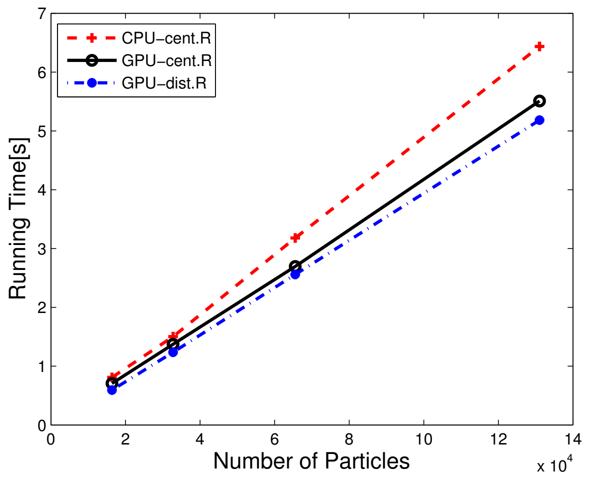


Figure 4.6: Average Execution Time; 3 Targets; 100 MC Runs.

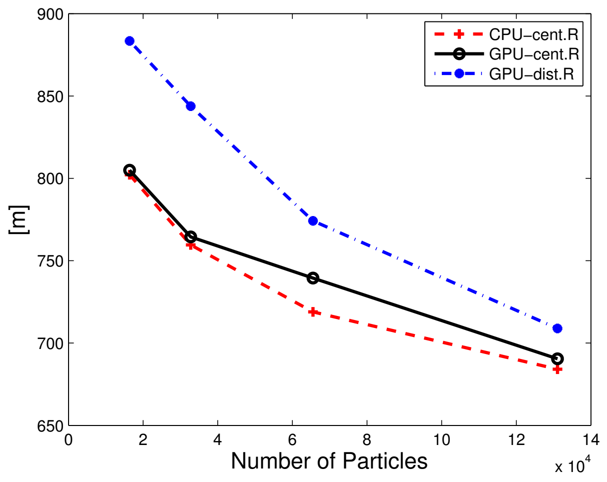


Figure 4.7: Position TARMSE; 20 Targets; 100 MC Runs.

The fully centralized algorithm (CPU-CR) is the best in terms of accuracy at a given number of particles but its computation time is worst. The fully distributed algorithm (GPU-DR) has shown the best running time but its accuracy is the worst. The partially distributed algorithm (GPU-CR), for the considered scenarios, has shown a computation time close to that of the fully distributed (GPU-DR) and accuracy not much worse than that of the fully centralized (CPU-CR). It appears that, for the considered hardware configuration, the partially distributed implementation may provide a reasonable tradeoff between filter

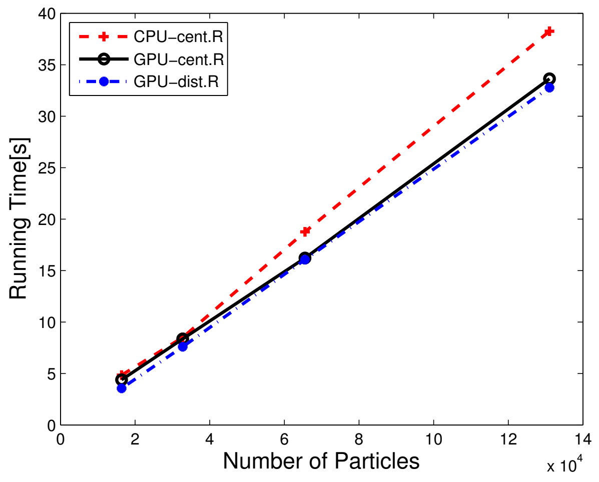


Figure 4.8: Average Execution Time; 20 Targets; 100 MC Runs.

accuracy and computation time as compared to the other two implementations.

4.3.2 Case Study III: PF vs. PFF on GPU for High Dimensional Problem

The main goal of this section is design, implementation, and performance comparison of parallel PFs and PFFs using a computer with GPU as a parallel computing environment. Four algorithms are implemented and experimented: PF-CPU, PF-GPU, PFF-CPU, PFF-GPU, respectively. Comprehensive simulation of the algorithms over high-dimensional nonlinear filtering scenarios (up to 40-dimensional state vectors) is conducted for performance evaluation, and a comparison is made based on the experimental data. The obtained quantitative experimental results provide helpful information and guidelines in a practical design [47].

The hardware used in our design and simulation (presented later) is described in Table 4.4.

The GPU PF and PFF described in section 4.2 were implemented. The particular configuration parameters of our GPU implementation are given in Table 4.5. They were determined

Table 4.4: Hardware Used for Design & Simulation

CPU	Model:	Intel(R) Core(TM) i5-3210M
	Clock Rate:	2.50GHz
	Memory:	4.0G
	Operating System:	Windows 7
GPU	Model:	NVS 5400M
	CUDA Driver:	5.0
	Clock Rate:	0.95 GHz
	Cores:	2 (MP) x 48 (Cores/MP) = 96
	Registers:	32K bytes/block
	Constant Memory:	64K bytes
	Shared Memory:	48K bytes/block

Table 4.5: GPU Blocks' Specification

Num of Particles =		1536	12288	49152	98304	196608	393216	786432	1572864
XDim = 10	Threads/Block = 256	6	48	192	384	768	1536	3072	6144
XDim = 20	Threads/Block = 128	12	96	384	768	1536	3072	6144	12288
XDim = 30	Threads/Block = 96	16	128	512	1024	2048	4096	8192	16384
XDim = 40	Threads/Block = 64	24	192	768	1536	3072	6144	12288	24576

based on the available hardware capability.

Model

We consider nonlinear filtering for the following model with cubic measurement nonlinearities[1]

$$x_{k+1} = \Phi x_k + w_k \tag{4.2}$$

$$z_k = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathrm{x}_{k,1}^3 \\ \mathrm{x}_{k,2}^3 \\ \mathrm{x}_{k,3}^3 \end{bmatrix} + v_k \tag{4.3}$$

[1] The target dynamics need not be nonlinear for the purpose of comparison because both nonlinear filters, PF and PFF, differ only in the measurement update part.

where $k = 0, 1, \ldots$ is time index, $x_k = [\mathsf{x}_{k,1}\ \mathsf{x}_{k,2}\ \ \ldots, \mathsf{x}_{k,d}]'$ is the state vector of dimension $d \geq 3$, Φ is a positive definite transition matrix that is generated randomly for each scenario in the simulation, $w_k \sim \mathcal{N}(0, 0.04^2 I_d)$ and $v_k \sim \mathcal{N}(0, 1.0^2 I_3)$ are zero-mean Gaussian white process and measurement noises, respectively, and the initial state vector $x_0 \sim \mathcal{N}(0.8, 25000 I_d)$ is randomly generated.

Four scenarios with different state dimension (i.e., $d = 10, 20, 30, 40$), each with different number of particles as specified in Table 4.5, are simulated.

Algorithms

Implemented are the following four filter algorithms: PFF-CPU, PFF-GPU, PF-CPU and PF-GPU as described earlier in section 4.2. The hardware used in our experiments is presented in Table 4.4. The code for all implementations is written in C++ and compiled using Visual C++ 2008.

Performance Measures

In order to obtain statistically significant evaluation of the performance metrics, $R = 50$ Monte Carlo (MC) runs are performed for each scenario.

The filters' estimation accuracy at each time step k is measured in terms of the average dimension-free error

$$e_k = \frac{1}{R} \sum_{i=1}^{R} e_k^{(r)}, \tag{4.4}$$

where

$$e_k^{(r)} = \frac{1}{d} \left(\hat{x}_k^{(r)} - x_k^{(r)} \right)' \left(\hat{x}_k^{(r)} - x_k^{(r)} \right) \tag{4.5}$$

and $x_k^{(r)}$ and $\hat{x}_k^{(r)}$ are the true and estimated state vectors, respectively, in MC run $r = 1, 2, \ldots, R$.

The filters' overall accuracy is measured in terms of the time-averaged error (TAE), defined as follows

$$\varepsilon_{[m,n]} = \frac{1}{n-m+1} \sum_{k=m}^{n} e_k \tag{4.6}$$

where $[m, n]$ is the time interval of averaging. TAE is sometimes used in target tracking as a single measure of "steady-state" filter accuracy. In the simulation $m = 21$, $n = 50$.

The computational performance of all four filters is measured in terms of average running time per one filter time-step. The computational performance of the parallel GPU filters is measured in terms of speedup with respect to the corresponding CPU sequential filter, i.e.,

$$Speedup = \frac{T_{CPU}}{T_{GPU}} \tag{4.7}$$

where T_{CPU} and T_{GPU} denote the average running time of the filter (PF or PFF) on CPU and GPU, respectively. The speedup characterizes the scalability of a parallel algorithm.

Results

Fig. 4.9 shows the dimension-free error plots of the filters [2] with 10 dimensional state vectors and 800K particles. It illustrates that PFF-CPU is the best in terms of accuracy (for the same number of particles), followed by PFF-GPU, PF-CPU and PF-GPU. Very similar results were obtained for different state vector dimensions (up to 40) and different number of particles. Based on all results, the GPU versions of both PFF and PF are apparently less accurate than their corresponding CPU versions. This is because the parallel GPU versions are actually approximations of the fully centralized CPU versions. For the PF this approximation is in the resampling step: it is global (uses all particles) in PF-CPU and local (uses only the particles within a block) in PF-GPU. A similar effect happens with the PFF:

[2] EKF's error plot is shown as a baseline only, and is excluded from further comparison.

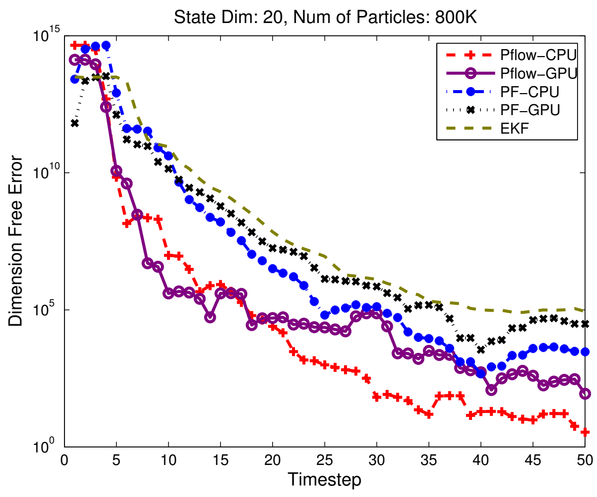


Figure 4.9: Dimension-Free Errors

the computation of $\bar{x}_k$ in PFF-CPU is global (the sample mean of all particles), while the computation of $\bar{x}_{k,b}$ in PFF-GPU is local and consequently less accurate.

More detailed overall accuracy comparison can be made based on the plots in Fig. 4.10 (left) that show the time-averaged dimension-free errors of the four filters with different dimensions of the state vector versus the number of particles. The differences in accuracy are quite significant. In particular, PFF-GPU is several orders of magnitude more accurate than both PF-GPU and PF-CPU. Of course, PFF-GPU is less accurate than PFF-CPU (for reasons explained above), but this does not seem significant, given the fact (discussed next) that the former is much faster than the latter.

The plots in Fig. 4.10 (right) show the average running times—the"prices" paid to achieve the accuracies shown in the corresponding plots of Fig. 4.10 (left). With the same number of particles, PFF-CPU is about 1.5-2 times faster than PF-CPU (depending on the number of particles), and PFF-GPU is about 4-5 times faster than PF-GPU. Table 4.6 is provided for more accurate comparison. Even though it might seem that PFF-CPU require more computation time per particle than PF-CPU, it actually appears otherwise. PFF update includes computing $A(\lambda)$, $b(\lambda)$ and solving the ODE via (4.1). Computing A, b is common for all particles and independent from the number of particles (except for $\bar{x}_k$). Therefore, for large number of particles it has insignificant contribution to the "per particle" computation time. The main portion of computation time per particle is spent on (4.1) which is a fairly simple and fast computation for small L. In the simulation $L = 10$ and the time spent on (4.1) is up to twice smaller than the time PF spends to draw a resampled particle from a large set of particles using stratified sampling. PFF with $L = 5$ and 20 was also run, but $L = 10$ was chosen as the best tradeoff between accuracy and speed. For $L = 20$ the computation times of PFF-CPU and PF-CPU are closer but the advantage of PFF in accuracy increases. The time results regarding PFF-GPU vs. PF-GPU are not surprising given the fact that PFF-GPU is almost completely (thread) parallel while the resampling of PF-GPU is only partially (block) parallel.

Fig. 4.11 illustrates the effect of the state dimension on the running time with different number of particles of both parallel filters: PF-GPU (left) and PFF-GPU (right).

Fig. 4.12 shows the speedup of PF-GPU (left) and PFF-GPU (right) with different state dimension and different number of particles. It appears that for both GPU filters the speedup most often increases with the state dimension (for the same number of particles) which supports using GPU for highly dimensional problems. On the other hand, the speedup for both GPU filters does not seem to vary very significantly with the number of particles (for the same state dimension). Finally, Fig. 4.13 compares the speedup of PF-GPU and PFF-GPU for different number of particles (for state dimension 40). PF-GPU provides a speedup of about six times with respect to PF-CPU and PFF-GPU provides a speedup of about fifteen times with respect to PF-CPU. PFF-GPU outperforms PF-GPU more than two times in terms of speedup due to its higher level of parallelization.

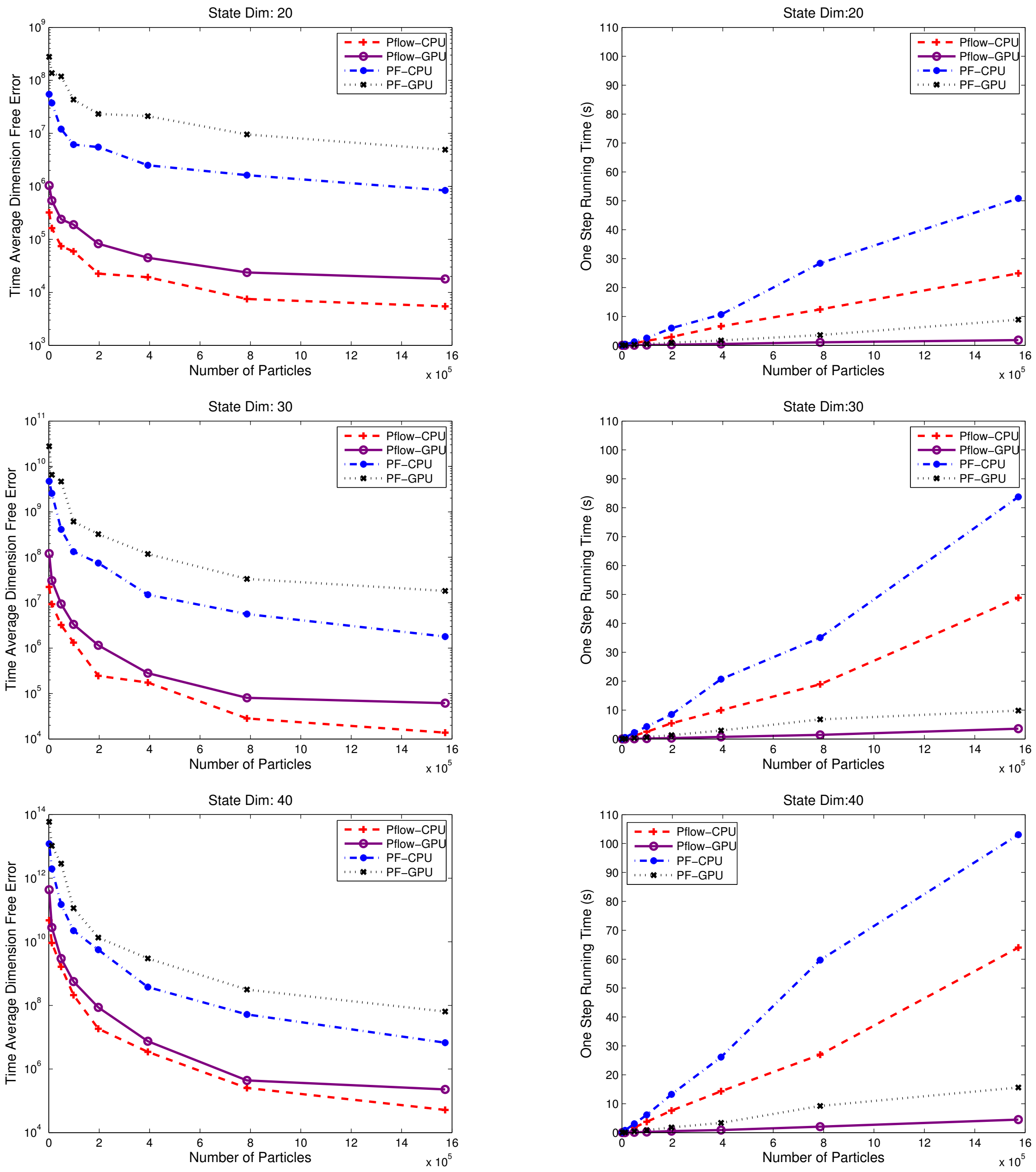
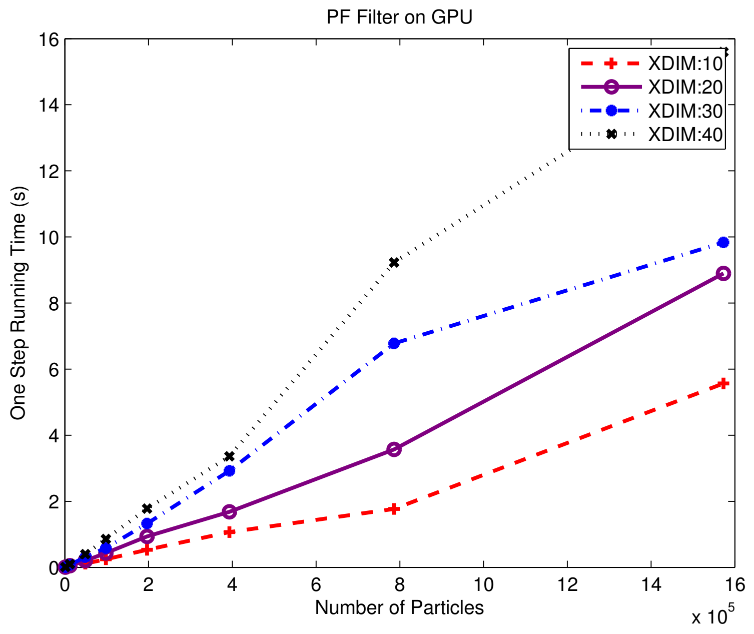


Figure 4.10: Time-Averaged Dimension-Free Error (left) & Running Time (right)



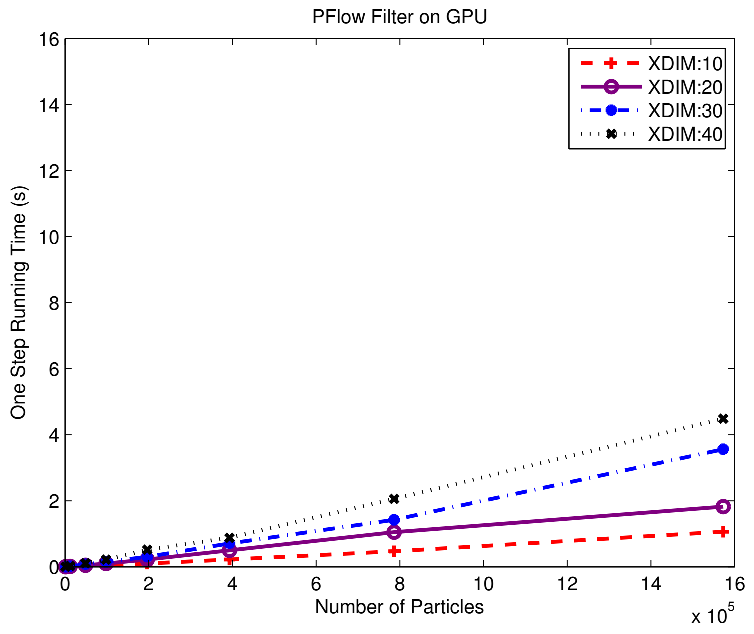
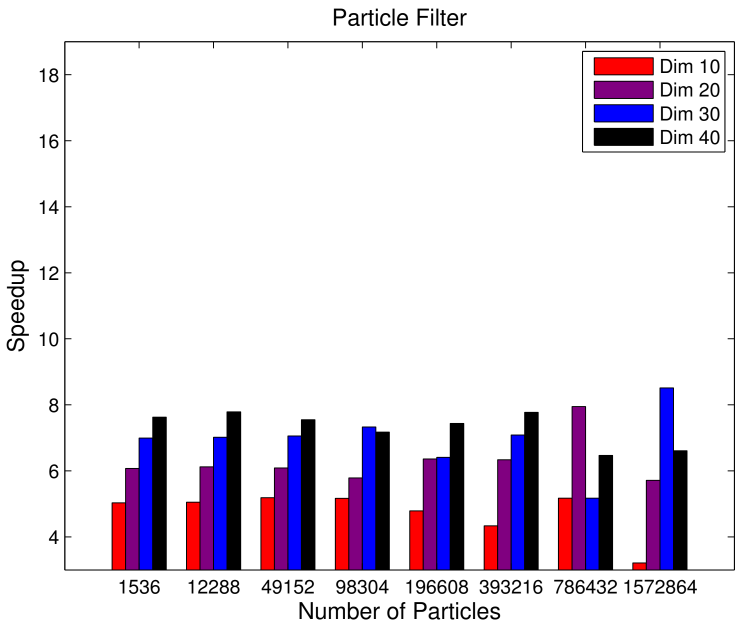


Figure 4.11: One-Step Running Time



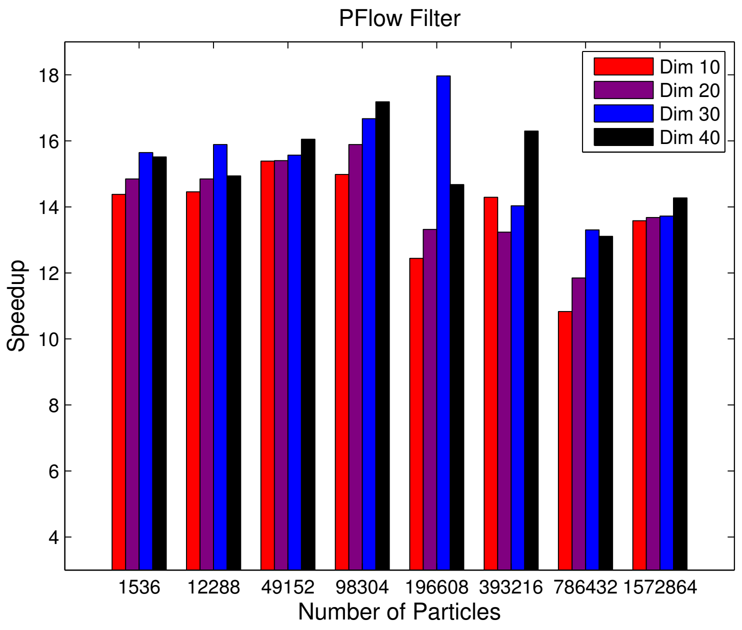


Figure 4.12: Speedup of PF & PFF

Table 4.6: Running Time (s)

Num of Particles=		1536	12288	49152	98304	196608	393216	786432	1572864
XDim= 10	PFF-CPU	0.0099	0.0792	0.3403	0.6682	1.2630	3.1733	5.0851	14.4671
	PFF-GPU	0.0007	0.0054	0.0218	0.0439	0.1000	0.2187	0.4625	1.0493
	PF-CPU	0.0172	0.1418	0.5856	1.2602	2.5559	4.6397	9.1571	17.8682
	PF-GPU	0.0034	0.0281	0.1129	0.2438	0.5337	1.0696	1.7700	5.5668
XDim= 20	PFF-CPU	0.0218	0.1760	0.7338	1.5952	2.9421	6.6210	12.4409	24.9088
	PFF-GPU	0.0014	0.0117	0.0469	0.0989	0.2175	0.4928	1.0343	1.7933
	PF-CPU	0.0382	0.3139	1.2319	2.5249	5.9612	10.6548	28.3821	50.8195
	PF-GPU	0.0063	0.0513	0.2023	0.4362	0.9371	1.6812	3.5717	8.8979
XDim= 30	PFF-CPU	0.0359	0.2925	1.1410	2.4776	5.4436	9.9433	18.9128	48.8515
	PFF-GPU	0.0023	0.0181	0.0722	0.1464	0.2984	0.6978	1.4003	3.5066
	PF-CPU	0.0643	0.5213	2.2102	4.2976	8.4932	20.7041	35.0581	83.7303
	PF-GPU	0.0092	0.0743	0.3132	0.5864	1.3253	2.9221	6.7780	9.8334
XDim= 40	PFF-CPU	0.0520	0.4127	1.7799	3.7476	7.6032	14.2963	26.9597	64.0159
	PFF-GPU	0.0033	0.0272	0.1092	0.2148	0.5102	0.8640	2.0259	4.4173
	PF-CPU	0.0947	0.7756	3.0457	6.1521	13.2333	26.1284	59.6716	103.0515
	PF-GPU	0.0124	0.0996	0.4036	0.8578	1.7792	3.3625	9.2275	15.5961

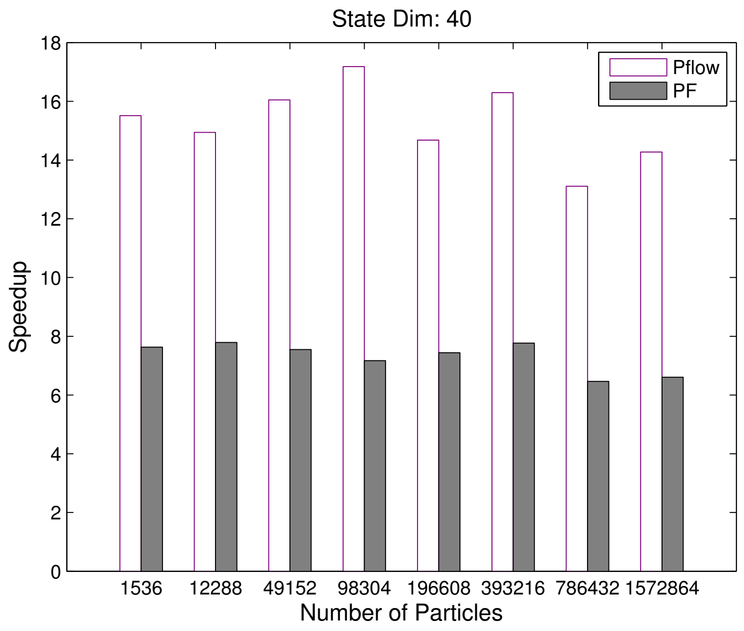


Figure 4.13: Speedup of PF *vs.* PFF

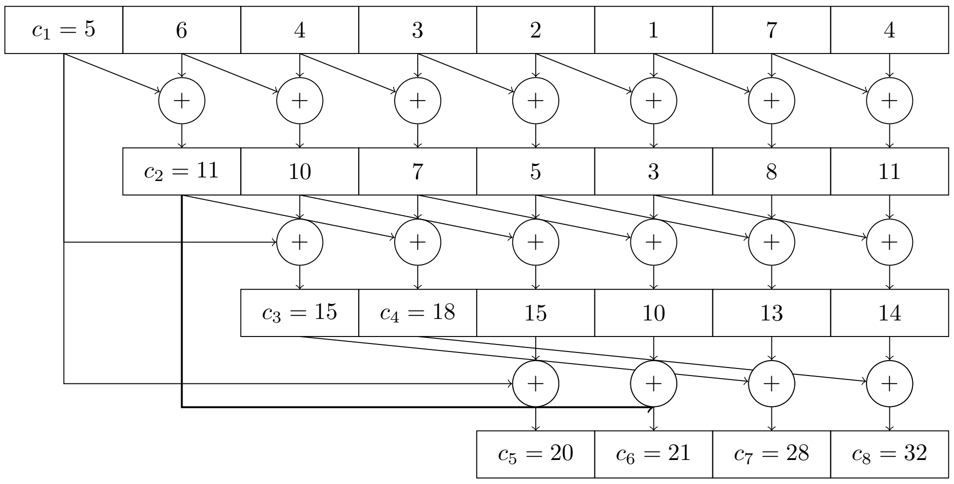


Figure 4.14: Parallel Cumulative Sum

4.4 Improved Algorithm for GPU PF

Based on the following reasons, we designed an improved algorithm implementation for GPU.

- Due to the shared memory, each thread can access all particles' information easily, so replicating particles could be parallelized in all threads.
- Cumulative weight sum, which is key for resampling, can be also executed in parallel in contrast with the generic PPFs (section 4.2.1) where it is done on a single processor.

When implementing cumulative weight sum on a single processor, the time complexity is $O(n)$ addition operations for an array of length n. This is the minimum number of additions required. A parallel algorithm is presented by [39]. Fig. 4.14 shows the operation. The time complexity is $O(\log n)$ if enough processors are available. This is fine for small arrays. In our application, particle filter algorithms use large number of particles, so large arrays are used. We updated the parallel particle filter algorithm by applying a modified parallel cumulative weighted sum algorithm, as follows (See Fig. 4.15 for an illustration).

Step 1. Implement importance sampling and weight computation (thread parallel).

Step 2. Compute cumulative weights: $c_k = w_1 + w_1 + \cdots + w_k$ (block parallel).

- Compute cumulative weight of each block.
- Write the total weight of each block to another temp block.
- Parallel compute the cumulative weight of the template block, then add to all elements in their respective blocks to get $c_1, c_2, \ldots, c_N$.

Step 3. Resample to get new particles according to the cumulative weights (thread parallel).

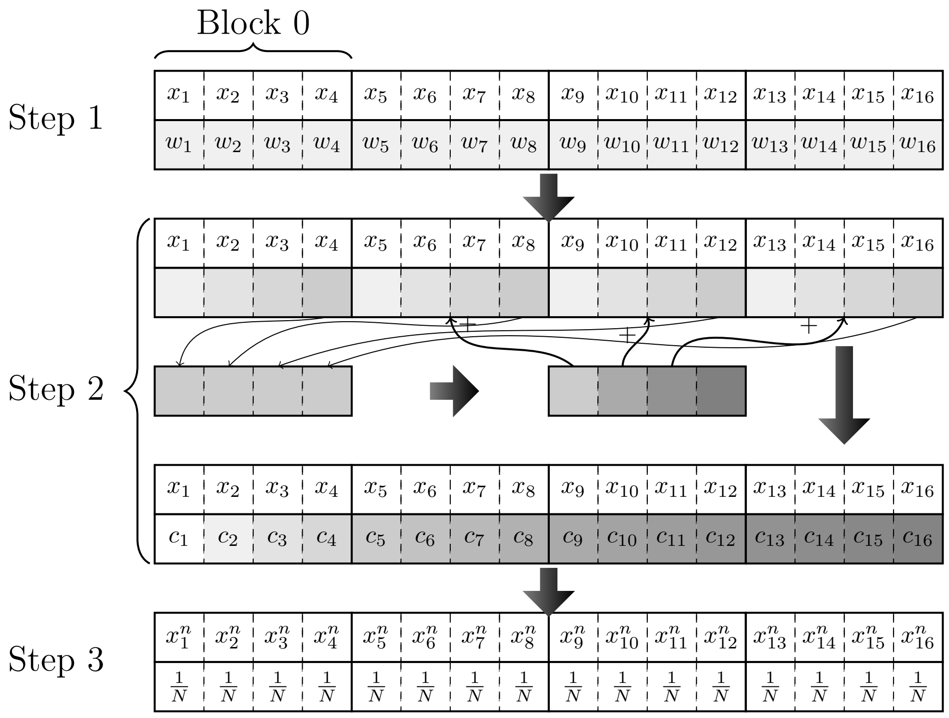


Figure 4.15: Illustration of the Improved GPU PF

Note that Step 3 can be done thread parallel because all thread can access all cumulative sums concurrently.

The updated parallel particle filter algorithm is shown in Table 4.7.

Table 4.7: Improved Parallel PF Algorithm for GPU

- Step 1: *Importance Sampling* (IS)
  - For $i = 1, \ldots, N$ (Thread Parallel)

    Draw a sample (particle)

    $x_k^i \sim \pi\left(\, x_k \middle|\, x_{k-1}^i,\, z^k\right)$

    Evaluate importance weights

    $w_k^i = w_{k-1}^i \frac{p(\, z_k|\, x_k^i)p(\, x_k^i|\, x_{k-1}^i)}{\pi(\, x_k^i|\, x_{k-1}^i,\, z^k)}$

- Step 2: *Cumulative Particle Weights* (Block Parallel)
  - Parallel execute thread cumulative weight for each block
  - Execute block cumulative weight
  - Parallel get all cumulative weight

- Step 3: *Resampling* (R)
  - For $i = 1, \ldots, N$ (Thread Parallel)

    Sample from $\left\{x_k^j\right\}_{j=1}^N$ with $\{c_j\}_{j=1}^N$ to obtain a new sample set $\left\{x_k^i = x_k^{j_i},\ w_k^i = \frac{1}{N}\right\}_{i=1}^N$

After optimizing the particle filter algorithm, the main bottlenecks left are global memory access. To better cover the global memory access latency and improve overall efficiency, we should let each thread do more computation. So we could let each thread processes two or four particles instead of one particle. Each thread performs a sequential access of four particles and stores them in registers. This method is more than twice as fast as the code which only processes one particle each thread.

4.4.1 Simulations – Case Study II

To evaluate and compare the novel GPU PF we used the Case Study II (see section 4.3.1). The hardware used in our design and simulation is described in Table 4.4. Our efforts to create a new parallel particle filter algorithm have paid off. Fig 4.16 shows resampling computation time of centralized resampling (cent.R), distributed resampling with RNA (dist.R

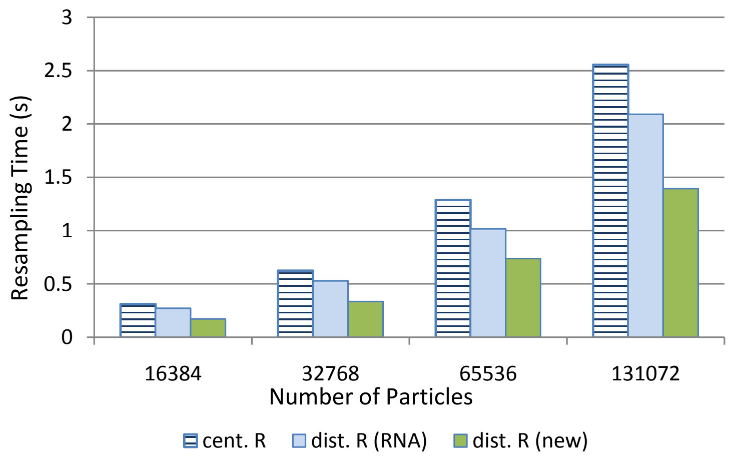


Figure 4.16: Computation Times of CR & DR

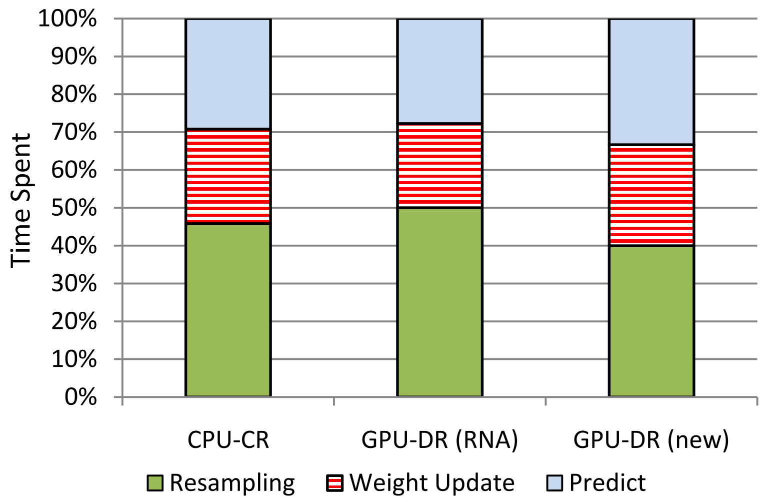


Figure 4.17: Relative Times Spent in the Different Steps Using GPU-DR

(RNA)) and our updated algorithm (dist.R (new)). It is seen that for different number of particles our updated algorithm has better performance than that of the generic algorithms. The new algorithm is up to 1.5 times faster than dist.R (RNA). Fig. 4.17 shows relative times spent in the different steps of three PF algorithms. The resampling step of dist.R (new) is almost the same computational cost with the step of importance sampling. It is a significant improvement of the resampling step as compared with cent.R and dist.R (RNA).

4.4.2 Case Study IV: UAV-MultiSensor Target Tracking using GPU PF & PFF

Scenario

The scenario is illustrated in Fig. 4.18.

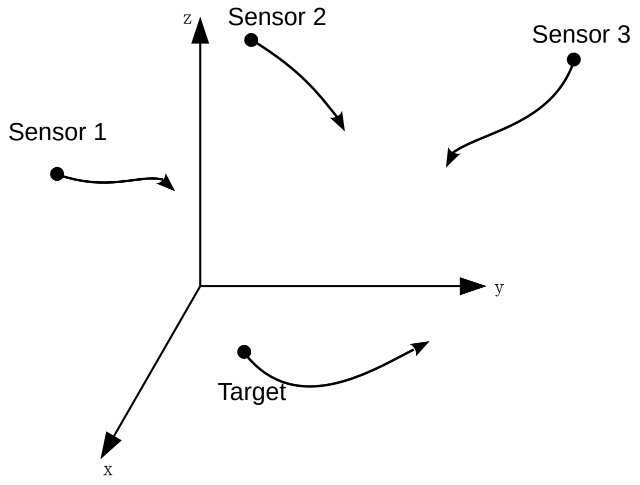


Figure 4.18: Scenario of UAV-Multisensor Tracking

State Model

The target motion state-space model is given by

$$\mathbf{x}_{k+1} = F\mathbf{x}_k + \mathbf{w}_k \tag{4.8}$$

where $\mathbf{x} = (x, \dot{x}, y, \dot{y}, z, \dot{z})'$ is the state vector,

$$F = \begin{bmatrix} 1 & T & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & T & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & T \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{4.9}$$

$\mathbf{w}_k$ is a discrete-time white noise sequence, and T is the sampling interval.

Multi-Sensor Measurement Model

The target measurements are modeled by

$$\mathbf{z}_k = \mathbf{h}(\mathbf{x}_k) + \mathbf{v}_k \tag{4.10}$$

where the target state $\mathbf{x}$ and process noise are in the Cartesian coordinates, but measurement $\mathbf{z}$ and its additive noise $\mathbf{v}$ are in the sensor coordinates. Let (x, y, z) be the true position of the target in the Cartesian coordinates. For the case of spherical measurements, we have $\mathbf{z} = (r, b, e)'$ and $\mathbf{h}(\mathbf{x}) = [r, b, e]' = [h_r, h_b, h_e]'$ with

$$h_r = \sqrt{x^2 + y^2 + z^2} \tag{4.11}$$

$$h_b = \tan^{-1} \frac{y}{x} \tag{4.12}$$

$$h_e = \tan^{-1} \frac{z}{\sqrt{x^2 + y^2}} \tag{4.13}$$

In order to use the particle flow method the model (4.10) needs to be linearized. The most widely used technique for linearizing a nonlinear measurement model in the form of

(4.10) is to expand the measurement function $\mathbf{h}(\mathbf{x})$ at the predicted state $\mathbf{x}$ and ignore all nonlinear terms:

$$\mathbf{h}(\mathbf{x}) \approx \mathbf{h}(\bar{\mathbf{x}}) + \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right|_{\mathbf{x}=\bar{\mathbf{x}}} (\mathbf{x} - \bar{\mathbf{x}}) \tag{4.14}$$

This amounts to approximating the nonlinear model (4.10) by the linear model

$$\mathbf{z} = H(\bar{\mathbf{x}})\mathbf{x} + \mathbf{g}(\mathbf{x}) + \mathbf{v} \tag{4.15}$$

where $H(\mathbf{x}) = \frac{\partial \mathbf{h}}{\partial \mathbf{x}}\big|_{\mathbf{x}=\bar{\mathbf{x}}}$ is the Jacobian of $\mathbf{h}(\mathbf{x})$

$$H(\mathbf{x}) = \begin{bmatrix} \frac{x}{d} & 0 & \frac{y}{d} & 0 & \frac{z}{d} & 0 \\ -\frac{y}{x^2+y^2} & 0 & \frac{x}{x^2+y^2} & 0 & 0 & 0 \\ -\frac{xz}{d^2\sqrt{x^2+y^2}} & 0 & -\frac{yz}{d^2\sqrt{x^2+y^2}} & 0 & \frac{\sqrt{x^2+y^2}}{d^2} & 0 \end{bmatrix}_{\mathbf{x}=\bar{\mathbf{x}}} \tag{4.16}$$

and $d = \sqrt{x^2+y^2+z^2}$, $\mathbf{g}(\bar{\mathbf{x}}) = \mathbf{h}(\bar{\mathbf{x}}) - H(\bar{\mathbf{x}})\bar{\mathbf{x}}$.

For 3 sensors (each sensor know its location $\mathbf{p}_i, i = 1, 2, 3$), the linearized measurement model is

$$\begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \mathbf{z}_3 \end{bmatrix} = \begin{bmatrix} H(\bar{\mathbf{x}} - \mathbf{p}_1) \\ H(\bar{\mathbf{x}} - \mathbf{p}_2) \\ H(\bar{\mathbf{x}} - \mathbf{p}_3) \end{bmatrix} \mathbf{x} + \begin{bmatrix} \mathbf{g}(\bar{\mathbf{x}} - \mathbf{p}_1) \\ \mathbf{g}(\bar{\mathbf{x}} - \mathbf{p}_2) \\ \mathbf{g}(\bar{\mathbf{x}} - \mathbf{p}_3) \end{bmatrix} + \begin{bmatrix} \mathbf{v}_1 \\ \mathbf{v}_2 \\ \mathbf{v}_3 \end{bmatrix} \tag{4.17}$$

where $\bar{x}$ is the state prediction.

Parameters

The noise is $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q}), \mathbf{v} \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$. where $\mathbf{Q} = diag([30, 0.8, 20, 0.6, 25, 0.7]), \mathbf{R} = diag([45, 0.001, 0.001, 45, 0.001, 0.001, 45, 0.001, 0.001])$.

Initial state value is $\mathbf{x}_0 = [450, 35, 850, 35, 1350, 35]'$.

Algorithms

Four algorithms are implemented:

PF-CPU	PF-GPU
PFF-CPU	PFF-GPU

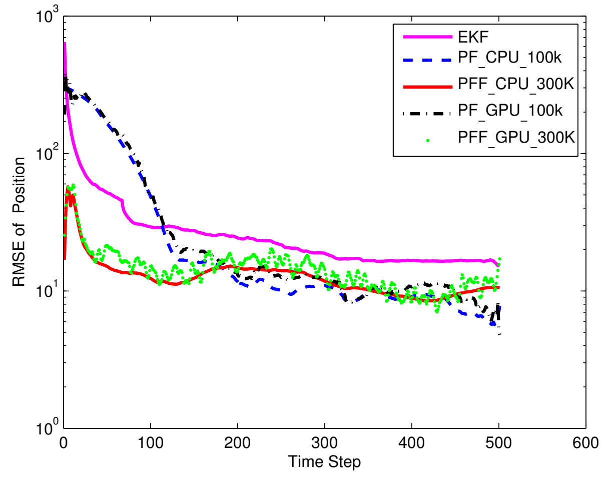
where CPU, GPU denote whether the algorithm is executed on CPU or GPU, respectively.

Results

Fig. 4.19 shows the position RMSE (top) and the execution times of one step versus the number of particles used (bottom). It indicates that for the PF algorithms using slightly more than 100K particles is a reasonable choice for practical purposes, and for the PFF using more than 300K particles is a reasonable choice. Fig. 4.19 also illustrates that PF and PFF are better than EKF (given here just for comparison) in terms of accuracy. Fig. 4.19 shows that the running time of the algorithms using GPU is much shorter than their corresponding algorithms using CPU. It also shows that the running time of PFF algorithms is shorter than PF algorithms using the same number of particles.

Fig. 4.20 shows the filters errors vs. running time as functions of the number of particles. It is seen that the proposed PF-GPU is faster than all others for the same level of accuracy. The next is PFF-GPU, and PF-CPU and PFF-CPU come after that.

Based on all results, it was observed that PF-GPU has a quite good scalability and efficiency in this case. It becomes clear from the results that within the GPU algorithms PF-GPU can help reduce considerably the computation time and keep the best accuracy. It should be reminded that PF-GPU stands for implementation of the improved version, proposed in section 4.4.



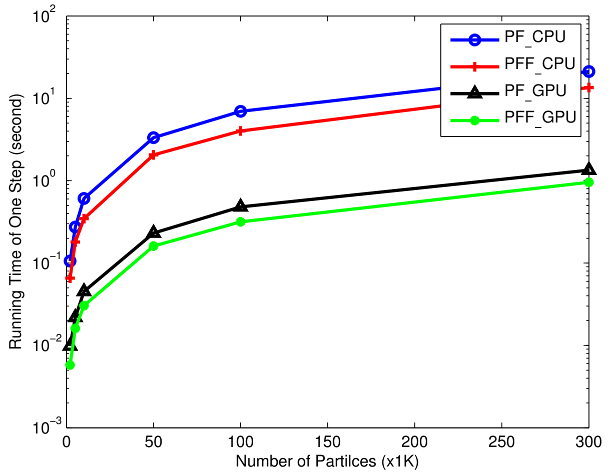


Figure 4.19: RMSE & Running Time

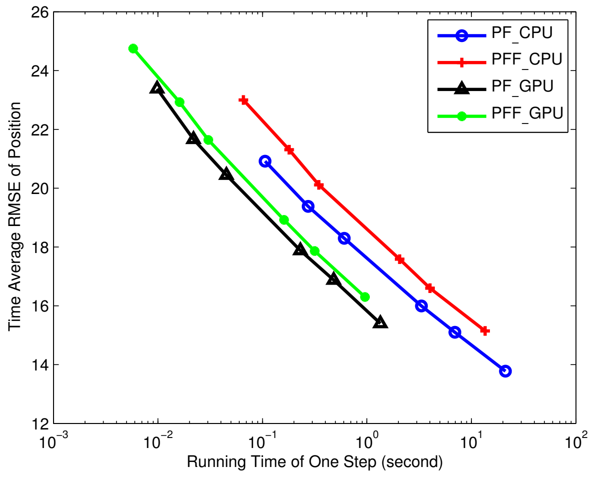


Figure 4.20: Time Average RMSE vs. Running Time

4.5 Summary

Since GPU has a special architecture, the parallel algorithms are different from the algorithms for computer cluster. Thus we redesigned the algorithms based on the group resampling idea. And we also proposed a new PF-GPU algorithm which is very efficient computationally and accurate for target tracking applications, as shown by the simulation.

Based on all simulation results:

- Using GPU can significantly accelerate particle filters and particle flow filters through parallelization of the computational algorithms at a tolerable loss of accuracy (as compared to the centralized algorithms using just the CPU).
- For the Case Study III, the parallel particle flow filter implementation is significantly superior to the generic parallel particle filter implementation in both estimation accuracy and computational efficiency—about six times for the implemented parallel particle filter and fifteen times for the particle flow filter.
- For the Case Study IV, using the newly proposed PF-GPU implementation is significantly superior to the generic PPF and to the PFF implementations in both estimation accuracy and computational efficiency.

Chapter 5

Implementation of Parallel Filtering Algorithms on FPGA

Field programmable gate arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnections [90]. An FPGA can be used to solve problems that are computable. The advantage is that it is sometimes much faster for some applications due to its parallel architecture.

To simplify the design of FPGAs, there exist libraries of predefined circuits which are called intellectual property (IP) cores. IP cores are reusable circuits.

Implementing both particle algorithms (PF and PFF) on FPGA requires that all particles be mapped into the multiple processing elements of the FPGA architecture. A processing elements (PE) compute the generation of a particle and its weight evaluation. Central unit (CU) computes global estimators and coordinates the PEs. In order to take advantage of the FPGA architecture, the key is to use the shared memory for fast particle data communication within each PE and avoid/reduce communications between different PEs, as much as possible. The particular configuration architecture [62] of our FPGA implementation is

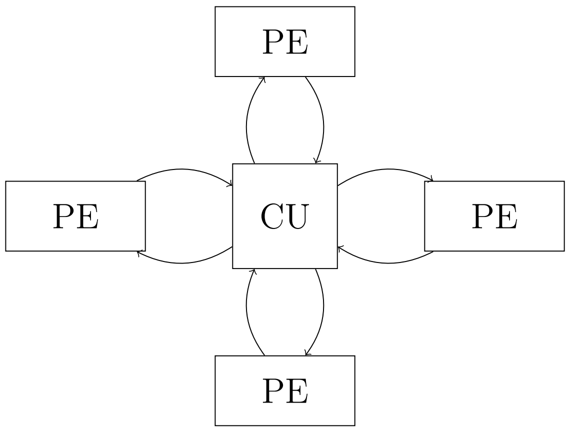


Figure 5.1: FPGA Architecture with four PEs

shown in Fig. 5.1. It consists of four PEs and one CU. Each PE communicates with the CU, but there is no communication among PEs. Fig. 5.1 also shows the data links between the PEs and the CU.

5.1 PF & PFF Algorithms for FPGA

Due to limited size and resources of the FPGA, it is not possible to calculate each particle in parallel, but sequential calculation using pipeline is a better way. The advantage of this solution is that all data go through the same calculation process and no additional FPGA resources are needed when the number of particles is increased. The PEs perform the major filter computational workload (particle generation, weight evaluation and so on), and the CU performs global computations and coordinates the PEs activities.

Other important issue arising in the FPGA implementation for both filters is how to generate random numbers. In our implementation, the random numbers are pre-stored in a ROM table. They are not generated by the hardware to save FPGA resources. The ROM table is implemented as a large buffer from the chosen probability distribution. Each use of the random number shifts the position point of the buffer [76].

5.1.1 FPGA Implementation of the Particle Filter

There is a number of publications that concern FPGA particle filters [8, 10]. In our implementation we followed the RNA method proposed in [10].

Importance sampling, weight calculation and a part of resampling are pipelined. If resampling is implemented in the CU, all the particle weights have to be transferred to CU, resulting in a large communication cost. We used the distributed RNA method [10] which can signicantly reduce data communication cost. Distributed RNA is based on stratified sampling. The sample space is partitioned into several groups and each groups corresponds to a PE. Nonproportional allocation among group is used. After the weights of the group are known, the number of particles that each group replicates is calculated at CU—using residual systematic resampling. Finally, resampling is performed inside the group (at each PE, in parallel). Each group of particles, as defined in RNA, is mapped into a PE. Thus, in terms of the FPGA architecture, the importance sampling is PE-parallel while the resampling part is partially parallel. The details of the algorithm is Table 5.1.

5.1.2 FPGA Implementation of the Particle Flow Filter

FPGA parallelization of PFF is much easier than that of the PF [89]. It is apparent that the prediction part and solving the ODE for each particle is independent across particles and can be executed completely in parallel by mapping particles to a PE. For computing $\bar{x}_k$, P_k, H_k, and the flow velocity parameters $A(\lambda)$, $b(\lambda)$ there are different options because they are common for all particles. One way is to have them computed by the CU ("externally" to the FPGA) by a point estimator like EKF/UKF and sent to each PE. Another way is to collect all predicted particles in the CU, compute sample mean $\bar{x}_k$, and based on it, compute P_k, H_k, $A(\lambda)$, $b(\lambda)$ and send the results to each PE. Involvement of the CU imposes a communication

Table 5.1: Parallel PF Algorithm for FPGA

- *Importance Sampling* (IS)
  - For $i = 1, \ldots, N$ (PE Parallel)

    Draw a sample (particle)

    $x_k^i \sim p\left(x_k^i \middle| x_{k-1}^i \right)$

    Evaluate importance weights

    $w_k^i = w_{k-1}^i p\left(z_k \middle| x_k^i \right)$
  - For $i = 1, \ldots, N$ (CU)

    Normalize importance weights:

    $w_k^i = \frac{w_k^i}{\sum_{j=1}^N w_k^j}$
- *Resampling* (R) (CU)

  Sample from $\left\{ x_k^j,\ w_k^j \right\}_{j=1}^N$ to obtain a new sample set $\left\{ x_k^i = x_k^{j_i},\ w_k^i = \frac{1}{N} \right\}_{i=1}^N$

  Compute estimate $\bar{x}_k$ (CU)

time overhead. That is why, as in the FPGA-PF, we divide all particles in groups and map each group to a PE. A predicted estimate $\bar{x}_{k,g}$ is computed for each PE, as the sample mean of its particles, and based on it, flow velocity parameters for this PE are computed. Thus, even though some extra approximation is incurred, any communication with the CU and among different PE is avoided. A high level outline is as follows:

- *Particle Prediction* (PE Parallel)
- *Particle Update* (PE Parallel)
  - Compute $\bar{x}_{k,g}$
  - Compute particle flow ODE

For computing $\bar{x}_{k,g}$ we use the sample mean.

$P_{k,g}$, $H_{k,g}$ are computed using the EKF equations.

For numerical computation of the ith particle's ODE we use the finite difference forward Euler method:

$$\begin{aligned} x_{[0]} &= \bar{x}_k^i \\ x_{[l+1]} &= x_{[l]} + \Delta\lambda f\left(x_{[l]}, l\Delta\lambda\right),\ l = 0, 1, \ldots, L-1 \\ x_k^i &= x_{[L]} \end{aligned} \tag{5.1}$$

where $L > 1$ is an integer defining the discretization step $\Delta\lambda = 1/L$ of the interval [0 1].

In accordance with the limitation of resource, the number of particles we include in each PE is given 250, 500, 750 and 1000 for each experiment, respectively. The same particle numbers we also use for the FPGA-PF implementation.

Note that, except for the PE sample mean part (which is PE parallel), this FPGA-PFF implementation is completely (PE) parallel as opposed to the FPGA-PF implementation wherein a significant part (the resampling) is only partially parallel. This clearly gives an explanation to the fact that the implemented FPGA-PFF is much faster than the FPGA-PF for the same number of particles, as seen in the simulation results presented in the next section.

5.2 Simulation

Model

We use an example of a one-dimensional non-linear system (typically studied in the context of stochastic systems) [35]. The model is as follows

$$x_{k+1} = 0.5x_k + w_k \tag{5.2}$$

$$z_k = \frac{1}{20}x_k^2 + v_k \tag{5.3}$$

where w_k and v_k are zero mean, Gaussian random variables with variances $\sigma_w^2 = 10$ and $\sigma_v^2 = 1$, respectively.

Linearizing the nonlinear measurement model needed for the flow filter, we get

$$z_k = H(\bar{x}_k)x_k + g(\bar{x}_k) + v_k \tag{5.4}$$

where $H(x_k) = \frac{x_k}{10}, g(x_k) = -\frac{x_k^2}{20}$.

Hardware Used for Simulation

Implemented are the following two filter algorithms: FPGA-PF and FPGA-PFF described earlier.

The Virtex-5 family provides powerful features in the FPGA market. Using the second generation ASMBL (Advanced Silicon Modular Block) column-based architecture, the Virtex-5 family contains five distinct platforms (sub-families), the most choice offered by any FPGA family. Each platform contains a different ratio of features to address the needs of a wide variety of advanced logic designs. XC5VFX30T is based on a 130 nm process. Our FPGA code was developed in VHDL under Xilinx ISE 12.1i.

Performance Measures

In order to obtain statistically significant evaluation of the performance metrics, $R = 50$ Monte Carlo (MC) runs are performed for each algorithm.

The filters' estimation accuracy is measured in terms of TARMSE. The computational performance of all two filters is measured in terms of average running time per one filter time-step.

Results

We used a $P = 4$ PE parallel architecture for numerical simulations, where each PE processes 250, 500, 750 and 1000 particles for each experiment, respectively.

Fig. 5.2 shows the TARMSE performance versus number of particles. It illustrates that FPGA-PF is better in terms of accuracy (for the same number of particles), than FPGA-PFF. The TARMSE is higher than the Matlab generated numerical results because of the 14 bits fixed-point FPGA implementation. Very similar results were obtained for other different numbers of particles.

The plots in Fig. 5.3 show the average processing periods of the FPGA implementations. With the same number of particles, FPGA-PFF takes about 1.1-1.2 times less cycles than FPGA-PF (depending on the number of particles). There is a simple explanation: FPGA-PFF update includes computing $A(\lambda)$, $b(\lambda)$ and solving the ODE via (5.1). Computing A, b is common for all particles and independent from the number of particles (except for $\bar{x}_k$). Therefore, for large number of particles it has insignificant contribution to the "per particle" computation time. The main portion of computation time per particle is spent on (5.1) which is a fairly simple and fast computation for small L. In the simulation $L = 10$ and the time spent on (5.1) is up to twice smaller than the time PF spends to draw a resampled

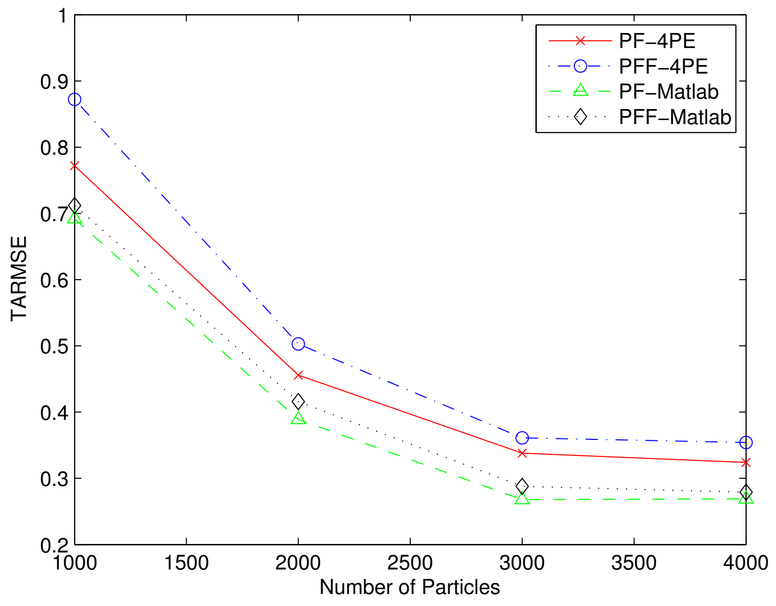


Figure 5.2: FPGA TARMSE

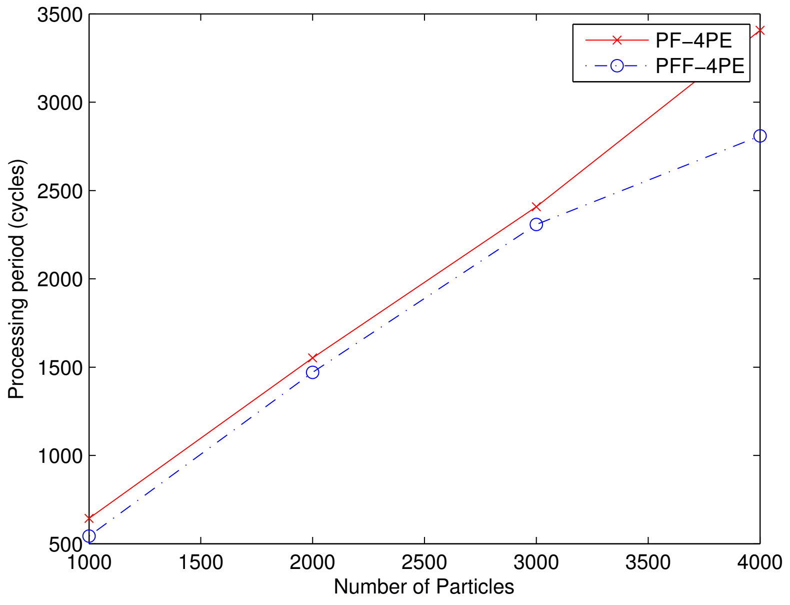


Figure 5.3: FPGA Processing Period

particle from a large set of particles using stratified sampling. PFF with $L = 5$ and 20 was also run, but $L = 10$ was chosen as the best tradeoff between accuracy and speed.

5.3 Summary

Overall, the simulation results show that:

- Using FPGA can speedup particle and particle flow filters through parallelization of the computational algorithms but at a certain loss of accuracy.
- For the considered example, the parallel particle flow filter implementation is superior to the parallel particle filter implementation in computational efficiency. On the other hand, the parallel particle filter is better than the parallel particle flow filter in estimation accuracy.

Chapter 6

Conclusions and Future Work

This chapter summarizes the principle ideas and contributions of this dissertation. It gives a summary of the dissertation and outlines directions for future research.

Particle and particle flow filters are among the latest innovations in the nonlinear filtering that are applied in many practical problems including target tracking, navigation systems and robotics. It has been shown that they can outperform traditional filters in complex practical scenarios, however particle and particle flow filters are computationally very intensive which is their main drawback in practice.

This dissertation has developed, designed, and built efficient parallel algorithms for particle and particle flow filters under different hardware architectures (Computer Cluster, GPU and FPGA), and brought them closer to practical applications. The issues tackled include reduction of computational complexity, improving scalability of parallel implementation and reducing memory requirements. This work has resulted in the development and implementation of parallel algorithms for particle and particle flow filtering. The computational performance in comparison with the implementation using traditional algorithms has been significantly improved.

The most notable contributions of this dissertation are as follows.

- We proposed an improved PF algorithm which is more suitable for computer cluster and applied it to realistic target tracking problems — space object tracking and ground multitarget tracking using image sensor. The new algorithm has shown quite significant advantage in terms of accuracy and computational performances as compared to other parallel PF algorithms.

- Based on GPU architecture, we proposed an improved parallel algorithm implementation which is computationally efficient and accurate for target tracking applications. The new algorithm reduces the complexity of realization. It has been applied and tested for ground multitarget tracking, UAV-multisensor tracking, and a highly dimensional nonlinear density problem. Our proposed method is significantly superior to the general filters implementation in both estimation accuracy and computational efficiency.

- Designed particle flow filter for FPGA. It is the first implementation on FPGA in this area. We also analyzed its performance on FPGA architecture, in terms of computational complexity and potential throughput.

The fact that particle and particle flow filters outperform traditional filtering methods in many complex practical scenarios, coupled with the challenges related to decreasing their computational complexity and improving real-time performance, makes this work worthwhile.

Future work can be extended. Since the main goal of this dissertation is high speed implementation of particle filters and particle flow filters, we used several specific architectures. However, these architectures are not universal. Future areas for research include an investigation of other parallel architectures and exploiting their specific features. Another research direction is to expand the application area with other types of practical problems that involve high fidelity and fast nonlinear filters for density estimation.

Bibliography

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, pages 483–485. ACM, 1967.

[2] Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for on-line non-linear/non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50:174–188, 2002.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[4] D.A. Bader and R. Pennington. Cluster computing: applications. *The International Journal of High Performance Computing*, 15(2):181–185, May 2001.

[5] Ying Bao, Junfeng Chen, Zhiguo Shi, and Kangsheng Chen. New real-time resampling algorithm for particle filters. In *International Conference on Wireless Communications & Signal Processing*, pages 1–5. IEEE, 2009.

[6] Anwer S Bashi, Vesselin P Jilkov, X Rong Li, and Huimin Chen. Distributed implementations of particle filters. In *Proceedings of the 6th International Conference on Information Fusion*, pages 1164–1171, 2003.

[7] Y. Boers, E. Sviestins, and H. Driessen. Mixed labelling in multitarget particle filtering. *IEEE Transactions on Aerospace and Electronic Systems*, 46(2):792–802, April 2010.

[8] Miodrag Bolic. *Architectures for efficient implementation of particle filters*. PhD thesis, Stony Brook University, 2004.

[9] Miodrag Bolic, Petar M Djuric, and Sangjin Hong. New resampling algorithms for particle filters. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, 2003.

[10] Miodrag Bolic, Petar M Djuric, and Sangjin Hong. Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing*, 53(7):2442–2450, 2005.

[11] James Carpenter, Peter Clifford, and Paul Fearnhead. Improved particle filter for nonlinear problems. *IEE Proceedings-Radar, Sonar and Navigation*, 146(1):2–7, 1999.

[12] Huimin Chen, Genshe Chen, Erik Blasch, and Khanh Pham. Comparison of several space target tracking filters. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2009.

[13] Satish Chikkagoudar, Kai Wang, and Mingyao Li. Genie: a software package for gene-gene interaction analysis in genetic association studies using multiple GPU or CPU cores. *BMC research notes*, 4(1):158, 2011.

[14] David E. Culler, Richard Karp, David A. Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. Technical Report UCB/CSD-92-713, EECS Department, University of California, Berkeley, Dec 1992.

[15] Fred Daum and Jim Huang. Particle flow for nonlinear filters with log-homotopy. In *SPIE Defense and Security Symposium*. International Society for Optics and Photonics, 2008.

[16] Fred Daum and Jim Huang. Nonlinear filters with particle flow. In *Proc. SPIE Signal and Data Processing of Small Targets*, volume 7445, 2009.

[17] Fred Daum and Jim Huang. Nonlinear filters with particle flow induced by log-homotopy. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2009.

[18] Fred Daum and Jim Huang. Exact particle flow for nonlinear filters: seventeen dubious solutions to a first order linear underdetermined PDE. In *IEEE 44th Asilomar Conference on Signals, Systems and Computers*, pages 64–71, 2010.

[19] Fred Daum and Jim Huang. A fresh perspective on research for nonlinear filters. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010.

[20] Fred Daum and Jim Huang. Generalized particle flow for nonlinear filters. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010.

[21] Fred Daum and Jim Huang. Numerical experiments for nonlinear filters with exact particle flow induced by log-homotopy. In *Proc. SPIE Signal and Data Processing of Small Targets*, volume 7698, 2010.

[22] Fred Daum and Jim Huang. Angle only tracking with particle flow filters. In *SPIE Optical Engineering+ Applications*. International Society for Optics and Photonics, 2011.

[23] Fred Daum and Jim Huang. Hollywood log-homotopy: movies of particle flow for nonlinear filters. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2011.

[24] Fred Daum and Jim Huang. Particle degeneracy: root cause and solution. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2011.

[25] Fred Daum and Jim Huang. Particle flow for nonlinear filters. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5920–5923, 2011.

[26] Fred Daum and Jim Huang. Fourier transform particle flow for nonlinear filters. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2013.

[27] Fred Daum, Jim Huang, Misha Krichman, and Talia Kohen. Seventeen dubious methods to approximate the gradient for nonlinear filters with particle flow. In *SPIE Optical Engineering+ Applications*. International Society for Optics and Photonics, 2009.

[28] Fred Daum, Jim Huang, and Arjang Noushin. Exact particle flow for nonlinear filters. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010.

[29] Randal Douc and Olivier Cappé. Comparison of resampling schemes for particle filtering. In *Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis*, pages 64–69. IEEE, 2005.

[30] Arnaud Doucet, Nando Defreitas, and Neil Gordon. *Sequential Monte Carlo methods in practice (statistics for engineering and information science)*. Springer, 1st edition, June 2001.

[31] Bradley Efron, Robert Tibshirani, and Robert J. Tibshirani. *An introduction to the bootstrap*. Chapman Hall, 1993.

[32] Paul Fearnhead. *Sequential Monte Carlo methods in filter theory*. PhD thesis, University of Oxford, 1998.

[33] Horace P Flatt and Ken Kennedy. Performance of parallel processors. *Parallel Computing*, 12(1):1 – 20, 1989.

[34] Phillip I. Good. *Resampling methods: A practical guide to data analysis*. Birkhauser, 2006.

[35] Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In *IEE Proceedings F (Radar and Signal Processing)*, volume 140, pages 107–113. IET, 1993.

[36] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message passing interface.* MIT Press, 2nd edition, 1999.

[37] Gustaf Hendeby, Jeroen Hol, Rickard Karlsson, and Fredrik Gustafsson. A graphics processing unit implementation of the particle filter. 2007.

[38] Gustaf Hendeby, Rickard Karlsson, and Fredrik Gustafsson. Particle filtering: the need for speed. *EURASIP Journal on Advances in Signal Processing*, 2010.

[39] W Daniel Hillis and Guy L Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[40] Sangjin Hong, M. Bolic, and P.M. Djuric. An efficient fixed-point implementation of residual resampling scheme for high-speed particle filters. *Signal Processing Letters, IEEE*, 11(5):482 – 485, May 2004.

[41] Sangjin Hong and P.M. Djuric. High-throughput scalable parallel resampling mechanism for effective redistribution of particles. *IEEE Transactions on Signal Processing*, 54(3):1144 – 1155, 2006.

[42] Kai Hwang and Faye Alaye Briggs. *Computer architecture and parallel processing*. McGraw-Hill, 1988.

[43] Kai Hwang and Naresh Jotwani. *Advanced computer architecture*. Tata Mcgraw Hill Education Private Limited, 2010.

[44] Leah H Jamieson, Dennis Gannon, and Robert J Douglass. *The characteristics of parallel algorithms*. Mit Press, 1987.

[45] Vesselin P Jilkov and Jiande Wu. Implementation and performance of a parallel multitarget tracking particle filter. In *Proceedings of the 14th International Conference on Information Fusion*. IEEE, 2011.

[46] Vesselin P Jilkov, Jiande Wu, and Huimin Chen. Parallel particle filter implementation on cluster computer for satellite tracking application. In *Proc. 4th International Conference on Neural, Parallel & Scientific Computations*, 2010.

[47] Vesselin P Jilkov, Jiande Wu, and Huimin Chen. Performance comparison of GPU-accelerated particle flow and particle filters. In *Proceedings of the 16th International Conference on Information Fusion*. IEEE, 2013.

[48] Genshiro Kitagawa. Monte Carlo filter and smoother for non-Gaussian nonlinear state space models. *Journal of computational and graphical statistics*, 5(1):1–25, 1996.

[49] Chris Kreucher, Alfred O III Hero, Keith Kastella, and Mark R Morelande. An information-based approach to sensor management in large dynamic networks. *Proceedings of the IEEE*, 95(5):978–999, 2007.

[50] Chris Kreucher, AO Hero, and Keith Kastella. A comparison of task driven and information driven sensor management for target tracking. In *44th IEEE Conference on Decision and Control*, pages 4004–4009. IEEE, 2005.

[51] Chris Kreucher, Keith Kastella, and Alfred O. Hero III. Tracking multiple targets using a particle filter representation of the joint multitarget probability density. *in Proc. SPIE Conf. Signal and Data Processing of Small Targets*, 5204:258–269, 2003.

[52] Chris Kreucher, Keith Kastella, and Alfred O. Hero III. Sensor management using an active sensing approach. *Signal Processing*, 85(3):607 – 624, 2005.

[53] V.P. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):379 – 391, 1994.

[54] Nosan Kwak, In-Kyu Kim, Heon-Cheol Lee, and Beom-Hee Lee. Analysis of resampling process for the particle depletion problem in fastslam. In *16th IEEE International Symposium on Robot and Human interactive Communication*, pages 200 –205, 2007.

[55] Lee and R.B. Empirical results on the speedup, efficiency, redundancy and quality of parallel computations. In *Proc. Int. Conf. Parallel Processing*, pages 91–96, 1980.

[56] Anthony Lee, Christopher Yau, Michael B Giles, Arnaud Doucet, and Christopher C Holmes. On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods. *Journal of computational and graphical statistics*, 19(4):769–789, 2009.

[57] Cui-Yun Li and Hong-Bing Ji. A new particle filter with GA-MCMC resampling. In *International Conference on Wavelet Analysis and Pattern Recognition*, volume 1, pages 146 –150, 2007.

[58] X Rong Li and Vesselin P Jilkov. A survey of maneuvering target tracking-part VIa: density-based exact nonlinear filtering. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2010.

[59] X Rong Li and Vesselin P Jilkov. A survey of maneuvering target tracking, part VIc: approximate nonlinear density filtering in discrete time. In *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2012.

[60] Jun S. Liu and Rong Chen. Sequential Monte Carlo methods for dynamic systems. *Journal of the American Statistical Association*, 93(443):1032–1044, 1998.

[61] E. Ma and D.G. Shea. *Downward scalability of parallel architectures*. Research Report. IBM T.J. Watson Research Center, 1987.

[62] Clive Maxfield. Programmable logic designline, Xilinx unveil revolutionary 65nm FPGA architecture: the Virtex-5 family, May 15 2006.

[63] Michael D McCool. Signal processing and general-purpose computing and GPUs [exploratory DSP]. *Signal Processing Magazine, IEEE*, 24(3):109–114, 2007.

[64] Joaquín Míguez. Analysis of parallelizable resampling algorithms for particle filtering. *Signal Processing*, 87(12):3155–3174, 2007.

[65] Mark R. Morelande, Christopher M. Kreucher, and Keith Kastella. A study of factors in multiple target tracking with a pixelized sensor. *in Proc. SPIE Conf. Signal and Data Processing of Small Targets*, 5913:155–167, 2005.

[66] M.R. Morelande, C.M. Kreucher, and K. Kastella. A Bayesian approach to multiple target detection and tracking. *IEEE Transactions on Signal Processing*, 55(5):1589 –1604, may. 2007.

[67] MPI Forum. MPI: A message-passing interface standard. Version 2.2, September 4th 2009. available at: `http://www.mpi-forum.org` (Dec. 2009).

[68] MPICH. High-performance portable MPI, September 4th 2014. available at: `http://www.mpich.org/` (Sep. 2014).

[69] L. Murray. GPU acceleration of the particle filter: the metropolis resampler. *ArXiv e-prints*, February 2012.

[70] L. M. Murray, A. Lee, and P. E. Jacob. Rethinking resampling in the particle filter on graphics processing units. *ArXiv e-prints*, January 2013.

[71] David M. Nicol and Frank H. Willard. Problem size, parallel architecture, and optimal speedup. *J. Parallel Distrib. Comput.*, 5(4):404–420, August 1988.

[72] Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Commun. ACM*, 34(3):57–61, March 1991.

[73] NVIDIA. *NVIDIA CUDA C programming best practices guide.* Santa Clara, CA, USA, 2009.

[74] NVIDIA. *CUDA CURAND library.* Santa Clara, CA, USA, 2010.

[75] Branko Ristic, Sanjeev Arulampalam, and Neil Gordon. *Beyond the Kalman filter: particle filters for tracking applications.* Artech House, 2004.

[76] Hartmut F-W Sadrozinski and Jinyuan Wu. *Applications of field-programmable gate arrays in scientific research.* CRC Press, 2010.

[77] A.C. Sankaranarayanan, A. Srivastava, and R. Chellappa. Algorithmic and architectural optimizations for computationally efficient particle filtering. *IEEE Transactions on Image Processing*, 17(5):737 –748, May. 2008.

[78] Simo Sarkka. *Bayesian filtering and smoothing.* Cambridge University Press, 2013.

[79] O.C. Schrempf and U.D. Hanebeck. Recursive prediction of stochastic nonlinear systems based on optimal dirac mixture approximations. In *American Control Conference*, pages 1768–1774, July 2007.

[80] J.Y. Shi, M. Taifi, A Pradeep, A Khreishah, and V. Antony. Program scalability analysis for HPC cloud: Applying Amdahl's law to NAS benchmarks. In *High Performance Computing, Networking, Storage and Analysis*, pages 1215–1225, Nov 2012.

[81] Julian L Simon. *Resampling: The new statistics.* Duxbury Press, 1997.

[82] T.L. Sterling. *Beowulf cluster computing with Linux.* Scientific and Computational Engineering Series. MIT Press, 2002.

[83] Zhimin Tang and Guo-Jie Li. Optimal granularity of grid iteration problems. In *ICPP (1)'90*, pages 111–118, 1990.

[84] V. Teuliere and Olivier Brun. Parallelisation of the particle filtering technique and application to doppler-bearing tracking of maneuvering sources. *Parallel Computing*, 29(8):1069 – 1090, 2003.

[85] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, June 1994.

[86] Michael Joseph Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[87] Gang Wu and Zhenmin Tang. A new resampling strategy about particle filter algorithm applied in Monte Carlo framework. In *2nd International Conference on Intelligent Computation Technology and Automation*, volume 1, pages 507 –510, 2009.

[88] Jiande Wu and Vesselin P. Jilkov. Parallel multitarget tracking particle filters using graphics processing unit. In *Proceedings of the 44th International IEEE Southeastern Symposium on System Theory*, 2012.

[89] Jiande Wu, Vesselin P. Jilkov, and Dimitrios Charalampidis. Implementation and performance comparison of fpga-accelerated particle flow and particle filters. In *SPIE Conference on Signal and Data Processing of Small Targets*, San Diego, CA, USA, August 9 - 13, 2015 (Submitted).

[90] Xilinx. Archived from the original, 2014. available at: `http://www.xilinx.com/fpga/index.htm` (2014).

Vita

Jiande Wu received the B.S and M.S. degree from North China Electric Power University, Beijing, China, in 1998 and 2005, respectively, both in Computer Science & Engineering. He received the M.S. degree from University of New Orleans, New Orleans, USA, in 2012, in Electrical Engineering. From January 2008 to December 2014, he is a research assistant in the department of Electrical Engineering, University of New Orleans, New Orleans, USA. His research interests include parallel processing, nonlinear filtering and target tracking.